INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Factors Influencing a Personal Software Process

Xiaoming Zhong

School of Computer Science

McGill University, Montreal

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science

February 1, 2000

Abstract

Following the development of Personal Software Process (PSP) by Humphrey, a number of efforts have been made to study the impact of PSP on the improvement of software quality and productivity. However, most of such studies have focused on the *results* of the execution of PSP. Little attention has been paid to the underlying factors that influence the output of the execution of PSP. By investigating which factors influence the output, and how, we would have an improved understanding of PSP, which, in turn, could lead to improvement of the design of PSP and hence of personal software process.

In this thesis, we describe an experiment involving 53 subjects carrying out PSP. In particular, we examine the factors underlying a personal software process and analyzed the impact of these factors on the output of PSP execution, namely, the improvement in software quality and productivity. Our study complements previous work on PSP by providing a "white-box" view of PSP in assessing the effectiveness of PSP.

Résumé

À la suite du développement du "Personal Software Process" (PSP) par Humphrey, plusieurs ont étudié l'impact du PSP sur l'amélioration de la productivité et de la qualité des logiciels. Cependant, la plupart de ces études se sont concentrées sur les *résultats* de l'exécution du PSP. Il y a eu très peu d'attention portée sur les facteurs influençant ces résultats. En examinant quels facteurs ont une influence sur le résultat, et comment ils les influencent, nous aurions une meilleure compréhension du PSP. Ceci pourrait mener à l'amélioration de la conception du PSP, et donc du processus personnel de développement de logiciel.

Dans cette thèse, nous décrivons une expérience où 53 sujets ont utilisé le PSP. En particulier, nous examinons les facteurs à la base du processus personnel de développement de logiciel, et nous analysons leur impact sur les résultats d'exécution du PSP, c'est-à-dire sur l'amélioration de la productivité et de la qualité des logiciels. Notre étude est complémentaire aux autres travaux sur le PSP, en apportant une vue de "boîte blanche" sur le PSP dans l'évaluation de son efficacité.

Acknowledgments

I wish to thank my supervisor, Prof. Madhavji, for his guidance and support, and his valuable suggestions and criticism during the course of this work. I also wish to thank my family for their continuous support and encouragement throughout my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is well known that software development projects are plagued with quality problem, cost overruns and schedule slippage [Neu93]. Thus, among the key goals of software development are to be able to build high quality software systems, on time, within budget and within a satisfactory development environment. A recently recognized way to help achieve these goals is to improve software development processes such that the concerned problems are mitigated. Various approaches to process improvement have been developed ranging from those at the personal-level to those at organizational-level: Personal Software Process (PSP) [Hum95a]; Capability Maturity Model (CMM) [PCCW93] [Hum89]; Quality Improvement Paradigm (QIP) [Bas92]; and Total Quality Management [Zul93]. Based on such models, many process improvement projects and activities have been carried out both in academia and the software industry [HT97] [ESM96] [Sho96] [Hum95b] [SM96] [She94] [Dio93]

[Woh93] [HSW91].

Focusing at the grass-roots level of software development is Humphrey's Personal Software Process (PSP) [Hum95a]. It is based on the premise that discipline in software development at the personal level can help increase the effectiveness of individual engineers [Hum94a] [Hum93]. This, in turn, is likely to improve the performance of software teams and projects.

PSP is a process framework and a set of methods designed to help engineers to be more disciplined in their work. It shows them how to estimate the size of individual projects (or tasks); plan their projects; measure and track their work; and improve the quality of the products they produce.

The PSP consists of: a series of scripts that define tasks; forms for recording data; and standards that govern such aspects as coding practices, size counting, and the assignment of defect types. When engineers follow PSP, they first plan their work and document the plan. As they do their work, they record development time and track and record every defect they find. At the end of the project, the engineers do a postmortem analysis and complete a project plan summary report.

PSP is being taught as a course in a number of universities, among them: the University of Massachusetts (at Lowell) [Hum95b], Carnegie Mellon University [HT98] [Hum94b], Embry-Riddle Aeronautical University [HT97] [Kha95], and McGill University [SM96] [She94]. PSP courses or concepts have also been used to train professional engineers in the software industry [FHK+97] [ESM96] [MKN+96] [Roy96]

[Hum94b] [Hum94c].

The introduction of a new software development method, however, is often accompanied by the assessment of the effectiveness of this method. Following the development of PSP, a number of efforts have been made to study the impact of PSP on software development [HT98] [FHK+97] [ESM96] [Hum96a] [Hum96a] [She94]. Their research helps to explain the logic behind PSP and contributes greatly to any possible refinement of PSP.

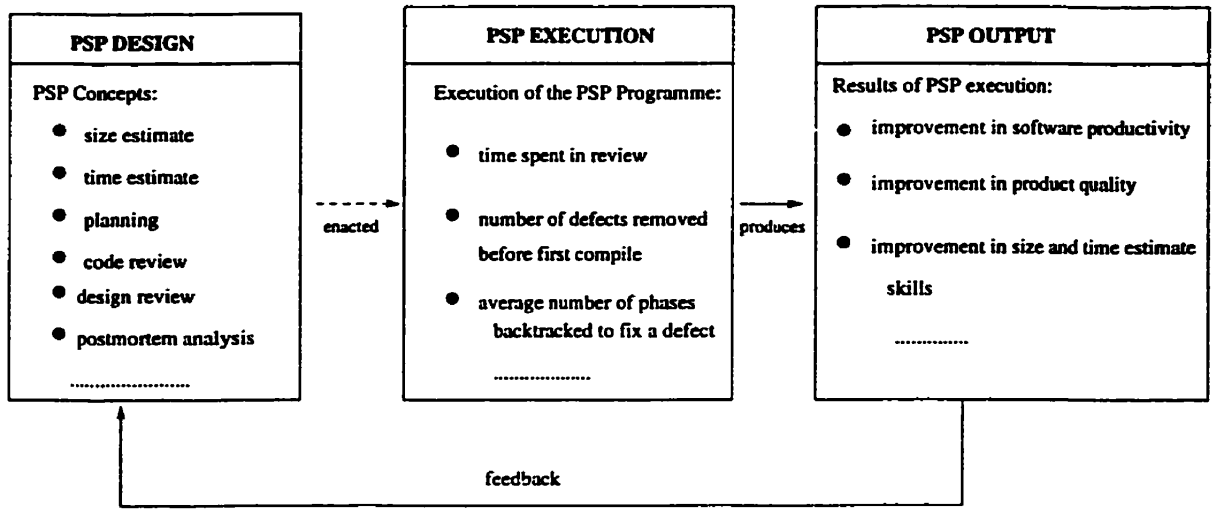


Figure 1.1: A Schematic Diagram Showing the PSP Design Concepts, PSP Execution, PSP Output, and Feedback

Most of the published research, however, has focused on the PSP concepts and the output of PSP execution, e.g., the relationships between code review and improvement in product quality (Figure 1.1). Such research helps to explain the degree to which PSP improves an individual's process. It is more like a *static analysis* of

PSP relationships. But many underlying factors, observed during PSP execution, that influence the output of PSP have received little attention. This thesis thus complements previous work on PSP by focusing on the dynamics of PSP, giving insight into the underlying factors for the observed effects in the output of PSP execution. Such insight can be used to improve specific aspects of PSP.

A personal software process involves many underlying factors, such as *the percentage of defects removed before the first compilation, time spent in test phase as a percentage of total development time*, etc. It is clear from previous studies that PSP, as a whole, has a positive impact on certain dependent variables of software development, such as *number of defects removed per hour, thousand lines of code developed per hour*, etc. But there is no reason to assume that all the underlying factors contribute equally to the dependent variables. We assume that certain underlying factors play a key role in the impact of PSP on the dependent variables. Which factors are there is not clear, however, the purpose of this study is to uncover the influence factors.

Thus, in this study, we built descriptive models of software quality and productivity. Based on these models, we investigated the factors, e.g., *average number of phases backtracked to fix a defect*, which most influence process improvement measured in terms of: *LOC per hour, number of defects per KLOC* and *the number of defects removed per hour*. We then studied how these factors vary through the PSP execution, so that we can have an improved understanding of the way in which PSP

affects software productivity and quality.

Our study was carried out in the context of a PSP course at McGill University. Fifty three full-time senior undergraduate students participated in the course during the period September to December 1997. Every week, two lectures, each of duration one and a half hours, were given to the students in which they were taught ways to carry out and improve their personal processes. After each set of lectures, the students were assigned a programming project, which utilized the techniques taught to them. There were a total of eight [1] programming projects, per student, which all used the C++ programming language. Based on the data gathered from these (53 x 8 = 424) projects, we had a number of findings [2], for example:

1. The percentage of defects found and fixed before the first compile (Yield) is a significant factor that influences process improvement measured in terms of: (i) defects per delivered thousand lines of code (Defect Density), (ii) defects found and removed per hour (Defect Removal Rate), (iii) and lines of code developed per hour (LOC/Hour).

   Our analysis indicates that, when Defect Density is used to evaluate software quality, yield should be taken into account. We also found that one of the

[1] In addition, there were three significant data analysis projects, probing into problems typically faced in software projects.

[2] The interpretation of these results are described in the later chapters of the thesis where details of the data analysis are also given.

obstacles of introducing review skills is because reviews have no significant positive effect on Defect Removal Rate unless Yield has achieved a high level.

2. Appraisal Time to Failure Time ratio (A/FR): the time spent in design review or code review as a percentage of the time spent in compile and test – is a significant factor that influences Defect Removal Rate and LOC/Hour.

   Our analysis indicates that the A/FR value can be a useful guide for software developers in adjusting their review time to achieve a high Defect Removal Rate during software development.

This thesis describes several other findings. By studying the relationships among the PSP concepts, process dynamics and the output of PSP execution (see Figure 1.1), our study gives a new insight into the rationale and logic behind PSP. In return, this can lead to possible refinements of certain aspects of PSP.

Chapter 2 gives the details of PSP. Chapter 3 states our research objective and experiment design. An in-depth analysis of data is done and results are shown in chapter 4. Chapter 5 concludes our study.

Chapter 2

Background

This section presents background work which is related to our study. It briefly describes Personal Software Process (PSP)[Hum95a]. This section also provides an overview of the basic PSP measures, forms, and measurement process to give the reader some context for the data that were analyzed for this study. Finally, some reported experiments of PSP were reviewed.

2.1 What Personal Software Process Is

This section describes the development of, the logic behind, the structure of, and the evolution of Personal Software Process.

2.1.1 Development of Personal Software Process

In 1987, the Software Engineering Institute (SEI) at Carnegie Mellon University published a software development capability maturity known as the Capability Maturity Model (CMM) [Hum87]. By establishing and defining the five levels of progressively more-mature process capability, the CMM provides an orderly way for organizations to determine the capabilities of their current process and to establish priorities for improvement. Its focus, however, is on large-scale software and large-scale software organizations. Humphrey recently extended the CMM by proposing an approach for scaling it down to small teams and small software organizations through Personal Software Process (PSP) paradigm [Hum95a]. PSP addresses the need to use CMM by individual software practitioners and small software organizations. It relies on a bottom-up approach rather than a top-down one: instead of imposing software processes from a managerial direction and targeting projects, PSP focuses on the individual programmer and targets his/her work.

2.1.2 The Logic Behind Personal Software Process

PSP provides a framework to help software developers to organize and plan their work, track their performance, manage software defects, and analyze and improve their personal process. The logic behind PSP is as follows:

- By defining, measuring, and tracking their work, software developers will better understand what they do.
- This understanding will enable the software developers to better recognize what methods work best for them and to see how they can more consistently apply them.
- The engineers will then have a defined process structure and measurable criteria for evaluating and learning from their own and others' experiences.
- With this knowledge, the software developers can select those methods and practices that best suit their particular tasks and abilities.
- By using a customized set of orderly, consistently practiced, and high quality personal practices, the software developers will be more effective members of their development teams and projects.

2.1.3 The PSP Approach and Structure

At the technical level, PSP is a structured set of process descriptions, measurements, forms, scripts and standards that guide software developers in size estimation, planning, reviewing and data gathering. Various data analyses are defined to determine the quality and productivity of software developers' work. This helps individuals to develop a quantitative understanding of their process products and processes.

PSP has a maturity framework as does the CMM. It has been structured in an evolving sequence of seven upward-compatible personal processes. Each process step is defined and used to guide the individuals through an evolutionary path from simple process concepts, such as project planning, to advanced levels of process maturity, such as defect prevention. The PSP evolutionary path consists of seven consecutive phases (see Figure 2.1). Appendix A contains a brief discussion of these seven PSP phases.

2.2 The PSP Measures

PSP is based on the principle of data-driven process improvement whereby measurements are central in highlighting process deficiencies and providing a focus for process improvement. There are three basic measures in PSP: the development-time, software defects, and software size. All other PSP measures are derived from these three basic measures. The measurement process and forms [1] for these measures are introduced during the first three assignments at the PSP process levels PSP0 and PSP0.1. Development-time and defect measures are introduced on the first assignment; size measures are deferred until a program for counting LOC has been developed in assignment 2.

[1] Some measurement definitions and templates sample in this section are from [Hum95a] [HO97].

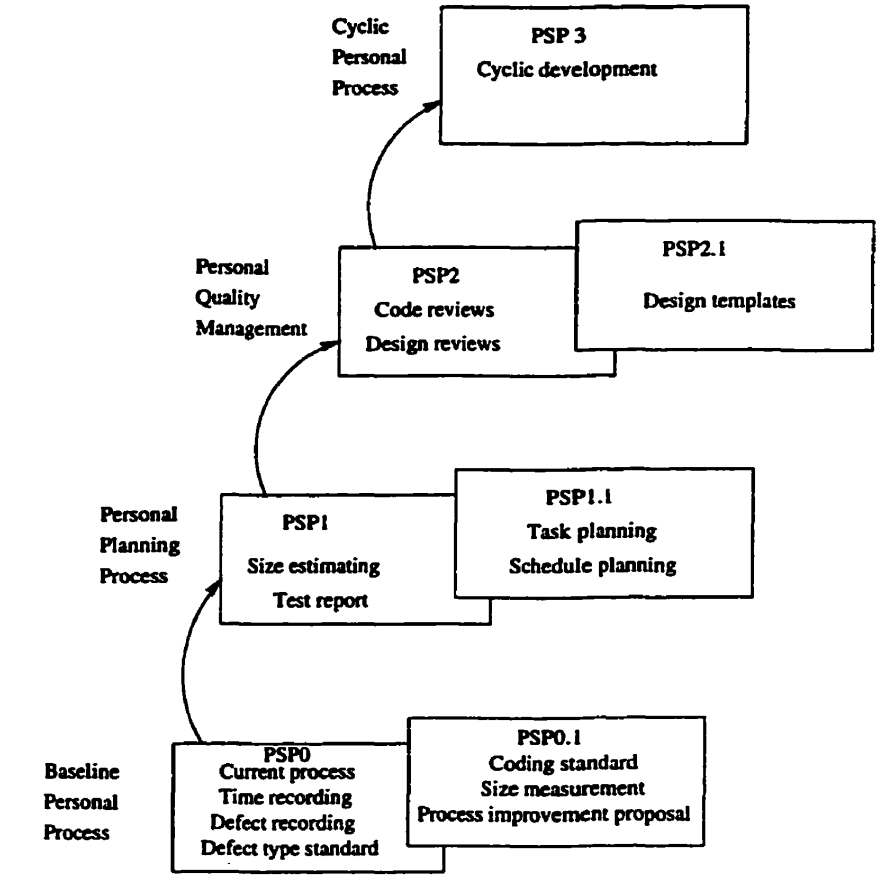


Figure 2.1: PSP Evolution

2.2.1 Development-time Measurement

Minutes are the unit of measure for development-time. Engineers track the number of minutes they spend in each PSP phase, less time for any interruptions such as phone calls, coffee breaks, etc. A form, the Time Recording Log, is used to record development-time.

The example Time Recording Log (Figure 2.2) illustrates how this form is used.

In the example, the engineer started the Plan phase of his project on Sept. 15 at 7:58 and finished planning at 8:45. The elapsed time was 47 minutes, but actual effort, or Delta Time, was only 44 minutes, due to an interruption of three minutes to take a phone call. The engineer started the Design phase at 8:47 and finished at 10:29. A two-minute interruption give a Delta Time of 100 minutes. The remaining phases, Code, Compile, and Test are recorded in a similar manner.

Date	Start	Stop	Interruption Time	Delta Time	Phase	Comments
9/15	7:58	8:45	3	44	Plan	phone call
	8:47	10:29	2	100	Design	create, review design
	7:49	8:59		70	Code	coded functions
	9:15	9:45		31	Compile	compiled and linked
	9:47	10:10		23	Test	ran tests A, B, and C
	4:33	4:51		18	Postmortem	

Figure 2.2: Time Recording Log

2.2.2 Defect Measurement

A defect is defined as any change that must be made to the design or code in order to get the program to function as desired. Defects are recorded on the Defect Recording Log as they are found and fixed. The example Defect Recording Log (Figure 2.4)

shows the information that is recorded for each defect: the date, sequence number, defect type, phase in which the defect was injected, phase in which it was removed, fix time, and a description of the problem and fix.

When an engineer injects a new defect while trying to fix an existing defect, proper accounting of fix time can become more complicated. A common mistake is to include the fix time for the new defect twice. To help with this problem, some space is provided to record a reference to the original defect that was being fixed. The number of the original defect that was being fixed is recorded in the fix defect reference of the new defect.

Each defect is classified according to a defect type standard as described in [Hum95a]. The standard includes 10 defect types (Figure 2.3) in a simple, easy-to-use classification scheme designed to support defect analysis. Engineers can refine the standard to meet personal needs, but they are encouraged to wait until they have sufficient data to justify a change.

In the example Defect Recording Log (Figure 2.4), the engineer found the first defect on Sept. 13. The defect was a type 20 (syntax error) that was injected during the code phase and removed during the compile phase. The engineer spent 22 minutes finding and fixing the defect. The second error, also a syntax error, was injected during the code phase and removed during the compile phase, and took 18 minutes to find and fix. Similarly, other defects are recorded in the log.

Type Number	Type Name	Description
10	Documentation	comments, messages
20	Syntax	spelling, punctuation, typos, instruction formats
30	Build, Package	change management, library, version control
40	Assignment	declaration, duplicate names, scope, limits
50	Interface	procedure calls and references, I/O, user formats
60	Checking	error messages, inadequate checks
70	Data	structure, content
80	Function	logic, pointers, loops, recursion, computation, function defects
90	System	configuration, timing, memory
100	Environment	design, compile, test, or other support system problems

Figure 2.3: Defect Type Standard

2.2.3 Size Measurement

The primary purpose of size measurement in PSP is to provide a basis for estimating development-time. Lines of code were chosen for this purpose because they meet the following criteria: they can be automatically counted, precisely defined, and are well correlated with development-effort based on the PSP research [Hum95a]. Size is also used to normalize other data, such as productivity (LOC per hour) and

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
9/13	1	20	CODE	CMPL	22	

Description: syntax error in scanf statement

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
9/13	2	20	CODE	CMPL	18	

Description: error in linked list struct type declarations within access functions

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
9/13	3-6	20	CODE	CMPL	1	

Description: missing;

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
9/13	7	20	CODE	CMPL	1	

Description: incorrect spelling of identifier in declaration

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
9/13	8	20	CODE	CMPL	1	

Description: function declaration error

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
9/13	9	20	CODE	CMPL	1	

Description: link error, missing include for math.h

Figure 2.4: Defect Recording Log

defect density (defects per KLOC). While LOC are suitable for the programming assignments in the PSP course, any measure that meets these same criteria can be used in practice.

In the PSP course, as in practice, each program involves some amount of new development, enhancement, and/or reuse. Therefore, the total LOC in a program will have several different sources, including some new LOC, some existing LOC that my have been modified, and some reused LOC. Because LOC are the basis for

estimates of development-time, it is important to account for these different types of LOC separately.

PSP uses the LOC accounting scheme shown in Table 2.1. Base LOC are any LOC from an existing program that will serve as the starting point for the program being developed. Deleted and modified LOC are those base LOC that are being deleted or modified. Added LOC is the sum of all newly developed object, function, or procedure LOC, plus additions to the base LOC. Reused LOC are the LOC taken from the engineer's reuse library and used without modification. If these LOC are modified, then they are considered to be base LOC.

New and changed LOC are the sum of added LOC and modified LOC. New and changed LOC, not total LOC, are the most commonly used size measure in PSP. For example, new and changed LOC are the basis for size and effort estimating, productivity (LOC/Hour), and defect density (Defects/KLOC). Finally, total LOC are the total program size, and total new reused LOC are those added LOC that were written to be reused in the future.

2.2.4 Project Summary Data

Project summary data are recorded on the Project Plan Summary form. This form provides a convenient summary of planned and actual values for program size, development time, and defects, and a summary of these same data for all projects completed to date. Figure 2.5 shows the four sections of the project plan summary

Type of LOC	Definition
Base	LOC from a previous version
Deleted	Deletions from the Base LOC
Modified	Modifications to the Base LOC
Added	New objects, functions, procedures, or any other added LOC
Reused	LOC from a previous program that is used without modification
New and Changed	The sum of added and modification LOC
Total LOC	The total program LOC
Total New Reused	New or added LOC that were written to be reusable

Table 2.1: PSP LOC Type Definitions

that were used: Program Size, Time in Phase, Defects Injected, an Defects Removed.

The data on the plan summary form has many practical applications for the software engineer. The data can be used to track the current project, as historical data for planning future projects, and as baseline process data for evaluating process improvements.

2.2.5 PSP Derived Measures

Each PSP level introduces new measures to help engineers manage and improve their performance. These measures are derived from the three basic PSP measures: development-time, defects, and size.

Program Size (loc):	Plan	Actual	To Date
Base(B)	0 (Measured)	0 Measured)	
Deleted(D)	0 (Estimated)	0 Counted)	
Modified(M)	0 (Estimated)	0 (Counted)	
Added(A)	112 (N-M)	137 (T-B+D-R)	
Reused(R)	174 (Estimated)	174 Counted)	316
Total New & Changed(N)	112 (Estimated)	137 (A+M)	759
Total LOC (T)	286 (N+B-M+D-R)	311 Measured)	1161
Total New Reused	0	0	326

Time in Phase (min.)	Plan	Actual	To Date	To Date %
Planning	42	44	287	17.2
Design	76	56	490	29.4
Design review				
Code	48	61	337	20.2
Code review	30	27	27	1.6
Compile	15	1	106	6.4
Test	26	38	326	19.6
Postmortem	13	20	94	5.6
Total	250	247	1667	100.0

Defects Injected	Plan	Actual	To Date	To Date%
Planning	0	0	0	0.0
Design	2	1	10	18.2
Design review	0	0	0	0.0
Code	6	6	40	72.7
Code review	0	0	0	0.0
Compile	0	0	2	3.6
Test	0	0	3	5.6
Total Development	8	7	55	100

Defects Removed	Plan	Actual	To Date	To Date %
Planning	0	0	0	0.0
Design	0	0	0	0.0
Design review	0	0	0	0.0
Code	0	0	1	1.8
Code review	0	5	5	9.1
Compile	4	0	24	43.5
Test	4	2	25	45.5
Total Development	8	7	55	100.0
After Development	0	0	0	

Figure 2.5: Sample PSP Project Plan Summary Form

Measure	Definition
Interruption Time	The elapsed time for small interruption from project work
Delta Time	Elapsed time in minutes from start to stop less interruptions
Compile Time	The time from the start of the first compile until the first clean compile
Test Time	The time from the start of the initial test until test completion
Total Time	The sum of the time for all phases of a project
Defect	Any element of a program design or implementation that must be changed to correct the program
Defect type	See Figure 2.3, Defect Type Standard
Fix Time	The time to find and fix a defect
LOC	A logical line of code as defined in the engineers' counting and coding standard
LOC/Hour	Total new and changed LOC developed divided by the total development hours
Compile Defects/KLOC	The defects removed in compile per new and changed KLOC
Test Defect/KLOC	The defects removed in the test phase per new and changed KLOC
Defect Density	The total defects removed per new and changed KLOC
Yield	The percentage of defects injected before the first compile that removed before the fist compile
Appraisal Time	Time spent in design and code reviews
Failure Time	Time spent in compile and test
A/FR	Appraisal Time/Failure Time

Table 2.2: Definitions of PSP Measures

Table 2.2 contains a partial list of the derived measures available in PSP and their definitions. These measurement definitions are included to provide context for the analyses that follow.

2.3 The assessment of PSP

Because the research on PSP is in its infancy, the experience with PSP is still limited. Several universities offer a course on PSP as a part of their software engineering programs. The reports from these courses are generally positive [HT98] [Hum95c] [She94]. Concerning PSP in industry, only a few reports are available at this stage [FHK+97] [ESM96]. In the following two sub-sections, we describe PSP in academia and industry.

2.3.1 PSP in the Classroom Setting

A one-semester graduate-level PSP course has been developed with a textbook [Hum95a], a series of exercises, and instructor's guide. This course is being adopted by a number of universities in the USA, Canada, Europe and South America. A number of PSP training experiments have been done since PSP was first introduced in 1993. In [Hum95b][2], Humphrey summarizes the results from five early completed PSP courses taught at university of Massachusetts (at Lowell), Carnegie-Mellon Univer-

[2]The data analyzed in this thesis is not to be confused with PSP data from McGill University

sity, Bradly University, McGill University, and Embry Riddle Aeronautical University. Substantial improvement in quality and productivity has been reported based on the data gathered from these five courses (see Table 2.3 and Table 2.4).

University	Bradly	CMU	ERAU	McGill	Umass
Number of students	6	12	19	12	4
% Reduction in compile defects	75.7	76.6	88.1	76.7	68.8
% Reduction in test defects	64.2	81.7	83.2	76.7	68.8
% Reduction in total defects	55.1	45.8	80.1	64.0	53.4

Table 2.3: Quality Improvement in the Five Early Courses

Table 2.3 shows the percentage improvement in the defects per KLOC found in compile, in test, and in total during the PSP exercise. These data compare the averages of the first two exercises with those of the last two.

Average Loc/Hour	Bradly	CMU	ERAU	McGill	UMass
The first two exercises	11.4	31.4	13.8	21.1	19.9
The last two exercises	26.9	38.6	22.3	29.3	36.6
Percent Improvement	136.0	22.9	61.6	38.0	82.4

Table 2.4: Productivity Improvement in the Five Early Courses

Table 2.4 shows the productivity increases were significant considering the last two exercises are substantially larger and more difficult than the first two and engi-

neers completed them with lower defect levels.

PSP addresses a number of desirable software engineering behaviors. Starting with PSP0 and extending through PSP1.1, the engineers learn and practice a series of measurement and planning methods. Another behavior addressed by PSP is software design. PSP2 addresses design by introducing design completion criteria to reduce redoing the design during coding. In [Hum95c], which aims to investigate how the PSP impacts people and how it changes their behaviors, it is claimed that the PSP training impacts the performance of engineers and students in a classroom environment. This claim is supported by the data gathered from a class taken by 43 engineers and graduate students. The performance is evaluated in terms of the number of defects per thousand lines of code, yield, defect removal rate, and estimate accuracy. It is reported that by completing the prescribed PSP exercises and following the defined processes, engineers can reduce their number of defects by 75% or more. While results vary considerably among engineers, on average they achieve productivity improvements of 25% or more.

An experiment using PSP was conducted at McGill University to study personal "progress function" in software process [She94]. Data from the 12 senior undergraduate students shows that, on average, learning takes place at a rate of 20%. Of this, second-order learning (or organizational training) amounted to approximately 13% more; whereas first-order learning (or self-learning) amounted to approximately 7%. In this study, detailed statistical methods were used to produce linear and log-

linear models of high correlations, involving four variables: productivity, defect-rate, complexity, and cumulative output.

In the early experiments, PSP was taught almost exclusively to practitioners or graduate students. However, more recently, PSP has been introduced at the undergraduate level in some universities. Hou et al.[HT98] conducted an experiment in applying components of PSP to 65 CS1 students. The results of their experiments show that PSP is also of value to novices of CS1 level, regardless of their background. It is reported that there is a steady drop in the amount of time spent on compile (from 15% of total time for assignment 6 to 12% of total time for assignment 8) and huge improvement in the errors in first compile (from 14% of total errors for assignment 5 to 7% of total errors for assignment 8). Compile time and compile error were primarily concerned because these two factors were believed to be "major hurdles" in software developing for a novice.

In a recent project [HT97] in Embry-Riddle Aeronautical University, PSP concepts were introduced into CS1 and CS2. Primarily, the impact of the PSP on their estimation skills and the quality of their work was investigated. Results for individual students show a lot of variation in their estimation errors. Thus, no conclusions about the overall estimation skills of the student group can be made. However, defect density and percent of effort spent in the compile and test phases were substantially decreased (see Figure 2.6 and Figure 2.7).

Figure 2.6 shows the defect density (the number of defects per thousand lines

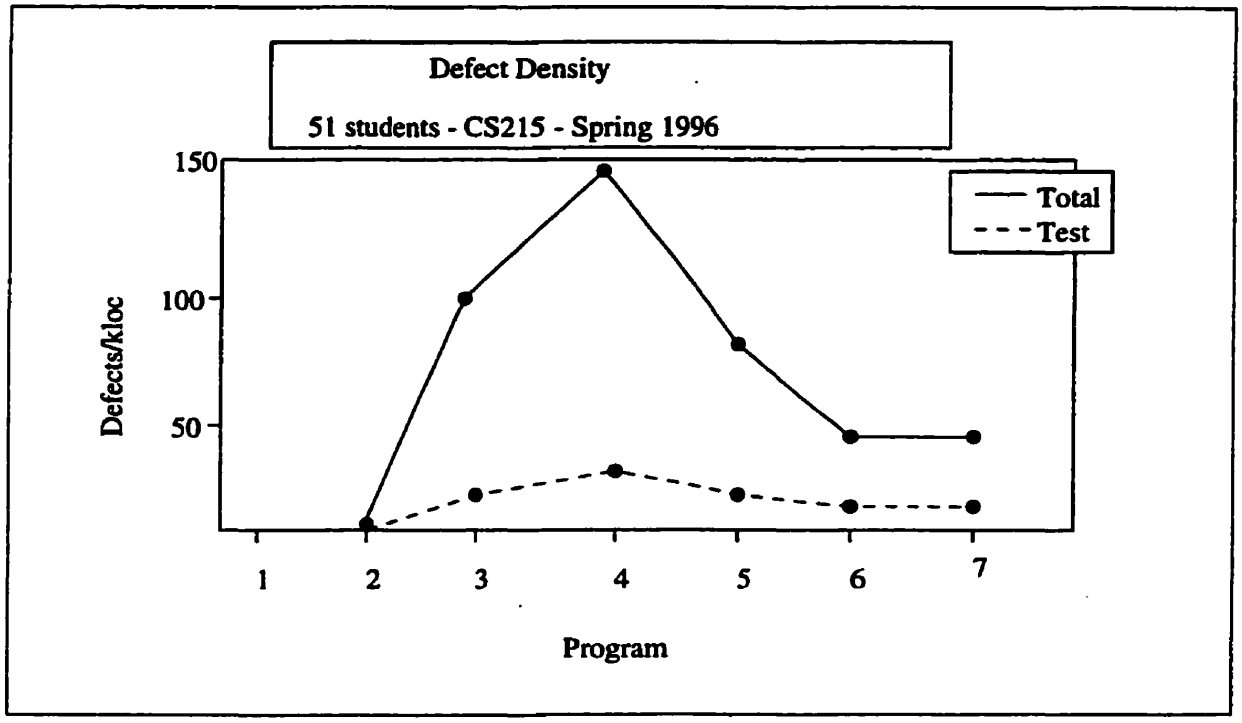


Figure 2.6: Defect Density – from CS215 in ERAU

of code) for program 3 through 7 (defects were not recorded for program 1 and 2). There is a clear drop in defects for the last three programs, both for total defects and for defects found in test.

Another indication of improvement in the development process is the decline of the percent of effort spent in the compile and test phases, shown in Figure 2.7.

2.3.2 PSP in Industry

PSP is also of interest to industry as a means of training their software engineers. While there are published reports on the teaching of PSP in classroom setting (at

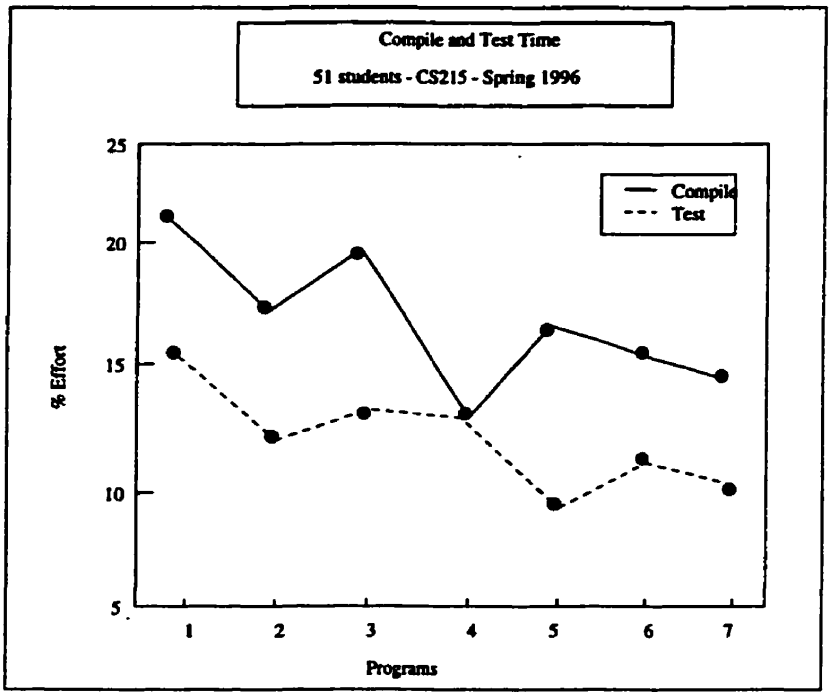


Figure 2.7: Compile and Test Time – from CS215 in ERAU

universities and industry), relatively little data on its use and effectiveness in industry are available. This problem is, in fact, due to the time required to introduce the PSP into a work place and by the length of many software development efforts. However, PSP courses have been used successfully to train professional engineers in several organizations. Advanced Information Services (AIS) Inc., Motorola Paging Products Group, and Union Switch & Signal (US&S) Inc. have trained several groups of engineers and measured the results of several projects that involved PSP [FHK+97]. In all the companies, the projects were a part of the normal operations and not designed for PSP study.

The three companies offered a variety of situations useful for demonstrating the

versatility of PSP. The projects at Motorola and US&S involved software maintainance and enhancement, while those at AIS involved new product development and enhancement. Among the companies, application areas involved commercial data processing, internal manufacturing support, communications products support, and real-time process control, and work was done in program language C or C++.

Project	PSP Staff	Non-PSP Staff	Product Size	Delivery:Planned/ Actual(months)	Acceptance Test Defects	Usage Defects
B	3	0	24 requirement	7/5	1	0
C	0	3	19 requirement	2/5	11	1
D	0	3	30 requirement	10/19	6	14
E	1	0	2,225 LOC	6/6	0	0
F	1	0	1,400 LOC	2/2	0	0
G	2	1	6,196 LOC	2/2	0	3

Table 2.5: Summary of AIS Project Data

Table 2.5 shows the data from AIS. The failure of Project A led to the introduction of PSP in AIS. So it is not listed in the table. Most of the projects involved one to three engineers. In a group, some of the engineers may received PSP training, the others may not. For example, Project G involved 3 engineers. Two of whom were PSP-trained staffs, while one of whom was non-PSP staff. This table shows the performance of different teams with different percentage of PSP staff and non-PSP staff.

Table 2.6 shows the data from 5 projects completed by PSP-trained engineers at US&S. All of the five projects were maintenance and enhancement releases for a large railroad information and control system, and each project required only one engineer. As shown in this table, no defects have been found in any project during installation or customer use.

Table 2.7 shows the results of 18 projects completed by PSP-trained engineers at Motorola. It is noteworthy that several of these products have been used for many months, and only one defect has been found in one of the products.

Product	Lines of Code	Months of Use	Defects in Test	Defect in Use
M45	193	9.0	4	0
M10	453	7.5	2	0
M77	6,133	4.0	25	0
M54	477	3.5	5	0
M53	1,160	1.0	21	0
Total	8,416	NA	57	0

Table 2.6: US&S Usage Data

Using Action Research empirical method [Che69] [Cla72], El Emam et al. [ESM96] carried out a joint study between McGill University and CAE Electronics Ltd. of the implementation of the PSP concepts at CAE Electronic Ltd., a leading supplier of flight simulators located in Montreal, Canada. In their study, two evaluations

were conducted. The first was the transfer of training defined as " the effective, and continued application to trainees' jobs of the knowledge and skills gained in training" [Gar93]. The authors claim that seven months after the start of the study, 46.5% of the participants were still using PSP concepts in their real programming tasks. The second evaluation was of the benefits of the training. The trends in productivity, defect density, yield, and the percentage of time spent on test were used as a measurement of the improvement due to PSP. In their study, the authors did not find any statistically significant trends in productivity. Defect density, however, increased from approximately 88 defects/KLOC to 256 defects/KLOC after the code review lecture. It was also found that average yield increased from approximately 12% before code reviews to 27.7% when code reviews were used and the percentage of time spent on testing dropped from approximately 37% before code reviews to approximately 17% afterwards.

Project Number	Size (LOC)	Months Used	Total Defects	Test Defects	Usage Defects
1	463	18	13	5	6
2	4,565	NA	69	10	0
3	1,571	NA	47	8	0
4	3,381	NA	69	22	0
5	5	9	0	0	0
6	22	5	2	0	0
7	1	18	1	0	0
8	2,081	10	34	0	1
9	114	8	15	2	0
10	364	NA	29	2	0
11	7	5	0	0	0
12	620	3	12	2	0
13	720	NA	9	2	0
14	3,894	NA	20	2	0
15	2,075	NA	79	27	0
16	1,270	NA	20	2	0
17	467	NA	17	3	0
18	3,494	8	139	50	0
Total	25,114	NA	575	136	1

Table 2.7: Motorola Operational Defect Data for PSP Projects

Chapter 3

Research Objective and Experiment Design

This section first defines the objective of this study followed by, in section 3.2, the design of the experiment. The experiment design describes the investigation approach, the research context and the descriptive models. Following the section on experiment design is section 3.3. This section describes the model for analysis of data.

3.1 Research Objective

Each programming assignment results in some 70 pieces of data being collected by each engineer. The data are used by the engineers to monitor their work on

the individual assignments as well as to analyze their personal software process for improvement decisions. The data are also a primary source for researchers to perform secondary analysis.

The PSP data characterize the attributes of the software process of an engineer and his/her product. The data collected during the execution of PSP also show the trends of these attributes through the PSP projects. Some attributes, such as *LOC per hour, number of defects per KLOC* and *number of defects removed per hour*, are often used to evaluate the effectiveness of PSP because these attributes are also used to assess an individual's performance. For example, the improvement of defects removed per hour usually indicates the improvement of the quality of an engineer's process. Also, how these attributes vary over time through the execution of PSP directly reflects the effectiveness of PSP. Thus, these attributes have received particular attention in published research. We define these attributes as *Process Improvement Attribute* in this thesis. The formal definition is given in 3.3.1.

Some other attributes, such as *the time spent in design review or code review as a percentage of the time spent in compile and test, average number of phases backtracked to fix a defect*, and *number of defects per KLOC found in test*, do not reflect an individual's performance directly but may have a great impact on the improvement of the engineers' performance. Many PSP techniques, such as *code review*, focus on changing such attributes and thus change the engineers' performance indirectly. We define such attributes as *Significant Attribute*. The formal definition

is given in 3.3.1.

Researchers in different organizations have employed various scientific methods to explore, from different aspects, the costs and benefits of PSP, based on the data gathered through PSP training. However, as reviewed in section 2.3, most of the published reports focus on the analysis of Process Improvement Attributes, such as *defects removed per hour, number of defects per KLOC* and *LOC/Hour.* That is, it is *black-box analysis.* Other attributes, such as *average number of phases backtracked to fixed a defect* and *the time spent in test phase as percentage of total development time*, are often neglected or are not systematically studied, even though these attributes detail many important aspects of a software process and product.

In this study, we built descriptive models of software quality and productivity which were used for quantitative analysis. Based on these models, we systematically studied various process attributes. Specially, by studying the relationships between Process Improvement Attributes and the other attributes, we identified the Significant Attributes, which influence Process Improvement Attributes the most. To achieve a better understanding of the impact of Significant Attributes on Process Improvement Attributes, we also established the noticeable relationships between Significant Attributes and Non-significant Attributes. Finally we studied how the Significant Attributes vary over time through the PSP execution. Thus, our study provides a way to systematically study the details of a personal software process.

3.2 Experiment Design

This section first introduces our investigation approach. Then it describes the research context. Finally the descriptive models are given.

3.2.1 Investigation Approach

We used a four-step method involving: measurement, modeling and analysis, as outlined below:

1. *Set the basis for quantitative analysis.* We built several descriptive models: process model used in the projects, defect-quality model and development-productivity model. These models lay the foundation for our analysis of personal process and products. We show these models in section 3.2.3.

2. *Define Goal/Question/Metric (GQM) models* [Bas84]. We specified the structure of the analysis by defining specific goals, questions, and metrics using the GQM models. We show these models in section 3.3.2.

3. *Identify the influential attributes.* In this step, we determined which attributes influence process and product improvement the most, using GQM models specified in step 2 and the descriptive models defined in step 1. The influential attributes are shown in section 4.1.

4. *Analyze the trends of the influential attributes, over time, through PSP projects.*

   This is shown in section 4.3.

This four-step method is logically sound because, it proceeds from laying the foundational pieces of the study in step 1 (namely, the descriptive models) to questioning about specific attributes in step 2, based on the foundational pieces. Only then, one can be confident with identifying influential attributes in step 3, and subsequent analyzing their trends in step 4.

3.2.2 Research Context

The PSP course integrated in this study is a senior undergraduate course entitled "Personal Software Engineering" taught at McGill University in the Fall of 1997. A textbook by Humphrey [Hum95a] was used as a basis for this course.

Subjects and background knowledge

Fifty-three full-time students had enrolled in this course. Also, at entry they already had exposed to object-oriented paradigm and software engineering, which permeates the course material. For example, size estimation, software design and related templates are all based on object-oriented software development. We required the use of programming language C++ for implementing programs. In addition, the students had at least a basic knowledge of probability and statistics since these topics are an essential part of PSP's measurement framework.

Programming Tasks

Every week, two lectures, each of duration one and a half hours, were given to the students in which they were taught ways to analyze and improve their personal process. After each set of the lectures, the students were assigned a programming project, which utilized the techniques taught to them, so that the students could actually implement those methods, and hence learned them. There were a total of eight programming projects and three analysis projects per student, through the course, making a total of 424 projects for all the students. All the eight projects are described in the textbook in the "A Series" exercises [Hum95a]. The three analysis projects were a local enhancement.

Data Collection

Perhaps the most critical issue in teaching the PSP course is to get across the message to the students that the course is not just a set of programming tasks. Thus, the students were told repeatedly that the programs themselves are, in general, incident to the course and the importance lies in the development process, the quality of the product and the data collected, and the analysis of data for feedback. They were reminded again and again, that they would not be graded on how good their productivity, defect quality, etc. are, but on how well they execute the development and quality-oriented processes and on how well they record the related data e.g., LOC, number of defects, etc. Thus, they were motivated in a number of different

ways that "*honesty and commitment in doing the tasks and learning*" is a key to their success in the PSP course.

Most of the data collection forms which were used in the course were provided in the textbook [Hum95a]. An overview of these forms is given in section 2.2. In addition to this, we used a locally designed questionnaire and an evaluation form dealing with motivation, which were prepared and validated by the software engineering group at McGill University.

3.2.3 Descriptive Models

This section describes the process model which was used by the subjects in the PSP experiment, and models of quality and productivity which lay the foundation for data analysis in this study.

Process Model

The subjects used a predefined process model, shown in Figure 3.1, to accomplish a project. In this model, project life-cycle activities are divided into chronological phases: planning, design, code, compile, test, and postmortem. Design review and code review are used only from PSP2.

In Figure 3.1, the thin arrows indicate defects found, thus necessitating feedback and rework. The origin of the arrows indicates the phase during which a defect is detected, and arrow head indicates the phase during which a defect is injected. The

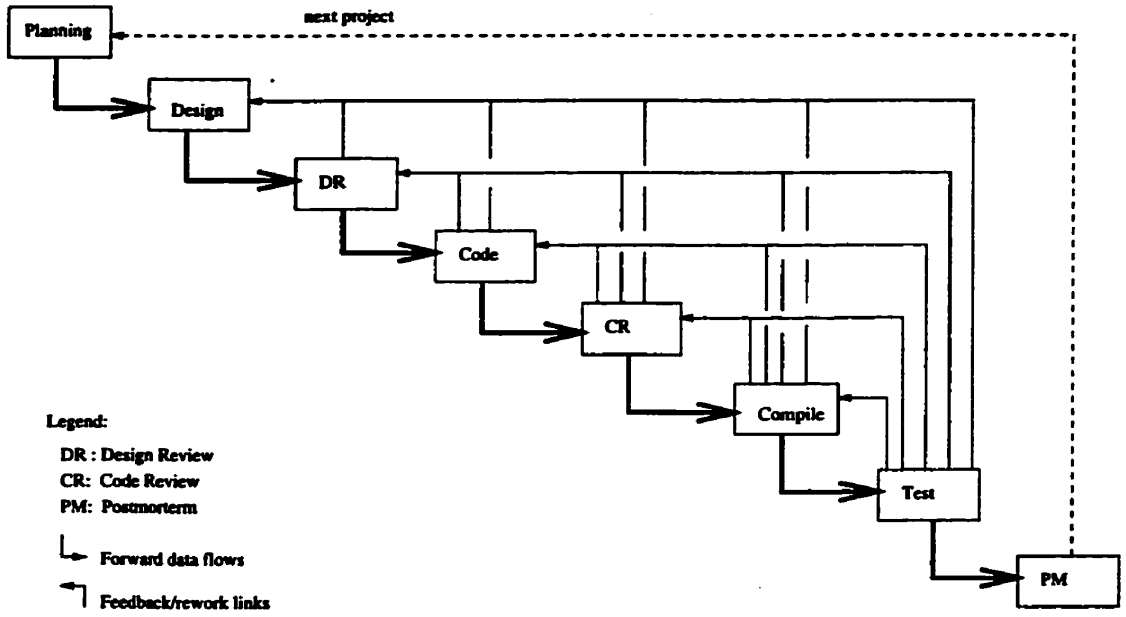


Figure 3.1: PSP Process Model

solid arrows indicate the flow of artifacts.

The predefined process model facilitates categorizing of time-related and defect-related data according to the specific phases and activities of the personal process. Such categorization can then help engineers to analyze their product and process, to understand the conditions they are in, and to improve them.

Defect Model

To manage software quality effectively, an engineer needs to keep track of every defect injected during development on a Defect Recording Log(see Figure 2.4). The defect model is represented as:

$$\mathbf{D_i(Di_i,\ Dr_i,\ Dt_i;\ Cst_i,\ Btk_i)}$$

where D denotes a defect; Di, Dr, Dt, Cst, and Btk are *attributes* of defect D and i is a unique identification number. The attributes are described below.

- **Di: Injection Phase**

This attribute denotes the phase during which the defect is injected. The value of Di is either DS(design), DR(design review), CD(code), CR(code review), CP(compile) or TS(test).

- **Dr: Detection Phase**

This attribute denotes the phase during which the defect is detected and removed. The possible values for Dr are DS(design), DR(design review), CD(code), CR(code review), CP(compile) and TS(test).

- **Dt: Defect Type**

This attribute denotes the type of the defect, as described by defect type standard, e.g., documentation, assignment, data, function, etc. (see Figure 2.3).

- **Cst: Cost of Defect**

This attribute denotes the cost of defect as the direct expense incurred in fixing an injected defect. This includes the following elements:

1. Determining that there is a problem.

2. Isolating the source of the problem.

3. Determining exactly what is wrong with the product.

4. Fixing the design as needed.

5. Fixing the implementation as needed.

6. Inspecting the fix to ensure that it is correct.

7. Testing the fix to ensure that it fixes the identified problem.

8. Testing the fix to ensure that it doesn't cause other problems.

9. Changing the documents to reflect the fix.

When a defect is not corrected properly in the first attempt, subsequent iterations of the correction must be done until the defect is eliminated. The cost of the original defect includes all iterative attempts to fix it.

- **Btk: Backtracking**

This attribute denotes the number of phases we have to backtrack in order to fix a given defect. For example, if a defect is injected in the phase *code* and removed in the phase *code review*, the backtracking for this defect is 1.

$$Btk = Dr - Di$$

The definitions of the defect attributes are summarized in Figure 3.2.

Attribute	Definition
Di	the phase during which the defect is injected
Dr	the phase during which the defect is detected and removed
Dt	the type of the defect, see (Figure 2.3)
Cst	direct expense incurred in fixing the defect
Btk	the number of phases we have to backtrack in order to fix the defect

Figure 3.2: Defect Attributes

Defect-Based Quality Model

The *quality* focus in PSP is centered around defect detection and prevention. Other aspects of software quality, such as reusability, portability, etc. are of secondary importance in PSP. Defects, once detected, are best handled as soon as possible and at the individual level. There are data which suggests the exponential rise in software costs for fixing bugs farther away from the point of origin [Boe81]. Furthermore, a significant portion of the post-delivery system evolution costs are attributed to latent or residual defects in software systems. All these point to the need for strong detection and prevention mechanisms in the personal processes.

Based on the process model and defect model outlined in the previous section, we can build a *model* of process quality:

Q(D1 .. Dn; Dds, Drr, RC/TC, A/FR, Yld, ABtk, DT/TDT,

NTD/ND)

where Q denotes quality of a particular process or product; D1 ... Dn are the defects detected and fixed in the project (defined in the previous section). Dds, Drr, RC/TC, A/FR, Yld, Abtk, Yld, DT/TDT, NTD/ND are quality attributes of software process.

The notation of the quality attributes are shown in Figure 3.3.

Attribute	**Definition**
Dds	Defect Density: number of defects per KLOC
Drr	Defect Removal Rate: number of defects found and removed per hour
RC/TC	reused LOC as a percentage of total LOC
A/FR	Appraisal Time/Failure Time(see Table 2.2)
Yld	Yield: number of defects removed before the first compile as a percentage of number of defects injected before the first compile
ABtk	average backtracking per defect
DT/TDT	design time as a percentage of total time
NTD/ND	number of defects removed in the test phase as a percentage of total defects

Figure 3.3: Quality Attributes

Development-Productivity Model

Another model that is important for this study is the productivity model. Productivity is generally measured as hours required to do a unit of work. It is a simple concept, but not simple to calculate. When we calculate productivity, we must take into account that without an accompanying assessment of product quality, speed of production is meaningless [FP96].

Thus, in our PSP experiment, every project was accompanied by an explicit requirement description in order to support product quality. Any ambiguities were removed before the subjects started a project. The testing criteria were also emphasized in every project. Test cases were carefully designed and test results were checked.

Behind quality, we must also consider the variation in the definition of software size (LOC is used in this study). Among the various projects, as this affects the calculation of "productivity", we used a particular *coding* standard and a *counting* standard in order to minimize the variation in project size.

Another problem usually rests with the variation in expressive power of different programming languages used in different projects. This was not an issue in our PSP experiment because C++ was the only programming language that was used.

In the PSP course, we used LOC/Hour as the measure of development productivity. While LOC/Hour may seem simple, the calculation of LOC and the hours

should be carefully done in order to obtain meaningful measures. There are various combination of the LOC types that can be used to measure development productivity(see Table 2.1). Added LOC plus modified LOC is chosen by PSP for productivity calculation. A fairly straightforward measure of development time (minute) is employed in PSP, as described in section 2.2.1.

The productivity model is specified as:

P(P0, P1, P2)

P0(Size, Effort; LOC/Hour, RC/TC)

P1(Dds, Drr, ABtk, Yld, NTD/ND)

P2(DT/TDT, TT/TDT, A/FR, Spl)

where P is productivity. Size, Effort, LOC/Hour, RC/TC, Dds, Drr, ABtk, Yld, NTD/ND, DT/TDT, TT/TDT, A/FR, and Spl are productivity *attributes.*

The notation of the productivity attributes is shown in Figure 3.4.

Summary

In this section, we have described the descriptive models: process model, defect model, defect-based quality model and productivity model. We also identify various attributes for each model. These descriptive models will be used as the basis of the analysis of data described in section 3.3.

Attribute	Definition
Size	Lines of Code (LOC)
Effort	Minute is the unit of the measure for development effort in PSP.
LOC/Hour	Lines of code added or modified per hour.
TT/TDT	Test Time as a percentage of Total Development Time.
Spl	System Spoilage, e.g., Total Fix Time as a % of Total Development Time
RC/TC Dds,Drr ABtk,Yld NTD/ND A/FR	see Figure 3.3.

Figure 3.4: Productivity Attributes

3.3 Model for Analysis of Data

This section first classifies the attributes we have discussed thus far in this chapter in order to facilitate the description of data analysis. It then describes our Goal/Question/Metric model and statistical analysis model.

3.3.1 Attribute Types

In our study, we focus on the attributes of quality and productivity, such as Defect Density (Dds) and Average Backtracking (ABtk), which we collectively call here as Quality and Productivity Attribute (QPAtrb). All of QPAtrb are identified and organized in the quality model and productivity model (see section 3.2.3, Figure 3.3 and Figure 3.4) based on the data captured during the PSP projects.

Process Improvement Attribute (PIAtrb)

Among the Quality and Productivity Attributes (QPAtrb) described above, there is a sub-set, such as Defect Density (Dds), Defect Removal Rate (Drr) and LOC/Hour, which characterizes directly the performance of an engineer. These attributes are repeatedly used by management to evaluate the improvement of engineers' processes when process improvement activities are being carried out. In this study, we call these attributes Process Improvement Attribute (PIAtrb). In the published research on PSP [She94] [Hum94b] [Hum95b] [Hum95c] [ESM96] [Hum96b] [SM96] [HT97], Defect Density (Dds), Defect Removal Rate (Drr) and LOC/Hour have received particular attention in assessing the effect of PSP on the engineers' performance improvement. Thus, these three attributes were identified as PIAtrb in this study.

Significant Attribute (SAtrb)

Among the Quality and Productivity Attributes (QPAtrb) described above, there is a subset of attributes which most influence, or significantly correlate to, a given Process Improvement Attribute (PIAtrb). We call this subset of attributes Significant Attribute (SAtrb).

Significant Attributes (SAtrb) do not reflect directly any improvement in engineers' processes, and therefore, are often neglected by management and researchers. These attributes, however, characterize many important aspects of software development, for example, software design, code and test.

For example, the average number of phases backtracked to fix a defect (ABtk) is a *possible* SAtrb for PIAtrb LOC/Hour because it is often reported that the farther a defect penetrates into the software life-cycle, the more efforts are needed to fix it [Boe81] [Dun84] [Pre92]. Low ABtk, for example, could imply faster fix-up tries, hence, higher productivity (LOC/Hour). Whether or not this is true in our study, however, can only be determined through data analysis.

Thus, it is essential to study SAtrb in order to improve our understanding of a personal software process. Moreover, techniques such as design review and code review, which are central to the design of PSP, have a direct influence on such attributes as ABtk and Yield.

Then, clearly, studying SAtrb could provide a deeper insight into the way PSP

influences the improvement in software quality and productivity.

Those attributes of QPAtrb that do not qualify as significant attributes are defined as Non-Significant Attribute (NSAtrb) for a given PIAtrb.

3.3.2 Goal/Question/Metric (GQM)

There are a number of frameworks for identifying and utilizing software engineering metrics [FP96] [Bas84] [Jon96] [MB97]. We have used the widely used Goal/Question/Metric (GQM) paradigm. An important aspect of GQM is to define all your goals and identify metrics in advance and then follow them strictly, instead of getting data first and then observing the trends and patterns found in it to identify "interesting" goals. In other word, it is top-down approach.

Goal of Analysis

The research objective was described earlier:

to investigate how the dynamics of personal software process influence software process improvement during the PSP execution.

We formulate the overall goal of analysis:

G: *To investigate how software Quality and Productivity Attributes (QPAtrb) influence Process Improvement Attributes (PIAtrb) during PSP execution.*

From this overall goal (G), we derived three subgoals:

(a) G1: we first identify the set of Significant Attributes (SAtrb), which have significant influence on, or significantly correlate to, PIAtrb, by analyzing the relationships between PIAtrb and other QPAtrb. This analysis helps us focus on the essential aspects of a software process while diminishing the non-essential aspects;

(b) G2: we then explore those relationships that are noticeable between the set of SAtrb identified in G1 and Non-significant Attributes (NSAtrb). This analysis gives us an improved understanding the way SAtrb influence PIAtrb (subgoal G2);

(c) G3: we then determine how SAtrb vary across the PSP projects life-cycle. This analysis helps to assess the effectiveness of PSP and helps to explain the way PSP improve software process.

The goal of analysis is summarized in Figure 3.5.

Questions and metrics of Interest

We devised the questions, which are shown in Figure 3.6, relevant to our analysis goals. Each question has an associate metric. The purpose of specifying these questions and metrics is that it directs data analysis explicitly towards the requirements of the goals.

Overall Goal:
G: To investigate how QPAtrb influence PIAtrb during PSP execution.
Subgoals:
G1: Identify the set of attributes that qualify as Significant Attributes(SAtrb), by analyzing the relationships between PIAtrb and other QPAtrb.
G2: Analyze the relationships between SAtrbs (identified in G1), and NSAtrbs, and identify those relationships that are significant.
G3: For those SAtrbs identified in G1, determine how they vary through the execution of PSP.

Figure 3.5: Goal of Analysis

3.3.3 Dependent Variables and Independent Variables

Dependent variables and corresponding independent variables were identified (Table 3.1) based on the descriptive models (see section 3.2.3).

The dependent variables (Dds, Drr, and LOC/Hour) are the elements of Process Improvement Attributes (PIAtrb). These are the variables that are affected by other contextual variables, which are listed as independent variables. For each dependent variable, there is a specific set of independent variables specified in the quality and productivity models (see Figure 3.3 and Figure 3.4 in section 3.2.3).

Dependent Variables	Independent Variables
Dds	RC/TC, A/FR, Yld, ABtk, DT/TDT, NTD/ND
Drr	Dds, RC/TC, A/FR, Yld, ABtk, DT/TDT, NTD/ND
KLOC/Hour	RC/TC, Dds, Drr, ABtrk, Yld, NTD/ND, DT/TDT, TT/TDT, A/FR, Spl

Table 3.1: Dependent Variables and Independent Variables

3.3.4 Regression Analysis

Regression methods bring out relation between variables, especially between variables whose relation is imperfect in that we do not have one y for each x [MT77]. In software engineering, we can cite the relation between *software size* and *development time*, or *defect density* and *yield* as examples of imperfect relations in that there is no one-to-one relationship. Regression methods have already been used in empirical software engineering studies [GR97] [Hum95a] [She94]. Variable regression models have been employed to estimate the (presumed) relationships between one variable and another by expressing one in terms of a regression function (such as linear function, quadratic function or log-linear function) of the other.

Two methods are often employed to choose a particular regression function. These methods are: (1) an analytical consideration of the phenomenon concerned, and (2) an examination of the *scatter diagrams* plotted from the observed data[Ost63].

In this study, we performed quadratic regression between each dependent variable

and its independent variables because the software development contexts suggest a non-linear relationship. The form of quadratic regression equations is:

$$DEP = a * INDEP^2 + b * INDEP + c$$

Where

DEP = Value of a dependent variable

INDEP = Value of an independent variable

a = Quadratic coefficient

b = Linear coefficient

c = Intercept

Sub-Goals	Question	Metric
G1	Q1.1: What attributes affect Defect Density, and of those, which are SAtrb?	M1.1: Strength of the relationship between Defect Density and other Attributes.
	Q1.2: What attributes affect Defect Removal Rate, and of those, which are SAtrb?	M1.2: Strength of the relationship between Defect Removal Rate and other Attributes.
	Q1.3: What attributes affect LOC/Hour, and of those, which are SAtrb?	M1.3: Strength of the relationship between KLOC/Hour and other Attributes.
G2	Q2: Are there any noticeable relationships between SAtrb and NSAtrb?	M2: Strength of the relationship between SAtrb and NSAtrb.
G3	Q3: What are the trends of the SAtrb across the PSP projects?	M3: Trends of SAtrb across the PSP projects.

Figure 3.6: Goals, Questions and Metrics

Chapter 4

Data and Results

Using the statistical regression analysis model (see section 3.3.4), GQM analytical framework (see section 3.3.2) and the descriptive models (see section 3.2.3), we analyzed the data collected from seven [1] projects in the PSP course and obtained the following results.

4.1 Significant Attribute (SAtrb) Identification

This section deals with the first subgoal (G1). By performing quadratic regression analysis on the data collected, we identified the Significant Attributes (SAtrb) for each Process Improvement Attribute (PIAtrb): Defect Density (Dds), Defect Removal Rate (Drr) and LOC/Hour. Two variables are considered significantly cor-

[1] The data from the first project is not included because the size of the first project could not be recorded before coding standard and counting standard were introduced in the second project.

Independent Variable	Quadratic Equation	R^2
RC/TC	Dds = 0.0038*RC/TC^2+0.2771*RC/TC+46.841	0.0046
A/FR	Dds = -0.0002*A/FR^2+0.0354*A/FR+45.073	0.0151
Yld	Dds = -0.0148*Yld^2+1.229*Yld+31.392	**0.2584**
ABtk	Dds = -4.5312*ABtk^2+21.254*ABtk+23.584	0.045
DT/TDT	Dds = -0.0033*DT/TDT^2-0.4208*DT/TDT+51.052	0.0164
NTD/ND	Dds = -0.0107*NTD/ND^2+0.952*NTD/ND+36.406	0.1829

Table 4.1: Regression Equations for Defect Density (Dds)

related if the *correlation coefficient (r)* is relatively high ($r \geq 0.5$) [Ost63] [PP97]. Since we can not expect a high r value if the dependent variable is a very complex variable associated with more than one independent variables [PP97], in our study, we consider a relationship to be significant if the r^2 is not less than 0.25.

4.1.1 Significant Attributes (SAtrbs) for Defect Density (Dds)

Table 4.1 lists the quadratic relationships between dependent variable Dds and its independent variables.

As can be seen from Table 4.1, Dds has a significant relationship only with Yield(Yld). Figure 4.1 shows the relationship equation graph between these two variables. The maximum Dds value is associated with projects with Yld = -1.229/(2*(-

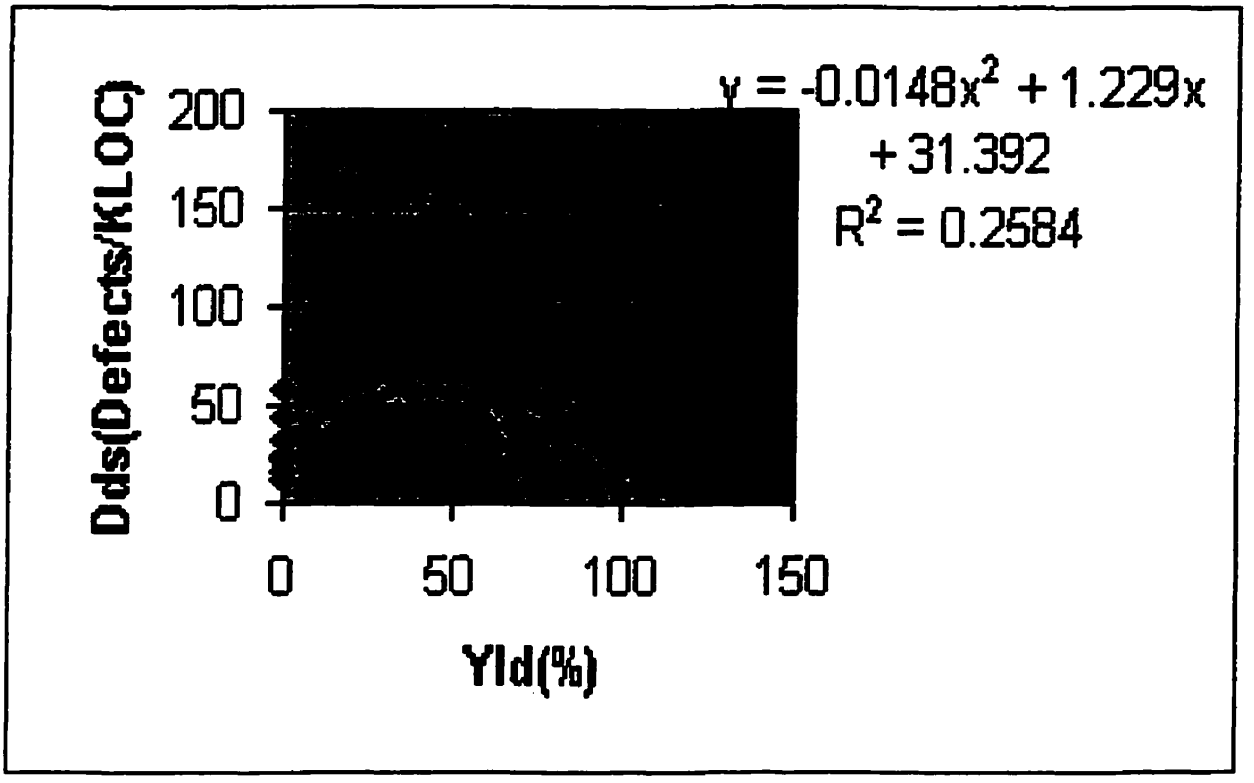


Figure 4.1: Relationship between Dds and Yld

0.01481)) = 42(%). Both low Yld values and high Yld values are related to low Dds. As described in section 3.3.1, Dds has been used as a measure of the benefit of using PSP concepts. Using this measure, it is assumed that if Dds value tends downward, the code quality is improving.

However, El Emam et.al. have argued that this assumption may not be appropriate in the PSP context since we have data only from unit testing [ESM96]. They argued that Low Dds could mean defect detection is poor or less defects have been injected.

In our opinion, low Dds associated with high Yld values would imply high code

quality since a high Yld value indicates strong defect detection. On the other hand, low Dds associated with low Yld values may be a result of poor defect detection and seems to imply poor code quality. From this, we note our first observation:

Observation 1: *When Dds is used to evaluate the quality of software, Yld may be an important attribute that should also be taken into account.*

Independent Variable	**Quadratic Equation**	**R^2**
Dds	Drr = -0.0024*Dds^2-0.3018*Dds+21.081	0.0486
RC/TC	Drr = -0.0029*RC/TC^2+0.2754*RC/TC+12.453	0.0295
A/FR	Drr = -0.0006*A/FR^2+0.228*A/FR+2.7327	**0.294**
Yld	Drr = -0.0033*Yld^2-0.1472*Yld+12.407	**0.2737**
ABtk	Drr = -1.8761*$ABtk^2$-10.521*ABtk+27.543	0.0221
DT/TDT	Drr = -0.0321*DT/TDT^2-1.2469*DT/TDT+5.4882	0.0281
NTD/ND	Drr = -0.0023*NTD/ND^2-0.3008*NTD/ND+20.626	0.0449

Table 4.2: Regression Equations for Defect Removal Rate (Drr)

4.1.2 Significant Attributes (SAtrbs) for Defect Removal Rate (Drr)

Table 4.2 lists the quadratic relationships between dependent variable Drr and its independent variables.

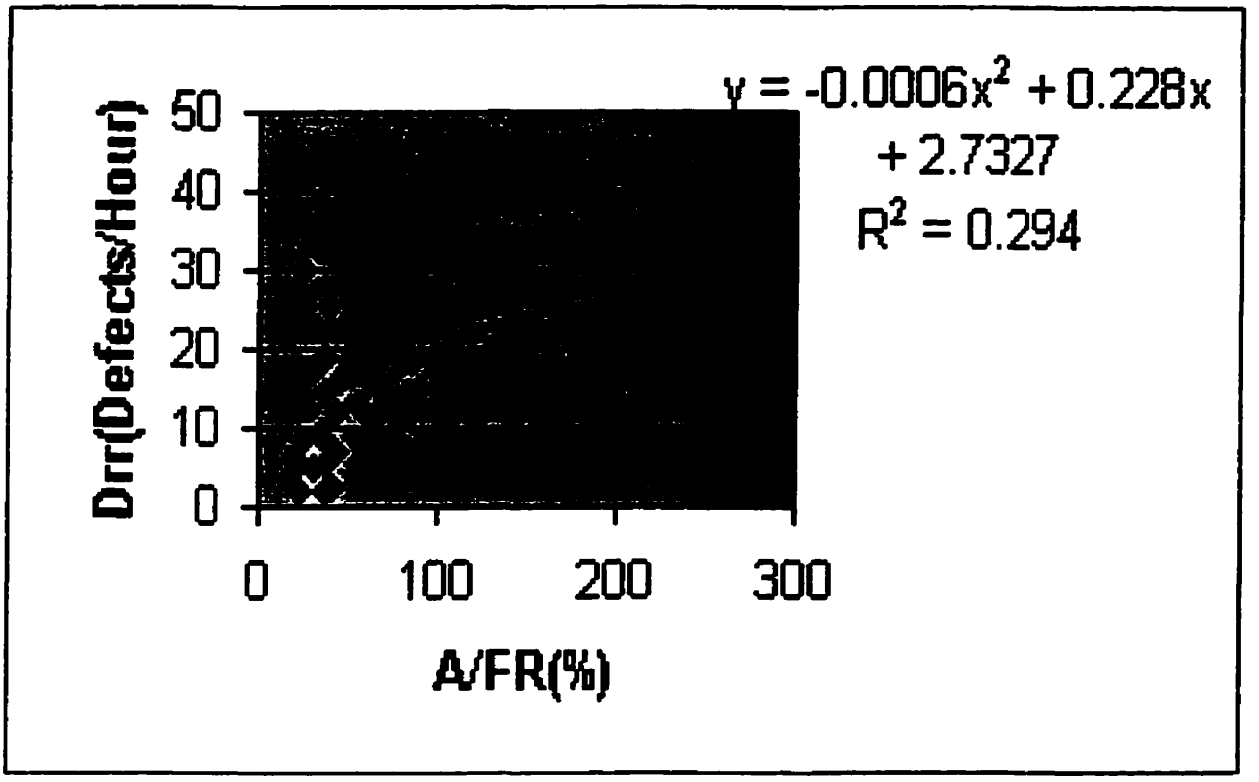


Figure 4.2: Relationship between Drr and A/FR

As can be seen from Table 4.2, Drr has significant relationships with both A/FR and Yld. Figure 4.2 and Figure 4.3 show the relationship equation graph Drr and A/FR and that between Drr and Yld respectively. As described in section 3.3.1, Drr has been used to assess the performance of engineers and the quality of their products. Using this measure, it is assumed that if Drr increases, the defect detection ability of engineers and the quality of their products [2] also increase. As can be seen

[2]High Drr implies that, in general, the defects have been relatively easy to fix which, in turn, implies that the defects are fixed close to their points of origin in the software life-cycle. Therefore, high Drr implies that the product, by the time it is completed, is relatively free of defects (i.e., of

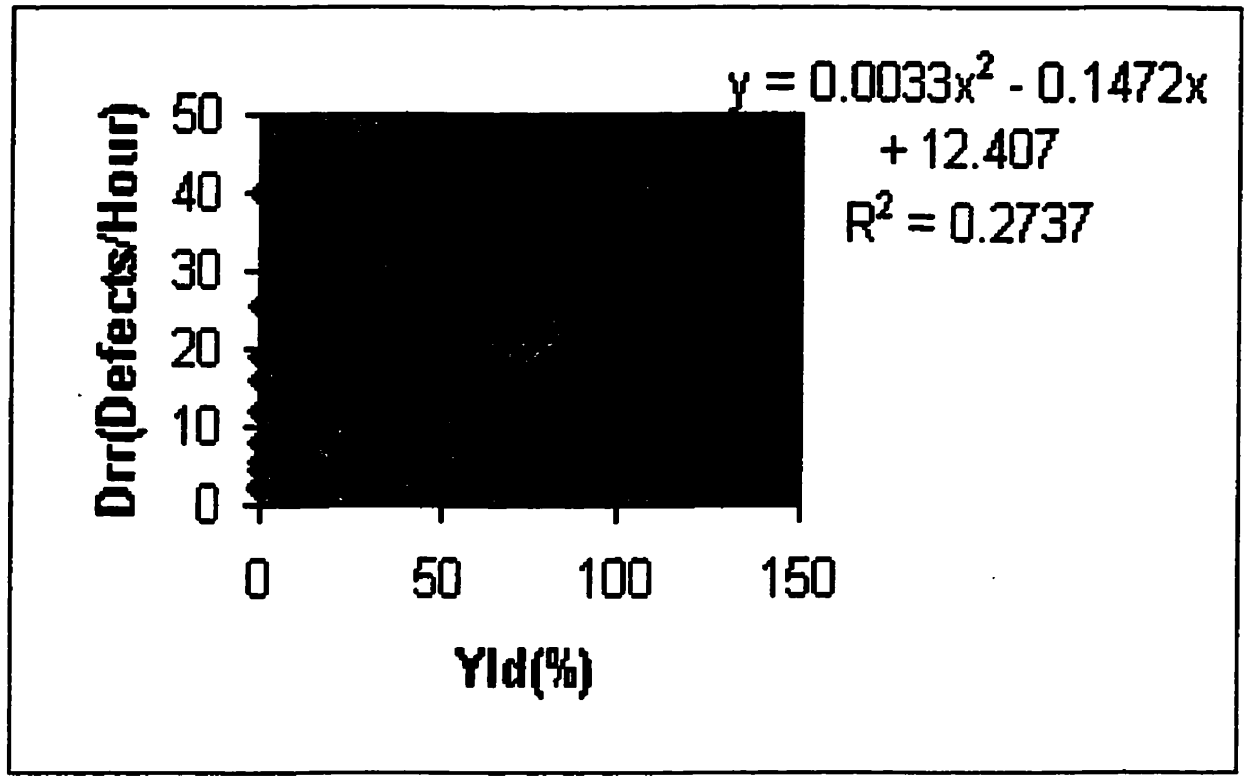


Figure 4.3: Relationship between Drr and Yld

from Figure 4.2 and Figure 4.3, both increasing A/FR value and increasing Yld value have positive effect on Drr.

From the relationship equation between Drr and A/FR, Drr value increases with the increase of A/FR value and an optimal value of Drr is achieved with an A/FR = -0.228/(2*(-0.0006)) = 190(%). We know that A/FR is the ratio of design and code review time to compile and test time. Also, we know that it measures the relative effort spent in early defect removal. Its objective is to detect defects in earlier phases and thus improve the Defect Removal Rate (Drr). However, once the objective is

a high quality).

met, further increase of A/FR will likely decrease Drr. From this, we obtain our second observation.

Observation 2: *The value of A/FR may be a useful guide for software developers in adjusting their review time so as to achieve a high Drr during software development.*

From the relationship between Drr and Yld, Drr value slightly decreases with the increase of the Yld value until Yld < -(-0.1472)/(2*0.0033) = 22.3(%) and then Drr substantially increases with the increase of Yld value (Yld > 22.3%). Form this, we obtain our third observation.

Observation 3: *A Low Yld value may imply that review skills are poor or that the effort spent on review is not adequate. In this case, most of the defects captured by review may be syntax or simple errors. However, compiling is more effective in capturing syntax errors than is a review. This may be interpreted as: Drr decreases when the Yld value is low. When the reviews are just introduced, software developers could be discouraged by the decrease of Drr (perhaps due to the fact that the review skills may as yet be low). They thus need to be encouraged to spend more efforts on reviews, especially at this early stage. By striving to increase their Yld, they will think more positively about their time*

and efforts spent on reviews.

Independent Variable	Quadratic Equation	R^2
Dds	LOC/Hour = 0.0032*Dds^2-0.6589*Dds+53.096	0.1811
Drr	LOC/Hour = -0.0003*Drr^2-0.527*Drr+24.853	0.1437
RC/TC	LOC/Hour = -0.00001*RC/TC^2+0.3364*RC/TC+28.185	0.1094
A/FR	LOC/Hour = -0.0005*A/FR^2+0.2895*A/FR+16.609	**0.244**
Yld	LOC/Hour = 0,0085*Yld^2+0.4715*Yld+30.936	**0.3899**
ABtk	LOC/Hour = 4.048*ABtk^2-23.985*ABtk+63.938	0.0655
DT/TDT	LOC/Hour = -0.0414*DT/TDT^2+1.3816*DT/TDT+23.587	0.0182
NTD/ND	LOC/Hour = 0.0037*NTD/ND^2-0.4322*NTD/ND+40.055	0.043
TT/TDT	LOC/Hour = -0.0054*TT/TDT^2+0.236*TT/TDT+30.34	0.0027
Spl	LOC/Hour = 0.0088*Spl^2-0.3407*Spl+34.04	0.027

Table 4.3: Regression Equations for LOC/Hour

4.1.3 Significant Attributes (SAtrbs) for LOC/Hour

Table 4.3 lists the quadratic relationships between dependent variable LOC/Hour and its independent variables.

As can be seen from Table 4.3, LOC/Hour has significant relationships with both A/FR and Yld. Figure 4.4 and Figure 4.5 show the relationship equation graph between LOC/Hour and A/FR and that between LOC/Hour and Yld, respectively.

As described in section 3.3.1, LOC/Hour is an important criterion for assessing

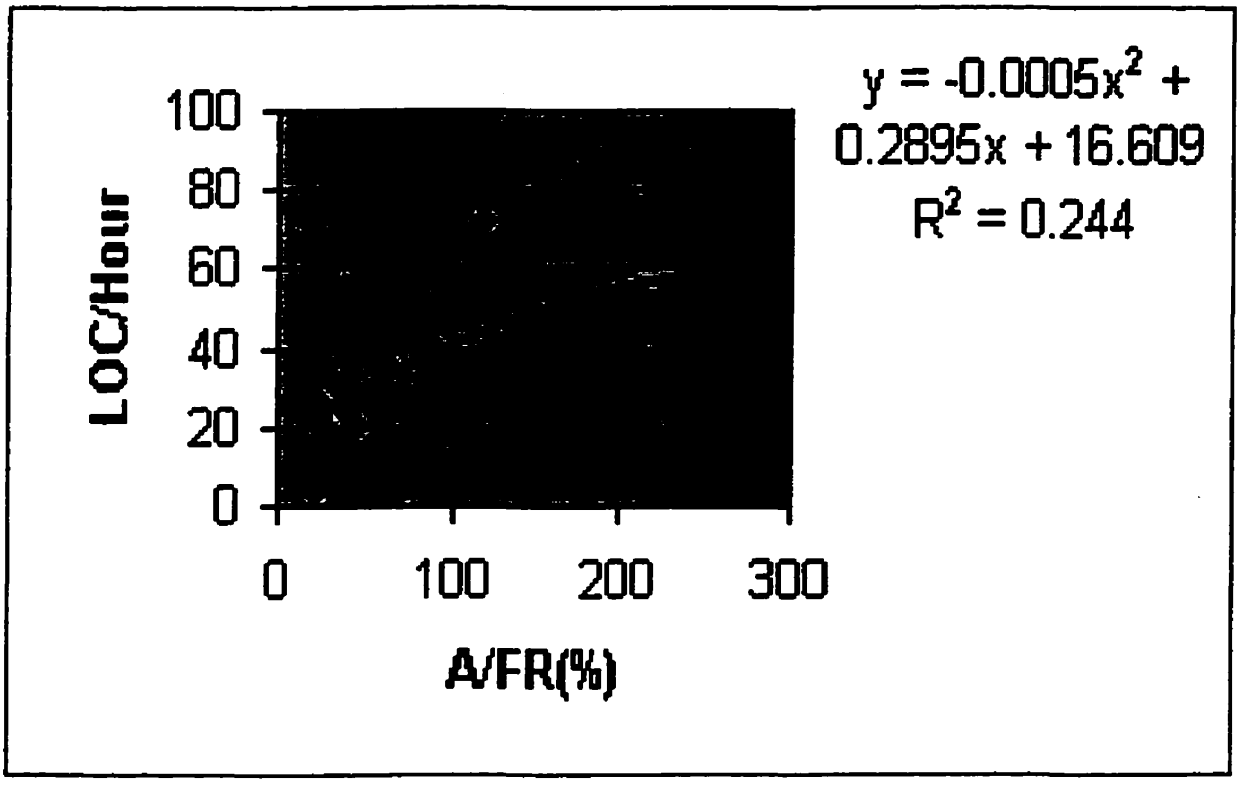


Figure 4.4: Relationship between LOC/Hour and A/FR

the software productivity of engineers. From the relationship between LOC/Hour and A/FR, LOC/Hour increases when A/FR increases (see Figure 4.4). From the relationship between LOC/Hour and Yld (see Figure 4.5), LOC/Hour slightly decreases with the increase of Yld value until Yld < -(-0.4715)/(2*0.0085) = 27.7(%). Then, it greatly increases with the increase of the Yld value (Yld > 27.7(%)). From this, we obtain our fourth observation.

Observation 4: *Reviews have significant positive effect on LOC/Hour since LOC/Hour increases with the increase in A/FR and also with the increase in Yld.*

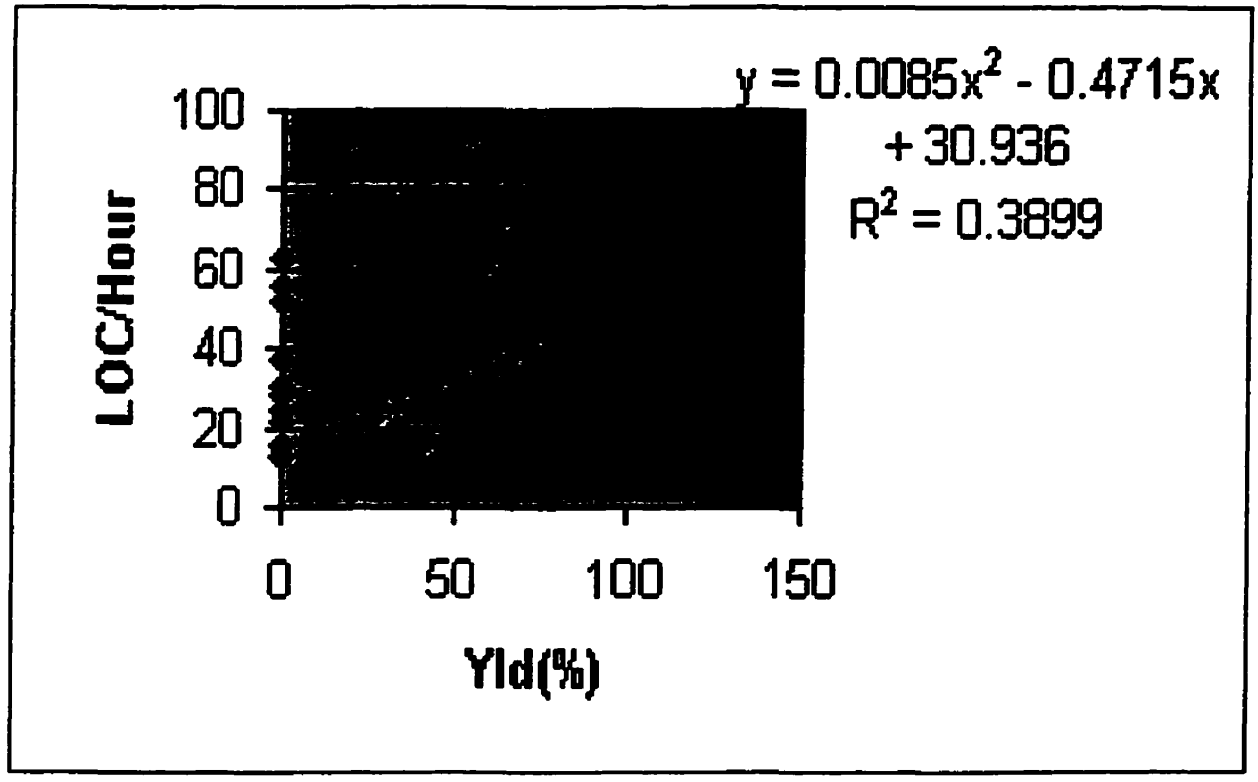


Figure 4.5: Relationship between LOC/Hour and Yld

It is worth noting that PSP has been reported to have little positive effect on LOC/Hour [ESM96] [Hum95c] [Hum96b]. This is perhaps because of the *overhead time* required to do the several tasks featured in PSP. These tasks, in practice, are often considered to include making plans, reviewing programs, and tracking and reporting results. However, our results show that reviews may in fact help increase productivity (LOC/Hour). Thus, the time spent in reviewing programs should not be classified as overhead time. This clarification may help engineers develop a positive attitude towards reviews.

PIAtrbs	SAtrbs
Dds	Yld
Drr	Yld, A/FR
LOC/Hour	Yld, A/FR

Table 4.4: Significant Attributes

4.1.4 Summary on Significant Attributes (SAtrbs)

From the above analysis, we have identified the SAtrbs for each Process Improvement Attributes, as summarized in Table 4.4: both Yld and A/FR are Significant Attributes (SAtrb) for LOC/Hour and Drr, and Yld is the only Significant Attribute for Dds.

In particular, Defect Density (Dds) may imply high quality of software product when it is associated with high Yld value and may imply low quality of software product when it is associated with low Yld value (see Observation 1). Thus, when Dds is used to evaluate the quality of software, the evaluation results may be more accurate if it's significant attribute, Yld, is also taken into account.

Yield (Yld) has no significant positive effect on Drr unless it has achieved a high level (see Observation 3). This result highlights one of the obstacles in introducing review skills and emphasizes the importance of developing strategies to encourage software developers to think positively about the review time in the early stages when review skills are implemented.

Quadratic Equation	R^2
ABtk = -0.00005*A/FR^2-0.0208*A/FR+3.3126	0.4316
NTD/ND = 0.0023*A/FR^2-0.8302*A/FR+79.566	0.3781
TT/TDT = 0.001*A/FR^2-0.3241*A/FR+31.71	0.3529
Spl = -0.0006*A/FR^2-0.1805*A/FR+19.73	0.1073
ABtk = -0.00003*Yld^2-0.0092*Yld+2.6081	0.2196
NTD/ND = 0.0026*Yld^2-0.1647*Yld+49.59	0.175
TT/TDT = 0.0018*Yld^2-0.0865*Yld+17.802	0.0776
Spl = -0.0031*Yld^2+0.2343*Yld+10.625	0.1362

Table 4.5: Regression Equations between SAtrbs and NSAtrbs

Defect Removal Rate (Drr) increases with the increase of A/FR when A/FR < 190% and decreases with the increase of A/FR when A/FR > 190%. This A/FR value may be a useful guide for software developers in adjusting their review time for achieving a high Drr during software development (see Observation 2).

LOC/Hour increases with the increase of A/FR and Yield (Yld). The analysis of the relationships between LOC/Hour and A/FR and that between LOC/Hour and Yld clarifies the often misunderstood concept that review time is an overhead of PSP (see Observation 4). This may help further refinement of PSP, for example, by developing strategies to encourage engineers to pay more attention to reviews.

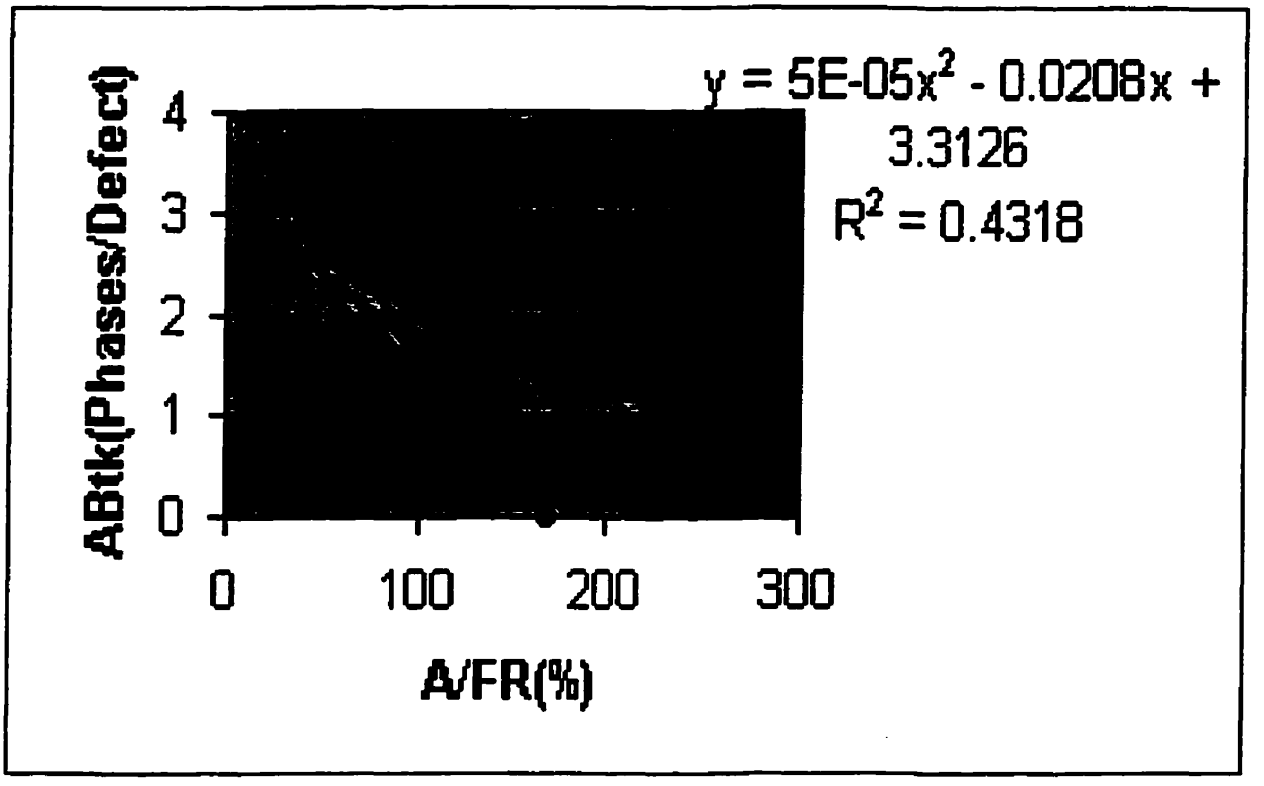


Figure 4.6: Relationship between ABtk and A/FR

4.2 Relationships Between Significant Attributes (SAtrbs) and Some Non-Significant Attributes (NSAtrbs)

Now, we know from observations 1, 2, 3, and 4 that the Significant Attributes (SAtrb) have a noticeable impact on the target Process Improvement Attributes (PIAtrb): Defect Density (Dds), Defect Removal Rate(Drr) and LOC/Hour. We

can theorize, and perhaps even back up with experience, *why*[3] there is such an impact. However, such arguments would be more credible if our current data-set can help explain our intuition.

Thus, there is a need for the subgoal (G2), whereby we analyze the relationships between SAtrb and Non-significant Attributes (NSAtrb). If we can determine that there are some significant relationships between SAtrb and NSAtrb, then these particular relationships might help explain our intuition about the impact of SAtrb on Dds, Drr and LOC/Hour. The argument is that, unlike SAtrb, the attributes NSAtrb, by themselves, clearly do not have significant impact on the Process Improvement Attributes (Dds, Drr and LOC/Hour). However, a significant relationship between a particular NSAtrb and a particular SAtrb might give some more insight into *why* that particular SAtrb is significant.

Table 4.5 lists the quadratic relationships between SAtrbs and some NSAtrbs. First three equations (ABtk, NTD/ND and TT/TDT) are considered significant.

4.2.1 Noticeable Relationships

Figure 4.6 relates A/FR to ABtk (the average number of phases backtracked in order to fix a defect). As can be seen from this figure, ABtk drops with the increase in A/FR. Considering Figure 4.2 and Figure 4.4 together with Figure 4.6, we can infer that the decrease in ABtk contributes to the increase in Defect Removal Rate

[3]This rationale is not self-evident from the findings.

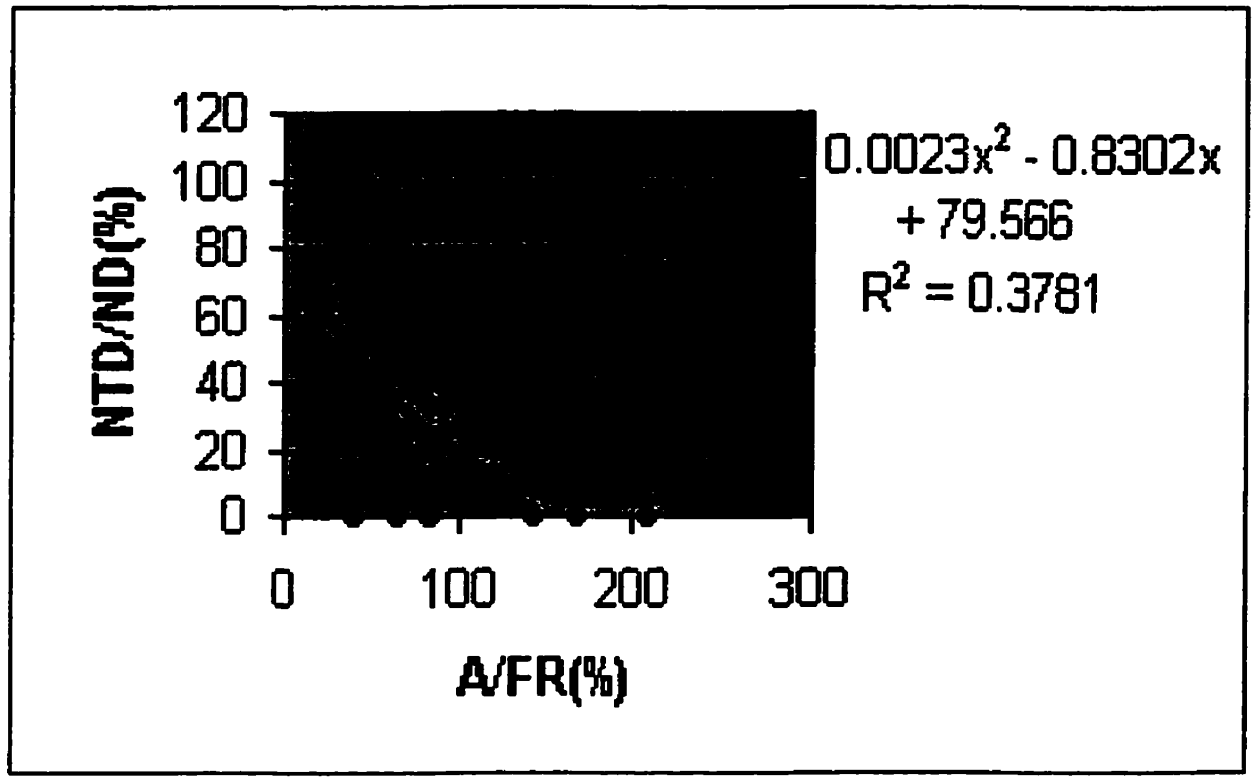


Figure 4.7: Relationship between NTD/ND and A/FR

(Drr) and LOC/Hour.

We hold this inference because low ABtk (from Figure 4.6) value can imply that software defects pertain to the current development phase or previous development phases not too far away from the current phase. Clearly, in these cases, the defects are relatively easily removed [4] *(Drr value is high in Figure 4.2) and software project has a tendency to move forward at an increased speed, giving rise to higher productivity (LOC/Hour) (see Figure 4.4) and lower cost.*

[4]This argument is also supported in the literature [Boe81].

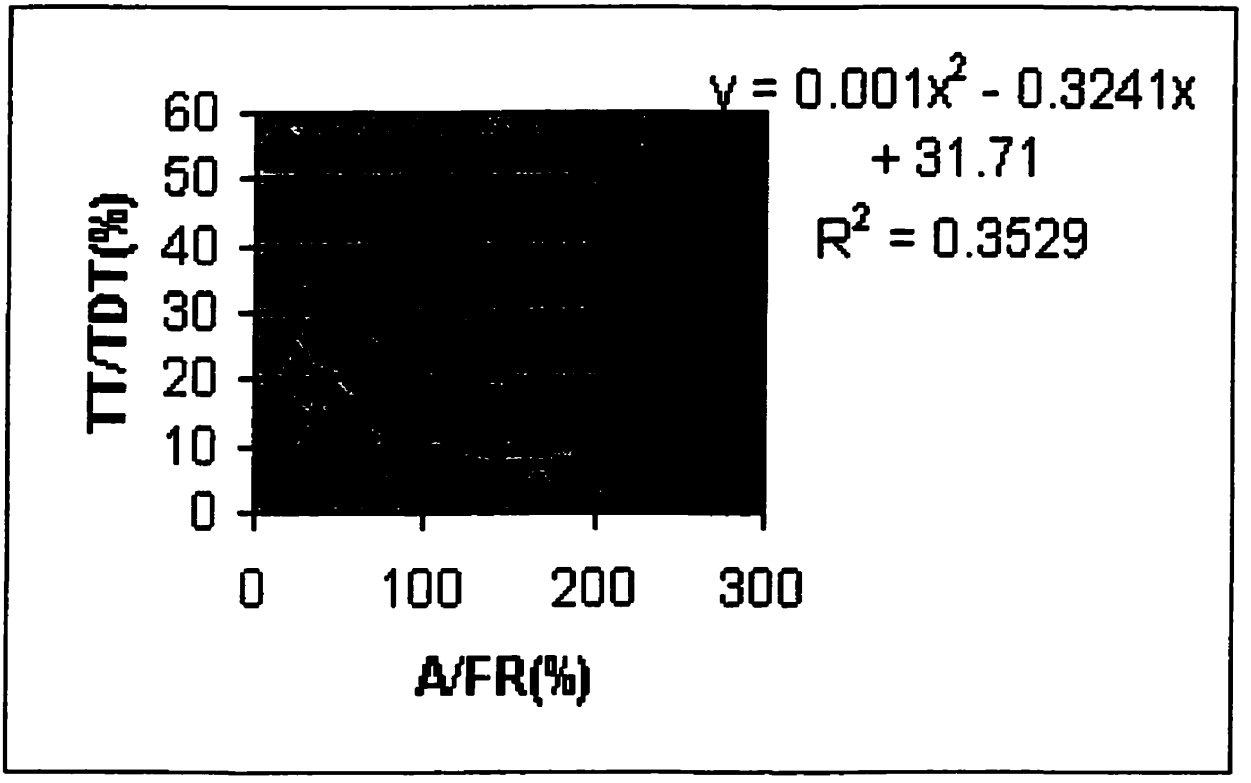


Figure 4.8: Relationship between TT/TDT and A/FR

Figure 4.7 shows the regression equation graph between A/FR and NTD/ND (the number of test defects as a percentage of total defects). Here, assuming a relatively constant number of total defects, A/FR values above 100% are associated with relatively low number of test defects; whereas, A/FR values below 100% are associated with relatively high number of test defects. *This means that, in general, higher effort in software reviews leads to fewer defects in the test phase.*

Reducing test defects is one of the important objectives in software development because: (i) a high number of test defects implies poor software quality, and (ii) test defects are relatively expensive to fix. Since it is generally difficult for engineers to

determine product quality during development, the A/FR measure is a useful guide to personal practice. While our finding hovers around 190% in general, how high the A/FR ratio should be needs further empirical studies. But it is clear that the cost of appraisal and the cost of fixing defects during the test phase needs to be considered.

Figure 4.8 presents the relationship of A/FR and TT/TDT (the test time as a percentage of total time). *This may imply that reviews reduce the time spent in the test phase because of fewer defects creeping into in the test phase.*

4.2.2 Summary of PIAtrbs, SAtrbs and NSAtrbs

From the above analysis, Yield, which is an element of Significant Attribute (SAtrb), has significant relationships with all the three Process Improvement Attributes (PI-Atrbs): LOC/Hour, Defect Density (Dds) and Defect Removal Rate (Drr). No significant relationships between Yield and Non-significant Attributes (NSAtrbs) were found.

A/FR has also significant relationships with: PIAtrbs (LOC/Hour and Drr), and three NStrbs (ABtk, TT/TDT and NTD/ND). In particular, A/FR, relative effort spent in early defect removal, has direct effects on several aspects of personal software process, e.g., average backtracking, the number of test defects and time spent on test phase. Through these aspects, A/FR thus influences software developers' performance in terms of Defect Density (Dds), Defect Removal Rate (Drr)

and productivity (LOC/Hour).

4.3 Trends of SAtrbs in the PSP Projects

We know from the analysis in the previous sections that attributes, Yld and A/FR, are significant in that they influence several aspects of software development. What we have not described, as yet, is how these two variables vary, over time, as PSP execution evolves from PSP0 to PSP3. This is the subject of sub-goal G3. It is important to know how Yld and A/FR vary over time because this knowledge can be *feed back* (see Figure 1.1) into the improvement (or re-design) of PSP itself.

Figure 4.9 shows A/FR and Yld trends over the seven PSP projects. Here, the sharp jump in A/FR and Yld with project 7 results from the introduction of design and code reviews at this point.

From the analysis of section 4.1 and section 4.2, an increase in A/FR contributes to the decrease in average backtracking (ABtk), the number of test defects and the time spent on test, and thus it improves Defect Removal Rate (Drr) and productivity (LOC/Hour). From Figure 4.9, the A/FR increases from 0 in assignment 6 to 72% in assignment 7 and 68% in assignment 8, which are significant increases.

Our analysis also indicates that the software developers' Defect Removal Rate (Drr) and productivity (LOC/Hour) improve with the increase in the value of Yld. From Figure 4.9, the Yld value increases to 32% in assignment 7 and 36% in assign-

ment 8 from less than 10% in the first 6 assignments. This helps to show that PSP actually improves the software developers' performance by improving the Significant Attributes (SAtrb) that influence the personal software processes the most.

4.4 Implications of the Findings for PSP

Despite this evidence and benefit of PSP, it begs the questions as to how the findings of this thesis can be used to improve PSP.

Firstly, in the PSP course, students are usually required to complete 10 programming projects [5]. However, design and code reviews, which directly influence the Significant Attributes (A/FR and Yld), are not introduced until assignment 7 (see Figure 4.9)when more than half of the projects have been completed. We have noted that developers have often expressed concern about significant improvement in product quality during the early stages of PSP when they themselves are not able to see the fruits of their efforts. Thus, it seems as though reviews can be introduced slightly earlier in the course, so that students can see rapid improvement in their performance and, hopefully, maintain their level of motivation in PSP.

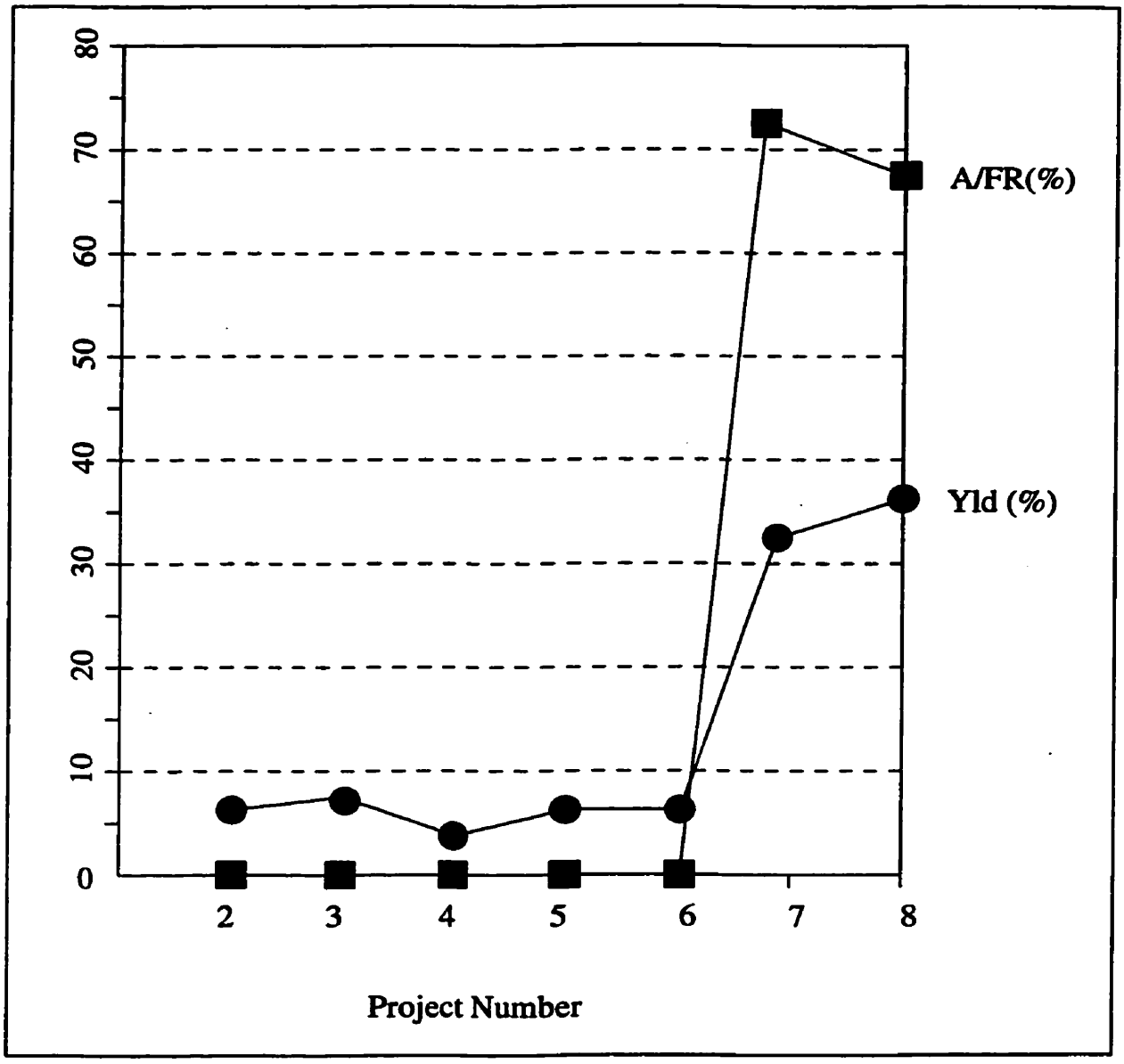
Secondly, our study shows that Yld have no significant positive effect on Defect Removal Rate (Drr) unless it has achieved a high level (see Observation 3 and Figure 4.3). In the projects where reviews have just been introduced, engineers can

[5]In our study, only 8 projects were required because significant time was devoted to three additional data-analysis projects, which gave students concrete feedback on the value of PSP.

be easily discouraged by the decrease because they have not put adequate efforts on reviews or because they have not mastered the skills as yet. Therefore, what they need, especially at this stage, is specific guideline for the *baseline* of the value of Yld that should be achieved for a positive impact on Drr. Currently, there is no such guideline in PSP. While our study suggests an optimal Yld value (not less than 50%)(see Figure 4.3), more studies are clearly needed to validate this proposal. The review procedures should then emphasize such a Yld value for maximum benefits.

Finally, Our study shows that the A/FR ratio could be a useful guide for the engineers in adjusting the review time so as to achieve a high Drr during software development. For example, the optimal value of A/FR in our study is 190% (see Figure 4.2 and Observation 2). Again, while more such studies are needed, such a value could become an integrated part of PSP training.

In summary then, our study highlights some new findings from the white-box study of PSP. In addition, we have also suggested above how such findings could help improve PSP, although, in general, we support the idea of more empirical studies prior to changing PSP.



The sharp jump in A/FR and Yld with Project 7 results the introduction of design review and code review at this point.

Figure 4.9: Trends of A/FR and Yld Over PSP Projects

Chapter 5

Conclusion and Future Work

Following the development of Personal Software Process (PSP) by Humphrey, a number of efforts have been made to study the impact of PSP on software process improvement. However, much of such research has focused on the *result* of the execution of PSP – the improvement of software quality and productivity. Little attention has been paid to the underlying factors that influence the output of the execution of PSP. By investigating how the underlying factors influence the output, it is argued that we would have an improved understanding of PSP and, in turn, this could lead to the improvement of PSP and eventually the processes and products of PSP.

In this study, we built an analytical framework consisting of descriptive software quality and productivity models, Goal/Question/Metric (GQM) paradigm, and quadratic regression analyses. We applied this framework in a PSP experiment

to investigate how the dynamics of software process influence the improvement of software quality and productivity during the execution of PSP. The key findings were:

- Yield and A/FR are the underlying factors that have significant effect on the output of PSP, evaluated in terms of the improvement in Defect Density (Dds), Defect Removal Rate (Drr) and LOC/Hour.
- Yield, combined with Defect Density, is an important software quality measurement.
- The A/FR ratio can be used to guide software developers to achieve high quality and productivity during software development.
- A/FR influences the improvement of software quality and productivity by influencing various underlying process factors, for example, average number of phases backtracked to fix a defect.

From these findings, it is clear that our hypothesis: *that not all factors underlying a personal software process have the same impact on quality and productivity*, is true. The factors uncovered by this study are Yield and A/FR, which are seldom emphasized in the software engineering literature. The findings, together with the contributing factors, are a contribution to software engineering knowledge.

Our study complements previous work on PSP by providing a "white-box" view of PSP. This helps to improve our collective understanding of personal software

process and it could lead to the improvement of PSP itself. For example, reviews could be introduced earlier in the PSP topics.

Because this study was conducted in a university environment, the results should not be generalized to PSP projects in industry. Rather, they provide hope and a basis for stronger hypothesis. Thus, we encourage that the results of this thesis could be considered in the design of empirical studies in an industrial setting, for example, to determine the optimal A/FR ratio.

Appendix A

PSP Evolution

This appendix contains a briefly discussion of the PSP evolutionary path.

A.1 Baseline Process(PSP0)

PSP0 is the initial step and establishes a baseline that includes measurements and a reporting format. This provides a consistent basis for measuring progress and a defined foundation on which to improve. PSP0 is essentially the current process the engineers use to write software, enhanced to provide measurements.

Following the first programming exercises, PSP0 is enhanced to PSP0.1 by adding a coding standard, size measurement, and the process improvement proposal(PIP). The PIP provides a structured way to record process problems, experiences, and improvement suggestions. PSP0.1 also enhances program size measure-

ment to separately count methods and procedures.

A.2 Personal Planning Process(PSP1)

PSP1 improves upon PSP0 by focusing on planning elements. Size measurement and estimation, resource projection, schedule planning and status tracking are introduced at this stage. The PSP0 forms and templates are expanded to include a size estimating template; in addition, the plan and summary report now includes data on program size, as well as reuse data.

While the importance of these techniques for large projects is well understood, few engineers apply them to their personal work. The PSP demonstrates the value of these methods at the personal level.

A.3 Personal Quality Management(PSP2)

PSP2 adds personal design and code reviews to PSP1. These reviews help the engineers to find defects earlier in their processes and to appreciate the benefits of doing so. They analyze the defects they find in their early programs and use these data to tailor review check lists to their personal defect propensities. "Review Yield", that is the percent of the defects in the program found during review, is introduced as a useful measure of review process effectiveness.

The design process is addressed in PSP2.1. Its intent is not to tell engineers how

to do design but to address the criteria for design completion. In PSP2.1 design completeness criteria are established and various design verification techniques are illustrated. While the design phase is used as an example of completeness criteria, the same approach can be used with such other process phases as requirement specification, documentation development, and test development. Phase entry and exit criteria are needed to provide review entry criteria, to define process measures, and to track development status.

Up to this point, the PSP stages focus on small, stand-alone programs developed by an individual. A principle role of PSP, however, is its use as a foundation for large-scale software development. Therefore, PSP must be able to address growing product complexity and to relate individuals to their teams. In addition, as teams form into project, the PSP principles should be scalable to address this broader need. The first step toward addressing scalability is the introduction of PSP3, a Cyclic Personal Process.

A.4 Cyclic Personal Process (PSP3)

PSP3 presumes incremental development of a large-scale software system. By utilizing abstraction principles, PSP3 guides individuals through the development cycles of complex software by subdividing the complex system into pieces each applicable to PSP2. PSP3 requires planning and specification of development cycles: design

and design review, test development and review. Then, code and code review, compile, and test phases are applied to each cycle. At the end of each cycle, recorded data is used to assess the current status against the base plan for adjustment or modification.

Bibliography

[Bas84] V.R. Basili. "A Methodology for Collecting Valid Software Engineering Data ". *IEEE Transaction on Software Engineering*, se-10(6):728–738, November 1984.

[Bas92] V.R. Basili. The Experimental Paradigm in Software Engineering. In *Proc. Int. Workshop on Experimental Software Engineering Issues*, pages 3–12, (Held at Schloß Dagstuhl, Wadern, Germany, September 14-18, 1992), 1992. Springer Verlag, Berlin, LNCS 706.

[Boe81] B.W. Boehm. *Software Engineering Economics.* NJ: Prentice-Hall, Englewood Cliffs, 1981.

[Che69] A. Cherns. "Social Research and its Diffusion". *Human Relations*, 22(3):209–218, 1969.

[Cla72] P. Clark. *Action Research and Organizational Change.* Harper and Row, 1972.

[Dio93] R. Dion. "Process Improvement and the Corporate Balance Sheet ". *IEEE Software*, pages 28–35, July 1993.

[Dun84] R.H. Dunn. *Software Defect Removal.* McGraw-Hill, Englewood Cliffs, 1984.

[ESM96] K. El Emam, B. Shostak, and N.H. Madhavji. "Implementing Concepts from the Personal Software Process in an Industrial Setting". In *4th International Conference on Software Process*, 1996.

[FHK+97] P. Ferguson, W.S. Humphrey, S. Khajenoori, S. Macke, and A. Matvya. "Results of Applying the Personal Software Process ". *Computer-IEEE Computer Magazine*, 30(5):24–31, 1997.

[FP96] N.E. Fenton and S.L. Pfleeger. *Software Metrics: a Rigorous and Practical Approach Second Edition.* International Thomson Computer Press, London, UK, 1996.

[Gar93] P. Garavaglia. "How to Ensure Transfer of Training". *Training and Development*, pages 63–68, October 1993.

[GR97] N. Gorla and R. Ramakrishnan. "Effect of Software Structure Attributes on Software Development Productivity". *Journal of Systems Software*, 36:191–199, 1997.

[HO97] W. Hayes and J. W. Over. The personal software process (psp): An empirical study of the impact of psp on individual engineers. Technical Report CMU/SEI-97-TR-001, Software Engineerin Institute, Pittsburgh, 1997.

[HSW91] W.S. Humphrey, T.R. Snyder, and R.R. Wills. "Software Process Improvement at Hughes Aircraft". *IEEE Software*, pages 11–15, July 1991.

[HT97] T.B. Hilburn and M. Towhidnejad. "Doing Quality Work: The Role of Software Process Definition in the Computer Science Curriculum". *SIGCSE Bulletin - Computer Science Education*, 29(1):277–281, 1997.

[HT98] L. Hou and J. Tomayko. "Applying The Personal Software Process in CS1: An Experiment ". *SIGCSE Bulletin - Computer Science Education*, 30(1):322–325, 1998.

[Hum87] W.S. Humphrey. Characterizing the software process: A maturity framework. Technical Report CMU/SEI-93-TR-024, Software Engineerin Institute, Pittsburgh, 1987.

[Hum89] W. S. Humphrey. *Managing the Software Process.* Addison-Wesley, Reading, Mass., 1989.

[Hum93] W.S. Humphrey. "The Personal Software Process, Rationale and Status". In *The 8th International Software Process Workshop*, 1993.

[Hum94a] W.S. Humphrey. "Process Feedback and Learning". In *the 9th International Software Process Workshop*, 1994.

[Hum94b] W.S. Humphrey. "The Personal Process in Software Engineering". In *Proceedings of the 3rd International Conference on the Software Process*, pages 69–77, 1994.

[Hum94c] W.S. Humphrey. "The Personal Software Process ". *Sotware Process Newsletter, IEEE TCSE*, (1):1–3, September 1994.

[Hum95a] W. S. Humphrey. *A Discipline For Software Engineering.* Addison-Wesley, Reading, Mass., 1995.

[Hum95b] W.S. Humphrey. "Introducing the Personal Software Process ". *Annals of Software Engineering*, 1:311–325, 1995.

[Hum95c] W.S. Humphrey. "The Power of Personal Data ". *Software Process Improvement and Practice*, 1:69–81, 1995.

[Hum96a] W.S. Humphrey. "The Personal Software Process and Personal Project Estimating ". *American Programmer*, 9(6):2–15, June 1996.

[Hum96b] W.S. Humphrey. "Using a Defined and Measured Personal Software Proces". *IEEE Software*, 13(3):77–89, 1996.

[Jon96] C. Jones. *Applied Software Measurement.* McGraw-Hill, 1996.

[Kha95] S. Khajenoori. "Personal Software Process: An Experiential Report". In *8th SEI CSEE Conference*, New Orleans, LA, USA, March29-April 1 1995.

[MB97] Y. Mashiko and V.R. Basili. "Using the GQM Paradigm to Investigate Influential factors for Software Process Improvement". *Journal of Systems and Software*, 36(1):17–32, 1997.

[MKN+96] S. Macke, S. Khajenoori, J. New, I. Hirmanpour, J. Coxon, A. Ceberio, and B. Manente. "An Industry/Academic Partnership that Worked: An In Progress Report". In *Proceedings of the 9th Conference on Software Engineering Education*, April 1996.

[MT77] F. Mosteller and J.W. Tukey. *Data Analysis And Regression.* Addison-Wesley, 1977.

[Neu93] P. Neumann. "System Development Woes". *Communications of the ACM*, page 146, 1993.

[Ost63] Benard Ostle. *Statistics in Research.* The Iowa State University Press, Ames, Iowa, U.S.A., 1963.

[PCCW93] M. Paulk, B. Curtis, M. Chrissis, and C. Weber. Capability maturity model for software (version 1.1). Technical Report CMU/SEI-93-TR-024, Software Engineerin Institute, Pittsburgh, 1993.

[PP97] R.C. Pfaffenberger and J.H. Patterson. *Statistical Methods for Business and Economics.* Richard D. Irwin, Inc., Homewood, Illinois, 1997.

[Pre92] R.S. Pressman. *Software Engineering, A Practioner's Approach.* McGraw-Hill, Inc., 1992.

[Roy96] D. Roy. "The Personal Software Process: An 'Ego-Centered' Improvement Paradigm". In *Proceedings of the Software Engineering Process Group Conference*, 1996.

[She94] K. Sherdil. "Personal 'Progress Functions' in the Software Process". Master's thesis, School of Computer Science, McGill University, 1994.

[Sho96] B. Shostack. "Adapting the Personal Software Process to Industry". *Software Process Newsletter*, (5), Winter 1996.

[SM96] K. Sherdil and N.H. Madhavji. "Human-Oriented Improvement in Software Process". In *Proceedings of the 5th European Workshop on Software Process Technology*, Springer Verlag, 1996.

[Woh93] H. Wohlwend. "Software Improvements in an International Company". In *15th International Conference on Software Engineering*, Baltimore, Maryland, May 1993.

[Zul93] R.E. Zultner. "TQM for Technical Teams". *CACM*, 36(10):79–91, Oct 1993.