Applying the Personal Software Process (PSP)^{sm} with Ada

Mr. David Silberberg
U. S. Department of Defense/Q574
9800 Savage Road Suite 6256
Fort Meade, MD 20755-6000
301-688-5931
dsilber@romulus.ncsc.mil

1. ABSTRACT

This report documents my successful experience applying the Personal Software Process to Ada software development. Using the PSP data generated during the completion of my PSP Instructor training at the Software Engineering Institute (SEI), this report shows examples of these improvements (e.g., improved size and time estimating accuracy, reduction of the amount of total project time spent in the compile and test phases, removal of design defects earlier, improved defect removal yield, etc.). I completed this training with Ada95 using the GNAT Ada95 compiler (version 3.04) on a Unix system.
This experience report also provides a brief introduction to the PSP and compares the PSP with the Capability Maturity Model for Software (CMM). It analyzes the process and product data collected during my PSP Instructor training to highlight how Ada in conjunction with the PSP helped me to improve the quality of my software products.

1.1 Keywords

Ada, Personal Software Process^{sm}, PSP^{sm}, Capability Maturity Model^{sm}, CMM^{sm}

2. INTRODUCTION TO THE PSP

The current practice of software engineering is more of a craft than a structured engineering discipline. There are known and effective quality management principles that can be applied to software. In addition, there are known and effective quality methods that can be used by software practitioners. The PSP introduces these methods and motivates the software practitioners to use them [1].

The PSP is described in [1]. The goal of the PSP is to make software practitioners aware of the processes they use to do their work, and the performance of those processes. Software practitioners set personal goals, define the methods to be used, measure their work, analyze the results, and adjust their methods to meet their goals. It is a strategy for professional self-development and enhanced effectiveness. Early data suggest improvement factors of two to three times are possible with the PSP approach [2]. Data collected on over 200 students and software practitioners that have completed PSP training show substantial improvement in the quality of the programs they produced (e.g., average test defects injected per KLOC dropped from 80 to 10). These and other improvements were observed for all levels of experience (from a few years to over twenty years) [5].

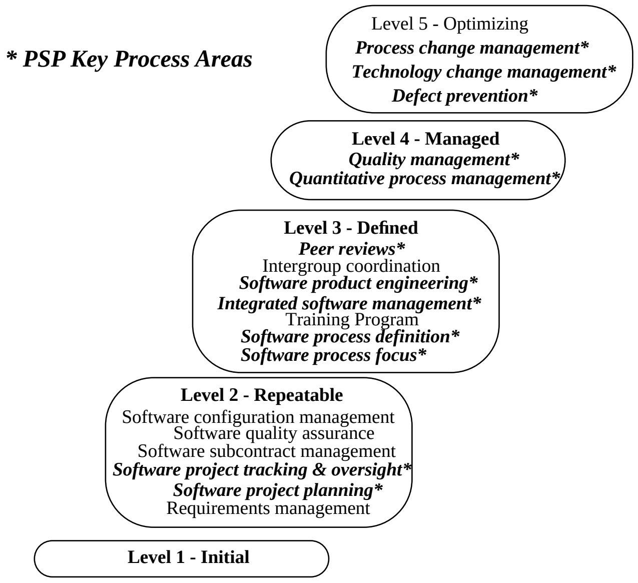
The PSP is based on the following principles: 1) the quality of the software system is governed by the quality of its worst components, 2) the quality of a software component is governed by the individual who developed it and their knowledge, discipline, and commitment, and 3) software practitioners should know their own performance, measure, track, and analyze their work, and learn from their performance variations and incorporate these lessons into their personal practices.

In addition, a stable PSP allows a software practitioner to estimate and plan his work, meet his commitments and resist unreasonable commitment pressures, understand his ability, and identify areas in need of improvement. A stable PSP also provides a software practitioner with a proven basis for developing and practicing industrial-strength personal disciplines, a discipline that shows him how to improve his personal performance, and the data to continually improve the productivity, quality, and predictability of his work.

The objectives of the PSP course are 1) introduce software practitioners to a process-based approach to software development, and 2) show software practitioners how to measure and analyze their personal software process, use process data to improve their personal performance, and apply these methods to their other tasks. The course strategy is to provide the opportunity for practitioners to learn, to use, and to see the benefits of working in a disciplined, process-driven environment. The belief is that in order for practitioners to accept and to use these methods and techniques, they must first believe that these methods and techniques are effective. To believe that a technique or method is effective, a practitioner must use them. The PSP course provides the practitioner with that opportunity [1].

2.1 The CMM^{sm} and the PSP^{sm}

The PSP has a maturity framework much like that of the CMM. The CMM was developed by the SEI with the help of leading software groups and characterizes the most effective large-scale software practices. The PSP applies the CMM to the work of an individual.
The CMM and its Key Process Areas (KPAs) are shown in Figure 2.1.1. The CMM provides the framework for effective process management. It assumes that the software practitioners will follow disciplined personal methods. The PSP provides the framework for disciplined individual work and it assumes effective process management. The KPAs in bold italics and noted with an * are at least partially addressed by the PSP [1].



* PSP Key Process Areas
Level 5 - Optimizing
Process change management*
Technology change management*
Defect prevention*
Level 4 - Managed
Quality management*
Quantitative process management*
Level 3 - Defined
Peer reviews*
Intergroup coordination
Software product engineering*
Integrated software management*
Training Program
Software process definition*
Software process focus*
Level 2 - Repeatable
Software configuration management
Software quality assurance
Software subcontract management
Software project tracking & oversight*
Software project planning*
Requirements management
Level 1 - Initial

Figure 2.1.1: CMM and the PSP

2.2 A PSP Overview

The PSP progression is shown in Figure 2.2.1. The PSP provides a defined personal process for developing software products. It uses a software practitioner's current design and development methods to create a baseline. It guides the software practitioner to gather data on his work (e.g., time spent by phase, defects injected and removed by phase, etc.). The PSP also provides feedback to the software practitioner via its summary reports.
PSP0 establishes a measured performance baseline. This baseline includes some basic measurements and a reporting format. It also provides a consistent basis for measuring progress and a defined foundation on which to improve. PSP0 is usually the process currently used by the practitioner, but enhanced to provide measurements. It includes the design, code, compile, and test phases. During the PSP training, one programming assignment is completed using PSP0. 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 problems, experiences, and improvement suggestions. During the PSP training, two programming assignments are completed using PSP0.1.

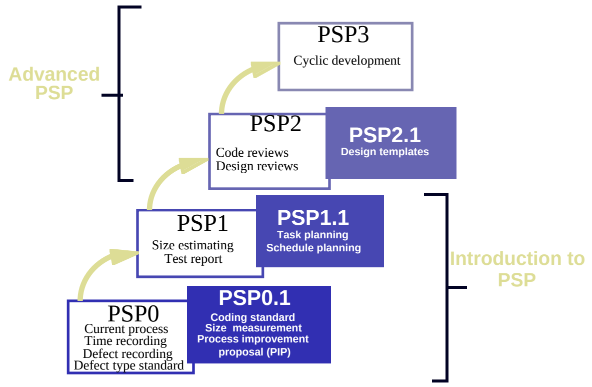


Figure 2.2.1: PSP Overview

Planning steps are added to PSP0 to create PSP1. Size, resource, and schedule plans are made with PSP1. The initial increment (PSP1) adds size and resource estimation and a test report. During the PSP training, one programming assignment is completed using PSP1. Task and schedule planning are introduced in PSP1.1. Explicit, documented plans for work help the practitioner understand the relationship between program size and development time, make commitments that can be met, produce an orderly plan for doing the work and a framework for determining the status of the work. Once a practitioner knows his own performance rate, he can plan his work more accurately, make commitments more realistically, and meet those commitments more consistently. During the PSP training, two programming assignments are completed using PSP1.1.
To enable a software practitioner to manage his defects he must know how many he makes. PSP2 adds review techniques to PSP1 to help the software practitioner find defects early when they are least expensive to fix. Defect

and yield management are practiced with PSP2 to show the software practitioner how to deal realistically and objectively with program defects that result from his errors. This is accomplished by gathering and analyzing the defects found in compile and test for the earlier programs. This data is then used by the software practitioner to establish tailored review checklists and make his own process quality assessments. During the PSP training, one programming assignment is completed using PSP2. The design process is addressed with PSP2.1. The PSP does not tell a software practitioner how to design but how to complete a design. PSP2.1 establishes design completeness criteria and examines various design verification and consistency techniques. This same approach can be used with many process phases, including requirements specification, documentation development, and test development. Phase completion criteria are important because without them, a software practitioner cannot do an effective review of the work completed in that phase. During the PSP training, two programming assignments are completed using PSP2.1.
PSP3 scales up the PSP methods to larger projects. The PSP3 strategy is to subdivide a larger program into PSP2-sized pieces. The first build is a base module or kernel that is enhanced in iterative steps. The cyclic PSP3 process effectively scales up to large programs only if each successive increment is high quality. If this is the case, the software practitioner concentrates on verifying the quality for the latest increment without worrying about the earlier ones. During the PSP training, one programming assignment is completed using PSP3 [1].

3. ANALYZING PSP DATA

I will now use the data from my PSP Instructor training to illustrate some of the increases in quality and productivity that result from using the PSP. I will analyze the time spent developing the ten programs, the defect and quality data for these programs, and the quality of the process used to develop these programs.

3.1 Program Size

For program size, the PSP objective is improved accuracy (Figure 3.1.1). Program size is measured as Lines of Code (LOC). LOC is defined in the PSP as all added or modified LOC. It is measured using an automated LOC counter developed as one of the PSP programming assignments along with a LOC counting standard and a coding standard. Estimates vary widely and neither extreme is good or bad. Typical causes of the wide variation include over engineering the product or the Object LOC Table not providing accurate data for the estimates. The estimates are compared to the actuals to see the improvements (Section 3.2).

3.2 Size Estimating Error

For size estimates the PSP objective is balanced estimates (Figure 3.2.1). That is about as many over as under. The software practitioner should also attempt to reduce estimating bias by following a consistent estimating pattern and not worry about prior errors. Wide variations are expected and remember, the initial estimates were made without any historical data. My accuracy should have improved with PSP's PROBE method (i.e., students typically end up in the +/- 10% range [2]), but I was making my estimates using an Object LOC Table for C++ since I had insufficient data available to create a personal Object LOC table for Ada.

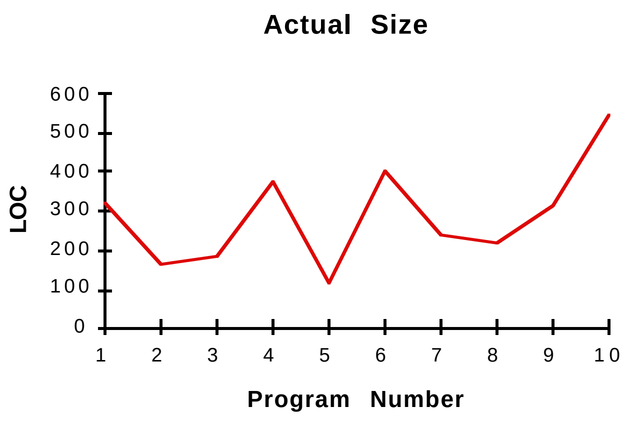


Figure 3.1.1: Program Size

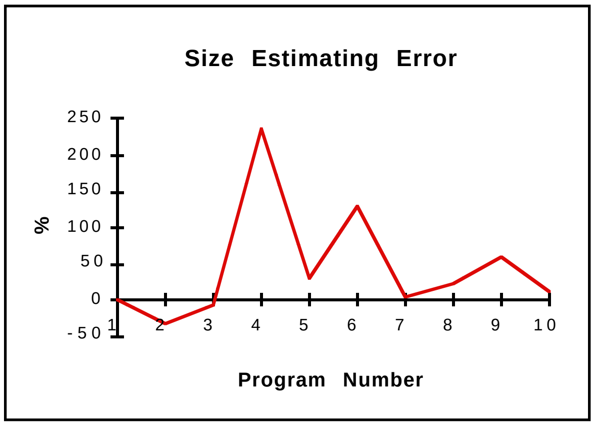


Figure 3.2.1: Size Estimating Error

3.3 Actual Development Time

For actual development time, the PSP objective is improved accuracy (Figure 3.3.1). The increased quality of the process and increased quality of the data collected will have an effect on development time. A wide variation is expected with neither extreme being good or bad. Some causes of the wide variations include over engineering the solution or a lack of experience with the development environment.

Again the estimates are compared to the actuals to see the improvements (Section 3.4).

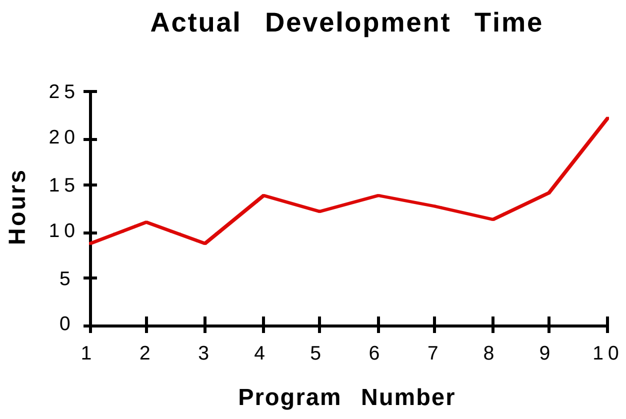


Figure 3.3.1: Actual Development Time

3.4 Time Estimating Error

For time estimates the PSP objective is improved accuracy of the estimates (Figure 3.4.1). Wide variations are expected. The initial estimates were made without any historical data. My accuracy should have improved with PSP's PROBE method (i.e., students typically end up in the +/- 10% range [2]), but I had to use the averaging method instead of linear regression since my data was not well correlated.

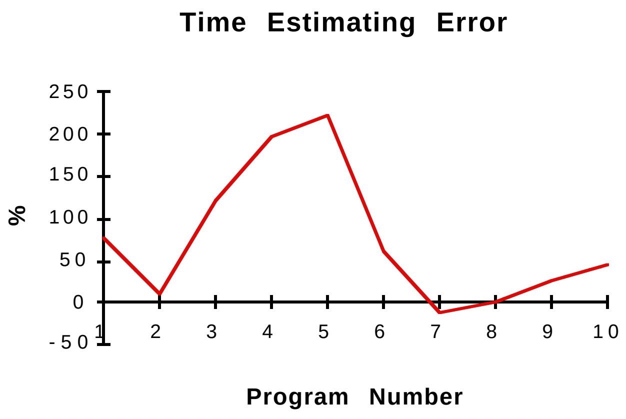


Figure 3.4.1: Time Estimating Error

3.5 Compile Time

For compile time the PSP objective is to reduce the amount of total project time spent in the compile phase (Figure 3.5.1). This is an early indicator of process quality since it is really the first independent check of the software product. A dramatic decrease occurred with the introduction of the design and code reviews with tailored checklists at Program 7A. This really amazed me since I used "code reviews" on Programs 3A through 6A, but without the tailored checklists based on my defect data.

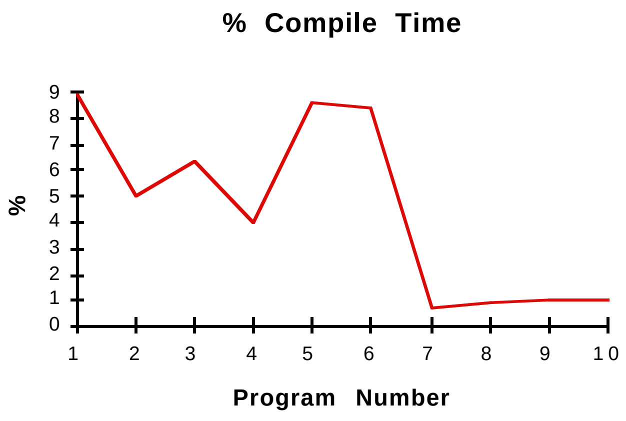


Figure 3.5.1: Compile Time

3.6 Test Time

For test time the PSP objective is to reduce the amount of total project time spent in the test phase (Figure 3.6.1). This is another early indicator of process quality since defects take time to find and fix and these defects are more costly to remove. A dramatic decrease can be seen again at Program 7A, when design and code reviews with tailored checklists were introduced.

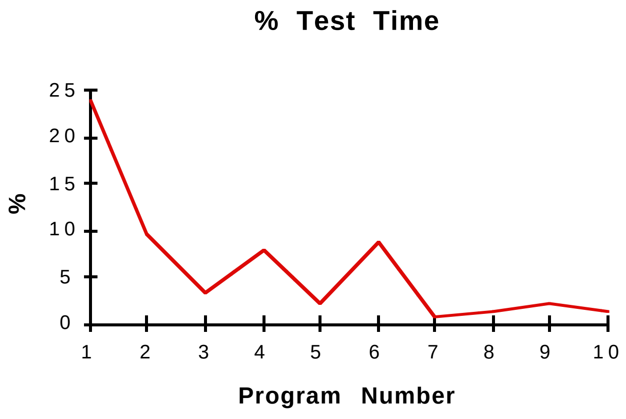


Figure 3.6.1: Test Time

3.7 Total Defects

For defects the PSP objective is to reduce the number of defects injected into a software product (Figure 3.7.1). A defect is counted every time a program change is made. Since defect removal is expensive regardless of the method used, it is most desirable to reduce the total number injected into a product. Total defects/KLOC is calculated: 1000*(Total defects removed)/Total new & Changed LOC). PSP data shows that even experienced software practitioners inject about 100 defects per 1000 lines of code. Typically, about 25 of these defects find their way into integration and systems test, at a cost of from 10 to 40 programmer hours each. Each of these defects will cost thousands of dollars to find, to repair, and to verify. Allow a defect to make its way to the field and the repair and fix costs quickly rise [1].

Wide variations are expected since the defects per KLOC measure is used (i.e., same number of defects but larger programs). Software practitioners need to periodically review the source of their defects (e.g., analyze their PSP defect log data) and act on defect prevention. A dramatic decrease can be seen again with the introduction of the design and code reviews with tailored checklists at Program 7A.

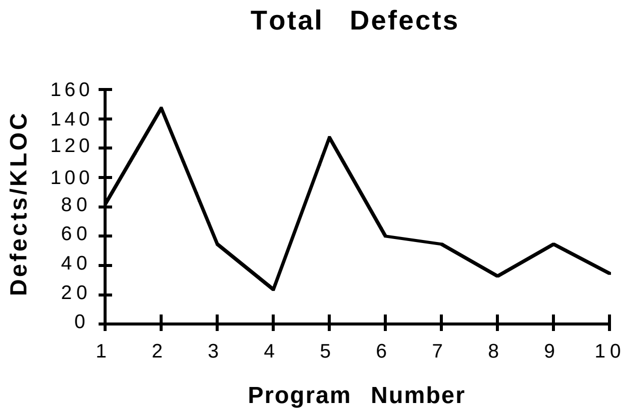


Figure 3.7.1: Total Defects

3.8 Defects Removed In Compile

For defects the PSP objective is to reduce as a percentage of the total defects removed, the number of defects removed in the compile phase (Figure 3.8.1). This is an indicator of process quality. The initial decline may be due to increased awareness and more care taken in the earlier phases. This should decrease over time as more quality activities are introduced earlier in the software development process (i.e., design and code reviews). This is also an indicator of the effectiveness of the design and code review checklists. For example Programs 3A through 6A used “code reviews” without the tailored checklists.

3.9 Compile Versus Test Defects

Look at the trend in Figure 3.9.1. By analyzing the defect data for the programs written, we see that many compile defects implies many test defects.

3.10 Design Defects

Look at the trend in Figure 3.10.1. The design defects increased with the design reviews, but this increase is good. It is less costly to remove these defects at this early phase of the project. There may have been more though that were labeled as code defects. I think Ada, with its separate specifications helped me to keep this number low.

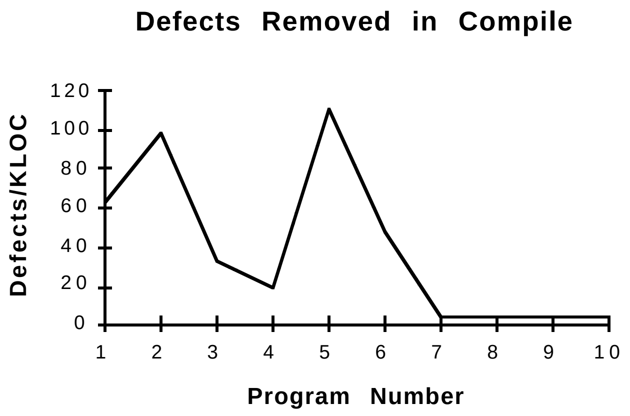


Figure 3.8.1: Defects Removed In Compile

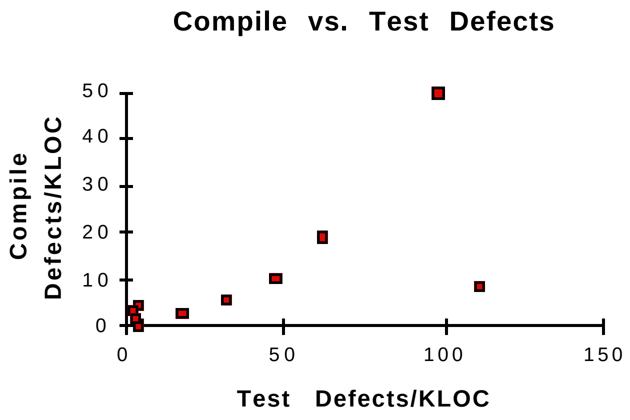


Figure 3.9.1: Compile Versus Test Defects

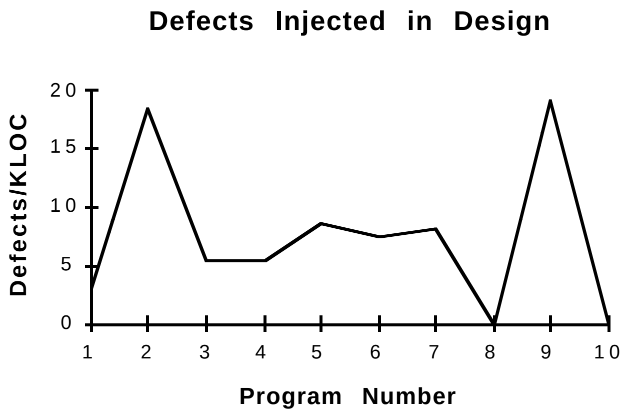


Figure 3.10.1: Design Defects

3.11 Code Defects

Look at the trend in Figure 3.11.1. Code defects may increase with effective code reviews, but they should decrease over time. The graph in Figure 3.11.1 shows the coding process is stable. The reduction in dispersion is a sign of improving process quality.

3.12 Total Cost of Quality (COQ)

The cost of quality has three components. One component is failure costs. These are the costs of diagnosing a failure, making necessary repairs, and getting back into operation. In the PSP, this is the total time spent in the compile and test phases. Another component is appraisal costs. These are the costs of evaluating the product to determine its quality level (e.g., costs of running test cases, costs of compiling when there are no defects, etc.). In the PSP, this is the total time spent in the design and code review phases. The third component is prevention costs. These are the costs associated with identifying the causes of the defects and the actions taken to prevent them in the future. These costs are not addressed in the PSP since they involve cross project activities.

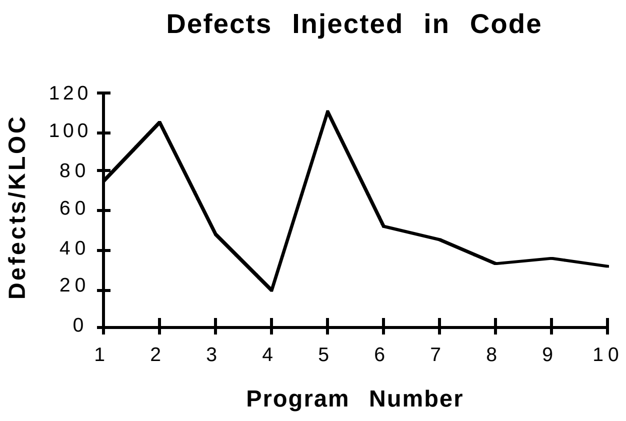


Figure 3.11.1: Code Defects

Look at the trend in Figure 3.12.1. It is in the 20% area which indicates that it is high and stable. Adding the design and code reviews did not change it much because review time was shifted out of the time previously spent in the compile and test phases. But now errors are found earlier in the process which implies the quality of the finished software product is increasing.

3.13 Defects Removed In Test

For defects the PSP objective is to reduce, as a percentage of the total defects removed, the number of defects removed in the test phase (Figure 3.13.1). This is an indicator of process quality. This should decrease over time as more quality activities are introduced earlier in the software development process (i.e., design and code reviews). This is also an indicator of the effectiveness of the design and code review checklists. Does this mean Crosby is right when he claims increased quality is free [4]?

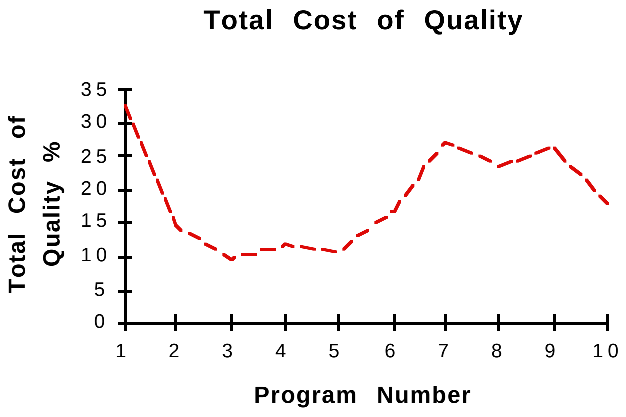


Figure 3.12.1: Total Cost of Quality (COQ)

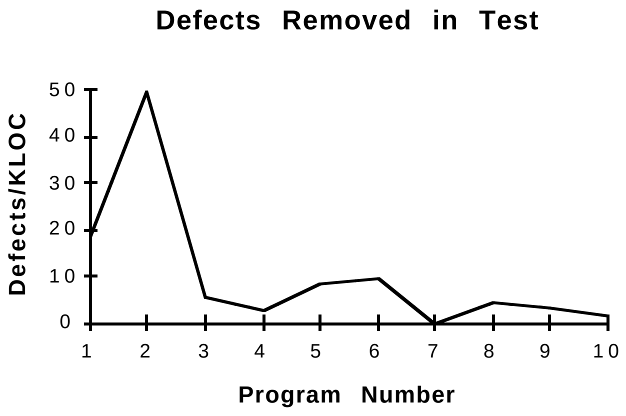


Figure 3.13.1: Defects Removed In Test

3.14 Appraisal Cost Of Quality

Look at the trend for appraisal costs as a percentage of the cost of quality (Figure 3.14.1). It should grow early with the addition of the early review steps and peak at Program 7A. If the defects removed in test (Figure 3.13.1) is compared with the appraisal cost of quality (Figure 3.14.1) a clear association between high appraisal costs and low test defects and a high association between high appraisal costs and improved product quality can be seen.

3.15 Appraisal To Failure Ratio (A/FR)

The appraisal to failure ratio is the ratio of appraisal costs to failure costs and indicates the degree to which the process attempts to eliminate defects prior to the compile and test phases. Adding the design and code reviews certainly raised my A/FR quickly, as seen in Figure 3.15.1. I was trying to achieve the PSP goal of 100% yield, that is finding all the software product defects before entering the compile phase. Software practitioners should look to obtain an upper limit on their A/FR. The PSP suggests a value of two. This means the software practitioner spends about twice as much time in the design and code review phases as in the compile and test phases [1].

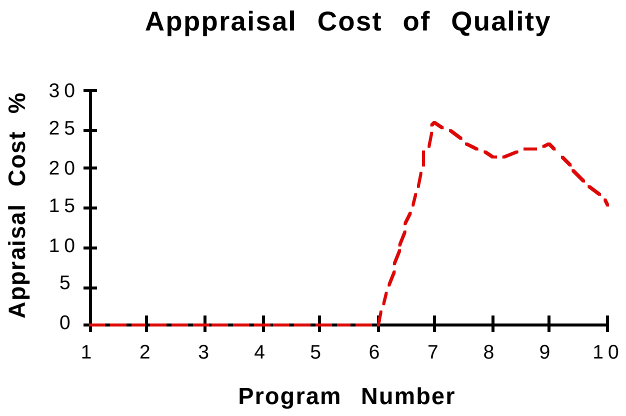


Figure 3.14.1

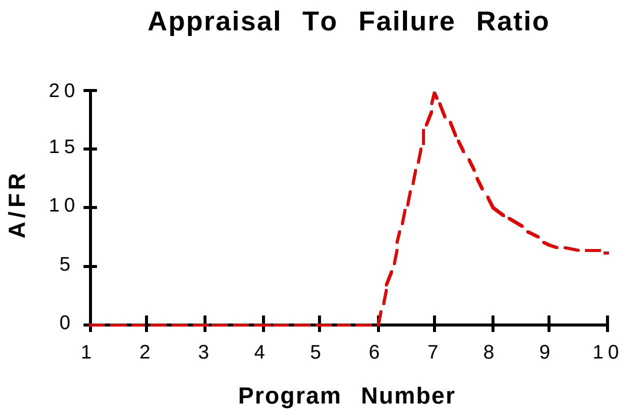


Figure 3.15.1: Appraisal to Failure Ratio (A/FR)

3.16 Defect Removal Yield

Look at the trend in Figure 3.16.1. The PSP objective for the defect removal yield or yield is 100%. That means all defects are found and removed before the compile phase and the test phase. The design and code reviews with tailored checklists helped me achieve an average yield of 84% on my last four programs.

3.17 Test Defects Versus A/FR

Look at the trend in Figure 3.17.1. Larger A/FRs should yield fewer test defects. Figure 3.17.1 shows that the number of test defects declines quite sharply with increases in my A/FR. Therefore, a high A/FR value is clearly associated with low test defects and thus is a useful indicator of a quality software process. Figure 3.17.1 also shows the relationship between process quality and product quality.

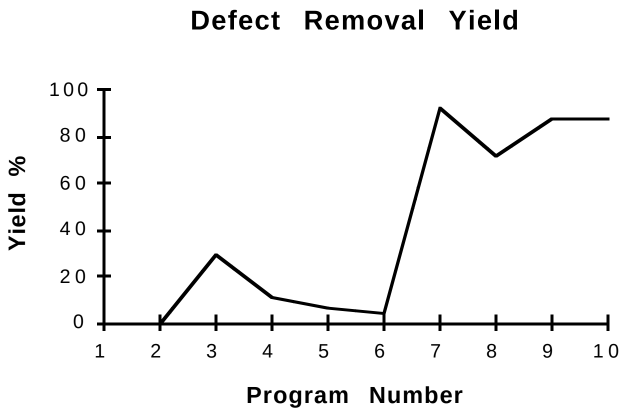


Figure 3.16.1: Defect Removal Yield

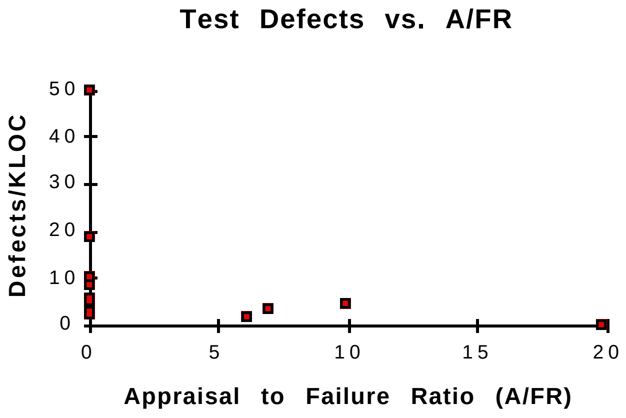


Figure 3.17.1: Test Defects Vs. A/FR

3.18 Productivity

Look at the productivity trends in Figure 3.18.1. Productivity was not significantly affected by adding all those process steps (i.e., planning, design reviews, code reviews, postmortem, measurement, analysis). The shape in Figure 3.18.1 is very similar to my actual program size graph (Figure 3.1.1).

3.19 Yield Versus A/FR

Look at the trend in Figure 3.19.1. Large A/FRs should produce better yields. The PSP goal is to maintain the high yields while reducing the A/FR.

3.20 Productivity Versus Yield

Look at the trend in Figure 3.20.1. My productivity versus yield remained relatively flat. Does this also mean that increased quality is really free? While there is no clear relationship in my data, it appears that higher yields are associated with higher productivity.

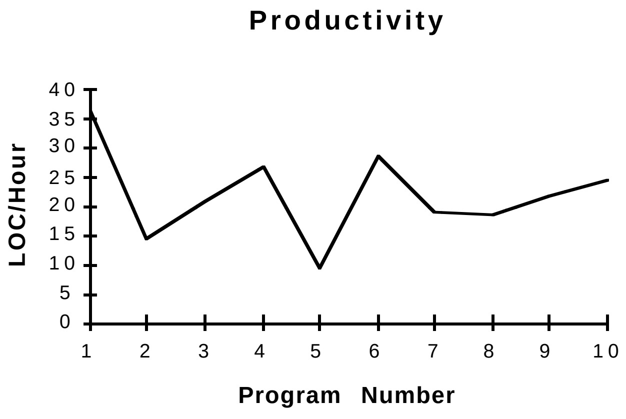


Figure 3.18.1: Productivity

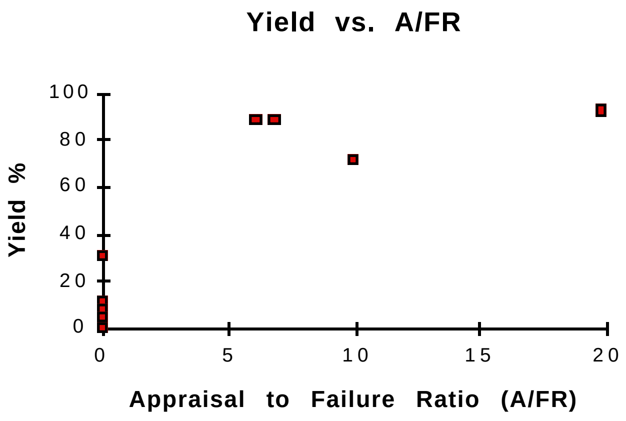


Figure 3.19.1: Yield Versus A/FR

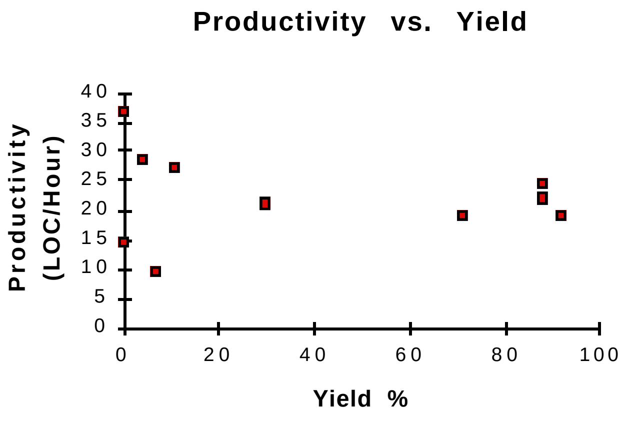


Figure 3.20.1: Productivity Versus Yield

4. COLLECTING PSP DATA

Figure 4.1 depicts the PSP flow and the basic elements of the PSP. The PSP has process scripts for planning, development, and postmortem. These scripts guide the software practitioner through the process.

During planning the software practitioner estimates development time at first and then estimates size to estimate the development time. This data is entered on the project plan summary form. During development the software practitioner develops the software product following these phases (design, code, compile, and test) and tracks the time spent in each phase and the defects injected and removed in each phase. The product is designed using design methods during the design phase. This design is implemented during the code phase. The developed code is compiled until defect free during the compile phase. The compiled product is then tested until it is defect free during the test phase. The time spent in each phase is recorded on the time recording log. The defects found and removed are recorded on the defect recording log. This includes the type of defect, the phases injected and removed, and the find and fix time. A defect type standard is used to define categories for the defects.
During postmortem the project plan summary form is completed. The actual time spent on the program and the defects found and injected in each phase are recorded on the plan summary form. This form keeps the data accessible so that it can be used on future projects.

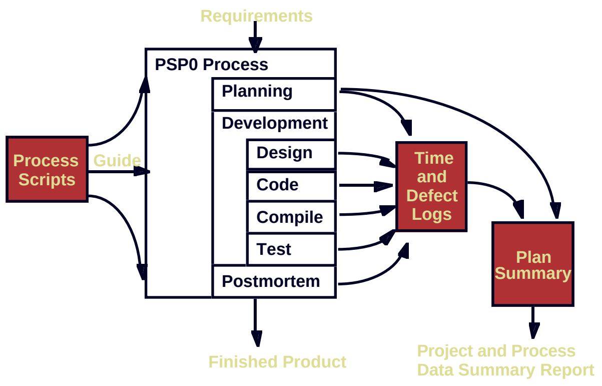


Figure 4.1: The PSP Flow

In the PSP, forms and templates are used to help the software practitioner gather data to improve his performance (e.g., planning and estimating accuracy, product quality, productivity, etc.). If the software practitioner does not collect accurate and precise data, then the data will not help the software practitioner improve his performance. To obtain useful data, the software practitioner must follow the process and be consistent, stick to his plan, and avoid unnecessary change [1].

4.1 Project Plan Summary

To track and improve his performance from project to project, a software practitioner needs current and historical data in a convenient, retrievable form. A software practitioner needs to know his planning history, what his actuals were, his performance to date, and the status of his current project. The project plan summary (Figure 4.1.1) holds project data in summary form (i.e., planned and actual data, to date history, time in phase, defects injected, and defects removed).

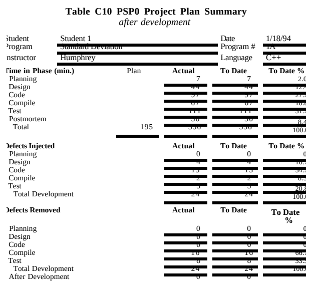


Table C10 PSP0 Project Plan Summary
after development

Student	Student 1		Date	1/18/94
Program	Standard Deviation		Program #	1A
Instructor	Humphrey		Language	C++

Time in Phase (min.)	Plan	**Actual**	**To Date**	**To Date %**
Planning		7	7	2.0
Design		44	44	12.4
Code		97	97	27.2
Compile		67	67	18.8
Test		111	111	31.2
Postmortem		30	30	8.4
Total	195	356	356	100.0

Defects Injected	**Actual**	**To Date**	**To Date %**
Planning	0	0	0
Design	4	4	16.7
Code	13	13	54.2
Compile	2	2	8.3
Test	5	5	20.8
Total Development	24	24	100.0

Defects Removed	**Actual**	**To Date**	**To Date %**
Planning	0	0	0
Design	0	0	0
Code	0	0	0
Compile	16	16	66.7
Test	8	8	33.3
Total Development	24	24	100.0
After Development	0	0	

Figure 4.1.1: Project Plan Summary

4.2 Time Recording Log

To improve the accuracy and efficiency of his process, the software practitioner needs to measure effort. The time recording log (Figure 4.2.1) records time spent in each phase. The time recording log is created in the PSP planning phase. To complete this form the software practitioner needs to know when he started and stopped, the actual minutes he worked, the length of any interruptions, and the process phase he was in, not the activity he was performing. Accuracy is important so the software practitioner should make the log entries as he works.

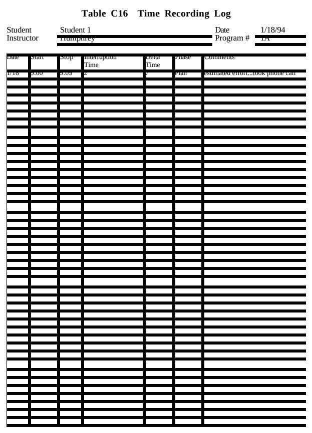


Table C16 Time Recording Log

Student	Student 1	Date	1/18/94
Instructor	Humphrey	Program #	1A

Date	Start	Stop	Interruption Time	Delta Time	Phase	Comments
1/18	9:00	9:09	2	7	Plan	estimated effort...took phone call

Figure 4.2.1: Time Recording Log

4.3 Defect Type Standard

The defect type standard provides a general set of defect categories. A software practitioner should start with simple type definitions until he has data to guide his changes. Figure 4.3.1 shows the defect type standard used with the PSP exercises.

4.4 Defect Recording Log

To improve his performance and the quality of his work, a software practitioner must keep accurate records of all defects. The software practitioner needs to know what defects he made, when they were introduced, when they were found, and how much effort was spent correcting them. The defect log (Figure 4.4.1) holds data on each defect as it is found and corrected.

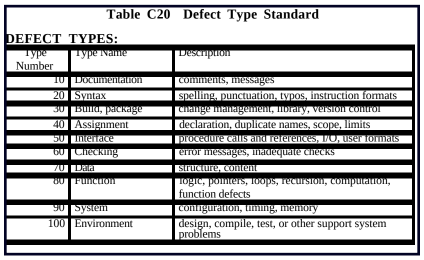


Table C20 Defect Type Standard

DEFECT TYPES:

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 4.3.1: Defect Type Standard

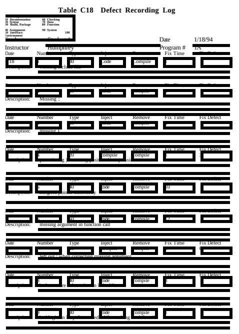


Table C18 Defect Recording Log

10 Documentation	60 Checking
20 Syntax	70 Data
30 Build, Package	80 Function
40 Assignment	90 System
50 Interface	100 Environment

Student	Student 1	Date	1/18/94
Instructor	Humphrey	Program #	1A

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect	Description
1/18	1	30	Code	Compile	1		missing include file
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]		Missing ;
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]		missing)
[illegible]	4	20	compile	compile	1	3	[illegible]
[illegible]	5	40	code	compile	10		assigned pointer incorrectly
[illegible]	6	50	code	compile	2		missing argument in function call
[illegible]	[illegible]	[illegible]	compile	compile	[illegible]	6	left out ; when correcting missing argument
[illegible]	8	40	code	compile	8		[illegible]
[illegible]	9	80	code	compile	1		[illegible]

Figure 4.4.1: Defect Recording Log

5. CONCLUSIONS

The results of all others that have completed the PSP course are described in [2]. The PSP has been shown to substantially improve the estimating and planning ability of software practitioners while significantly reducing the defects in their software products. While the results in [2] are not broken down by programming language, I believe using Ada in conjunction with the PSP not only helped me reduce the amount of time I spent in the compile and test phases but also helped me increase the quality of my software products (i.e., reduce the number of defects I found in the test phase).

I believe Ada's separate specifications helped me reduce the number of design errors generated. Using these Ada specifications in the design phase also enabled me to have a product to review in the design review phase and helped me achieve a high defect removal yield.

6. REFERENCES

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

[2] Hayes, W. and Over, J, "The Personal Software Process^{sm} (PSP^{sm}): An Empirical Study of the Impact of PSP on Individual Engineers," Software Engineering Institute Technical Report, CMU/SEI-97-TR-001 (December 1997).

[3] Paulk, M., Curtis, W., and Chrisis, M., "Capability Maturity Model for Software, Version 1.1," Software Engineering Institute Technical Report, CMU/SEI-93-TR-024 (February 1993).

[4] Crosby, P., *Quality Is Free, The Art of Making Quality Certain* (New York, NY: Mentor, New American Library, 1979).

[5] Humphrey, W. S., "Using a Defined and Measured Personal Software Process," IEEE Software, May 1996, pages 77-88.

[6] Ferguson, P., Humphrey, W., Khajenoori, S., Macke, S., and Matvya, A., "Results of Applying the Personal Software Process," IEEE Computer, May 1997, pages 24-31.