THE COLLABORATIVE SOFTWARE PROCESS

by

Laurie Ann Williams

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

The University of Utah

August 2000

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Laurie Ann Williams

This thesis had been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

_____ _____

Chair: Robert R. Kessler

_____ _____

Martin L. Griss

_____ _____

Christopher R. Johnson

_____ _____

J. Fernando Naveda

_____ _____

Joseph L. Zachary

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Laurie Ann Williams in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the supervisory committee and is ready for submission to The Graduate School.

_____ _____

Date

Robert R. Kessler
Chair, Supervisory Committee

Approved for the Major Department

_____

Robert R. Kessler
Chair

Approved for the Graduate Council

_____

David S. Chapman
Dean of The Graduate School

ABSTRACT

Anecdotal and qualitative evidence from industry indicates that *two* programmers working side-by-side at *one* computer, collaborating on the same design, algorithm, code, or test, perform substantially better than the two working alone. Statistical evidence has shown that programmers perform better when following a defined, repeatable process such as the Personal Software Process (PSP). Bringing these two ideas together, the Collaborative Software Process (CSP) has been formulated. The CSP is a defined, repeatable process for two programmers working collaboratively. The CSP is an extension of the PSP, and it relies upon the foundation of the PSP.

To validate the effectiveness of CSP, an experiment was run in 1999 with approximately 40 senior Computer Science students at the University of Utah. All students learned both the CSP and the PSP. Two-thirds of the students worked in two-person collaborative teams using the CSP to develop their programming assignments. The other students worked independently using the PSP to develop the same assignments. Additionally, a significant amount of input and confirmation from professional engineers who practice collaborative programming was factored into the research.

The research contributed a defined, repeatable process, the Collaborative Software Process, for collaborative programming pairs. The experiment validated the following quantitative findings about collaborative teams using the CSP:

1. Collaborative pairs spend approximately 15% more time than do individuals on the same task. This additional time, however, is not statistically significant.
2. Collaborative pairs achieve a higher quality level for programming products. Pairs had 15% less defects in their code. The higher quality level is statistically significant.
3. Considering the long-term field support savings of higher quality programming products, collaborative programming is cheaper for an organization than individual programming.
4. Consistently, 95% of collaborative programmers asserted that they enjoy their work more and are more confident in their work than when they program alone.

Additionally, the research resulted in many qualitative findings about collaborative programming. Most notable are the positive effects of increased problem solving skills, better designs, augmented learning, and improved team building for collaborative pairs. Organizations in which the engineers consistently switch partners also note increased communication, enhanced teamwork, and reduced product risk.

To Danny, Christopher, Kimberly and Brian
Your love is the greatest gift of all.

TABLE OF CONTENTS

LIST OF TABLES AND FIGURES

Figure Page

ACKNOWLEDGMENTS

It seems to me shallow and arrogant for any man in these times to claim he is completely self-made, that he owes all his success to his own unaided efforts. Many hands and hearts and minds generally contribute to anyone's notable achievements. *– Walt Disney*

This dissertation would not have been possible except for contributions of many hearts and minds over the years. I will begin by thanking my Ph.D. committee members; in particular, my advisor Dr. Robert R. Kessler. During my entire time at the University of Utah, Dr. Kessler has been a wise and dependable mentor and an exemplary role model in helping me achieve my professional goals. Dr. Kessler always has given me invaluable guidance, support and enthusiastic encouragement. Heartfelt thanks are also extended to other committee members Dr. Martin Griss, Dr. Christopher Johnson, Dr. Fernando Naveda, and Dr. Joseph Zachary. Your suggestions and guidance greatly improved my research.

A special thanks is given to several other people. First, Jim Coplien of AT&T Bell Labs planted the seeds in my brain from which my research hypothesis grew. He also been a wonderful friend and sounding board for me and has given graciously of his time in helping me succeed. Alistair Cockburn of Humans and Technology was also very influential in my research direction and has been great to collaborate with for several years; Alistair has also changed the way I view the people aspects of Software Engineering. Lastly, I would like to thank Dr. William Thompson. In his own way, Dr.

Thompson dramatically changed my research direction and, ultimately, my academic career path.

From the bottom of my heart, I thank my family for your unconditional love and support during these challenging years. Danny, like the song goes, “You gave me wings so I could fly. You catch me if I fall . . . Your love is the greatest gift of all.” I could not have done it without you! The absolute and unqualified love of my children, Christopher, Kimberly, and Brian gave me the strength to persevere. I always knew that even if a paper was rejected or I didn’t do as well as I would have liked on an exam or presentation, their smiling eyes would help put life back in perspective. I know I have been a very busy mom for them, but I only hope I have instilled in them the importance of life-long learning. Lastly, I want to thank my parents for their love and support. From childhood, they inspired me to always strive for excellence and gave me a love of learning.

CHAPTER 1

INTRODUCTION

> *Quality is free. It's not a gift. What costs money are the unquality things -- all the actions that involve not doing jobs right the first time. . .Quality is not only free, it is an honest-to-everything profit maker.* [1]

1.1 Research Motivation

In the early days of computing, most of the programming was done by scientists trying to solve specific, relatively small mathematical problems. The programming model that emerged from these days has been called the "code-and-fix model . . . [which] denotes a development process that is neither precisely formulated nor carefully controlled [2]." Ghezzi and others [2] describe the code-and-fix model as consisting of two steps:

1) write code
2) fix code to eliminate errors, enhance existing functionality, or add new feature

Through time, computers became cheaper and more common. More and more people started using them to solver larger and larger problems, still using and evolving the original programming model.

Alas, the code-and-fix model, often still used today, is not adequate to handle the complexities of large scale software development. Some 40 years ago, the term "Software Crisis" emerged to describe the software industry's inability to provide customers with high quality products on schedule. "The average software development project

overshoots its schedule by half; larger projects generally do worse. And, some three quarters of all large systems are "operating failures" that either do not function as intended or are not used at all [3]."

> *. . . the failure of the code-and-fix process model lead to the recognition of the so-called software crisis . . . In particular, the recognition of a lack of methods in the software production process led to the concept of the software life cycle and to structured models for describing it in a precise way in order to make the process predictable and controllable. [2]*

What is notable is the progression in the past 40 years of the visibility of the Software Crisis from mainly scientists and software developers to the general public. "Today, software is working both explicitly and behind the scenes in virtually all aspects of our lives, including the critical systems that affect our health and well-being [4]." Certainly, Y2K brought the impact of software problems to the forefront!

Unfortunately, advances in software development techniques have been thwarted by exponential increases in software complexity and size. The challenge, then, lies with bridging this gap and devising techniques to successfully handle this ever-increasing complexity. The motivation behind this research is to make an advance toward the end of the Software Crisis – to help the software industry more reliably produce high quality software.

1.2 Pair-Programming

Each day, software applications grow larger and more complicated; these applications are then used in an infinite myriad of user systems. Perhaps, then, it is best for the complexity of these applications to be tackled by two humans at a time. The idea of pair-programming, two programmers working collaboratively on the same design, algo-

rithm, code, or test, has independently emerged several times over the last decade. The practice of pair-programming is gaining popularity, primarily with the rise in the eXtreme Programming methodology [5].

Each collaborative pair sits shoulder-to-shoulder at one computer during all phases of development. One is the 'designated driver.' This engineer has control of the mouse, keyboard, or writing utensil and is actively creating the design, code, or test. The non-driver is observing the work of the driver and identifying tactical and strategic deficiencies in their work. Explicitly, the pair periodically takes turns being the driver and the non-driver. To date, anecdotal [5, 6] and preliminary statistical [7] has suggested that pairs produce higher quality code faster than code produced by individual programmers.

(Note: the terms pair-programming and collaborative programming are used interchangeably throughout this document.)

1.3 Software Process

> *Successful software engineering requires the application of engineering principles guided by informed management. The principles must themselves be rooted in sound theory. While it is tempting to search for miracles and panaceas, it is unlikely that they will appear. The best course of action is to stick to age-old engineering principles. There simply are no "silver bullets." [2]*

In the early engineering days ships sank and bridges collapsed [8]. Now, these engineering fields have matured enough that these types of accidents rarely occur because their procedures are grounded in age-old engineering principles. Generally in these fields, customers are able to enumerate very specifically the acceptable parameters and tolerance levels; these parameters and tolerance levels are clearly understood by the engineers. Then, the engineers are equipped with tools and mathematical methods to

understand the consequences of these specifications and to design accordingly. Lastly, the production and manufacturing processes in these mature fields are studied extensively in order to reliably, predictably, and efficiently produce high quality products.

Software engineering, a relatively young discipline, still seeks these verified procedures and solutions. Some computer scientists research design patterns to capture proven solutions to common design problems. Other computer scientists research mathematical methods for verifying the correctness of software algorithms. Lastly, inspired by the work of Deming [9] and Juran [10], the software engineering community has realized that it takes a high-quality software development process to yield high-quality products. Process standards such as ISO 9000 and the Capability Maturity Model (CMM) have been developed to aid organizations achieve more predictable results by guiding them to incorporate proven procedures into their process. Companies that have embraced the standards advocated in ISO 9000 and CMM have typically shown tremendous improvements. For example, by "improving its development process according to CMM "maturity," Hughes Aircraft improved its productivity by 4 to 1 and saved millions of dollars [4]."

1.4 The Research Approach

This research combines the proven need for an established, documented software process with the novel incorporation of pair-programming into such a process. A new software process, The Collaborative Software Process (CSP), was synthesized as a defined, repeatable method for two collaborating software engineers to develop software. In the CSP, recommended steps for each stage of the development process – from analy-

sis to test – is guided by detailed scripts, templates and forms. Information provided on the templates and forms can be analyzed to provide measurement-based feedback to the collaborative pair. The pair uses this feedback in order to improve the effectiveness of their own pair-process.

The research hypothesis was that *collaborative pairs following the CSP would out-perform individual engineers* using a proven process similar to the CSP, the Personal Software Process (PSP) [11] designed for solo programmers. The specific metrics used to compare the performance of individuals vs. collaborative pairs are cycle time, productivity and quality. In order to validate the hypothesis, a structured experiment was run at the University of Utah in 1999 with an upper-level software engineering class. In the experiment, collaborative pairs and individual students completed the same assignments. The above metrics were used to compare the performance of all the students.

1.5 Research Contributions

Through this research, a defined, repeatable process for collaborative programmers, the CSP, was synthesized and validated. The superiority of this process versus a known, proven, process, the PSP, was proved via a structured experiment. The experiment showed that together two pair-programmers produce software almost as fast (total elapsed time for the two) as one engineer. More notably, they produce software of statistically significantly higher quality. Because the two pair-programmers work in tandem, their cycle time is essentially half of that of individual engineers. Additionally, engineers prefer to work collaboratively. Through the documentation of the CSP, the

process can now be used by other organizations seeking to maximize the performance of their engineers.

Recently, an overwhelming amount of anecdotal and qualitative evidence has supports the use of pair-programming as a means of producing software of higher quality on schedule. However, many still resist the practice, assuming that pair-programming will prohibitively double software development costs. The only experimental variable in the structured experiment of this research was the use/non-use of collaborative programming. Therefore, the results can also be used to quantitatively support general anecdotal claims of the benefits of collaborative programming in programming environments that do not use the CSP.

1.6 Summary of Remaining Chapters

Chapter 2 provides a survey of related work. It discusses the two established software methodologies that inspired much of this work, the Personal Software Process [11] (PSP) and eXtreme Programming (XP). It discusses other anecdotal, qualitative, and quantitative emergences of the benefits of pair-programming.

Chapter 3 defines the details of the Collaborative Software Process (CSP). It describes the process steps and the rationale behind CSP. A discussion compares PSP to CSP.

Chapter 4 discusses qualitative findings of the research. First, theoretical and observed reasons for the benefits of collaborative programming are discussed. Then, factors for successful collaboration are enumerated.

Chapter 5 presents convincing quantitative evidence that CSP and collaborative programming are superior to PSP and individual programming. This quantitative analysis is based on data obtained from a carefully planned empirical study of advanced undergraduates at the University of Utah. (Details of this experiment are explained in Appendix A.) Benefits to the software firm and to the engineers are quantified.

Chapter 6 summarizes the conclusions and contributions of the dissertation. Chapter 7 suggests future research to further validate and collaborative programming.

Six appendices provide detailed background information to support each of the seven chapters.

CHAPTER 2

A SURVEY OF RELATED WORK

The Collaborative Software Process was formulated after investigating the contributions of and successes in several areas of software engineering and cognitive science.

2.1 The Personal Software Process (PSP)

Structurally, the largest influence comes from the Personal Software Process [11] (or PSP), authored by Watts S. Humphrey of the Software Engineering Institute (SEI). PSP defines a software development framework that includes defined operations or sub-processes and measurement and analysis techniques to help engineers understand their own skills in order to improve their own personal performance. Each sub-process has a set of scripts giving specific steps to follow and a set of templates or forms to fill out to ensure completeness and to collect data for measurement-based feedback. This measurement-based feedback allows the programmers to measure their work, analyze their problem areas, and set and make goals. For example, programmers record information about all the defects that they remove from their programs. They can use summarized feedback on their defect removal to become more aware of the types of defects they make to prevent repeating the same mistakes. Additionally, they can examine trends in their defects per thousand lines of code (KLOC) and are able to see when they are making real improvement.

PSP has several strong philosophies. The first is that the longer a software defect remains in a product, the more costly it is to detect and remove it. Therefore, thorough design and code reviews are performed for most efficient defect removal. The second philosophy is that defect prevention is more efficient than defect removal. Careful designs are developed, and data is collected to give additional input on where the programmer should adjust their own personal software process to prevent future defects.

Lastly, PSP rests on the notion that the best estimates, and therefore the best commitments, for schedule and defect rates can be made with a historical database of information. Data regarding how long previous products took to develop and defect rates are kept in a database for use with history-based estimation procedures. These processes and philosophies work together to produce excellent results. According to Ferguson, Humphrey and others at the SEI,

> *"SEI's data on 104 engineers shows that, on average, PSP training reduces size-estimating errors by 25.8 percent and time-estimating errors by 40 percent. Lines of code written per hour increased on average by 20.8 percent, and the portion of engineers' development time spent compiling is reduced by 81.7 percent. Testing time is reduced by 43.4 percent, total defects by 59.8 percent, and test defects by 73.2 percent* [12]*."*

The PSP is a defined, repeatable process for an individual engineer; the CSP is a defined, repeatable process for two programmers working collaboratively. The CSP is an extension of the PSP, and it relies upon the foundation of the PSP.

2.2 eXtreme Programming

CSP is also heavily influenced by success factors of the ***eXtreme Programming*** [5] (or XP) methodology, developed primarily by Smalltalk code developer and consultant

Kent Beck with colleagues Ward Cunningham and Ron Jeffries. XP does not have statistical evidence, as does PSP, to prove its effectiveness. The evidence of XP's success is highly anecdotal, but is so impressive that it has aroused the curiosity of many highly respected software engineering researchers and consultants. The largest example of its accomplishment is the sizable Chrysler Comprehensive Compensation system launched in May 1997. The payroll system pays some 10,000 monthly-paid employees and has 2,000 classes and 30,000 methods [13]. Additionally, programmers at Ford Motor Company, spent four unsuccessful years trying to build the Vehicle Cost and Profit System (VCAPS) using a traditional waterfall methodology. Then, the engineers duplicated that system, this time successfully, in less than a year using Extreme Programming [14].

XP strongly advocates the use of pair programming. All production code is written with a partner, to the extent that even prototyping done solo is scrapped and re-written with a partner. Working in pairs, the engineers perform a continuous code review, noting that it is amazing how many obvious but unnoticed defects another person at your side notices. This is, perhaps, the ultimate implementation of PSP's "defect prevention" and "efficient defect removal" philosophies.

XP's requirements gathering, resource allocation and design practices are a radical departure from most accepted methodologies, such as PSP or the Rational Unified Process [15]. Customer requirements are written as fairly informal "User Story" cards, a rough estimate of required resources is assigned to the cards, these are assigned to a programming pair, and coding begins. With no *formal* design procedures or discussions on overall system planning or architecture, the pair determines which code in the ever-enlarging code base needs to be added or changed and then does it, without asking any-

one “permission”. This practice requires the use of "Collective Code Ownership" whereby any programming pair can modify or add to any code in the code base, regardless of the original programmer.

Programming pairs routinely "refactor" the code base by continuous change and enhancement. They view the code as *the* self-evolving design – they do not spend time on a design document and, therefore, have strict self-documenting code style and comment guidelines. XP also has particularly thorough testing procedures. Comprehensive test cases are written and automated prior to actual code changes. The results of running these automated new tests and previous, regression test cases determine if the change/enhancement to implement a User Story has been done correctly without harming the implementation of other User Stories. While departing significantly from traditional development practices, anecdotally, XP appears to be very effective. Additionally, programmers report that developing with XP practices is much more exciting and enjoyable than with traditional processes. From XP, CSP incorporates the successful use of pair programming and automated test case generation and execution.

2.3 Distributed Cognition

While those practicing XP are the largest known group of pair programmers, the idea of pair programming did not originate with XP. In 1991 Nick Flor, a masters student of Cognitive Science, reported on distributed cognition in a collaborative programming pair he studied. Distributed cognition is a field of cognitive science based on the following beliefs:

> *“Anyone who has closely observed the practices of cognition is struck by the fact that the “mind” rarely works alone. The intelligences revealed*

> *through these practices are distributed – across minds, persons, and the symbolic and physical environment . . . Knowledge is commonly socially constructed, through collaborative efforts toward shared objectives or by dialogues and challenges brought about by differences in persons' perspectives.* [16]"

Flor recorded via video and audiotape the exchanges of two programmers working together on a software maintenance task. He correlated specific verbal and non-verbal behaviors of the two under study with known distributed cognition theories.

1. *The Sharing of Goals and Plans: Collaborating programmers attempt to maintain a shared set of goals and plans during interactions. Goals specify what needs to be done, and plans specify the means by which the goals are achieved. . . The sharing of goals and plans leads to several different system properties: efficient communication, searches through larger spaces of alternatives, and shared memory for 'old' alternative plans.*
2. *Efficient Communication: Conversational details do not have to be fully specified, thus minimizing the amount of talk required to encode that which must be communicated. The current state of the problem combined with the programmers' shared goals and plans are sufficient to determine the intent of most utterances.*
3. *Searching Through Larger Spaces of Alternatives: A system with multiple actors possesses greater potential for the generation of more diverse plans for at least three reasons: (1) the actors bring different prior experiences to the task; (2) they may have different access to task relevant information; (3) they stand in different relationships to the problem by virtue of their functional roles. . . An important consequence of the attempt to share goals and plans is that when they are in conflict, the programmers must overtly negotiate a shared course of action. In doing so, they explore a larger number of alternatives than a single programmer alone might do. This reduces the chances of selecting a bad plan.*
4. *Shared Memory for Old Plans: A memory for old alternative plans is useful in situations where the subjects are exploring a course of action, decide on it being unproductive, and have to backtrack to one of the possibly many, older alternative plans. A single programmer alone may forget one of these alternative plans* [17].

2.4 Organizational Pattern

In 1995 Jim Coplien published the "Developing in Pairs" Organizational Pattern [18]. Organizational Patterns make explicit the patterns of organization, process, and introspection that most highly productive organizations exhibit.

> *Using the emerging discipline of generative pattern languages, we can capture the patterns underlying successful projects and use them to establish organizational structures and practices that will improve the prospects for success in a new software development organization. [18]*

The "Developing in Pairs" pattern professes that organizations should pair compatible designers to work together – that together, they can produce more than the sum of the two individually.

2.5 Other Studies

Two other studies support the use of collaborative programming. Larry Constantine, a programmer, consultant, and magazine columnist reports on observing "Dynamic Duos" during a visit to P. J. Plaugher's software company, Whitesmiths, Ltd. He immediately noticed that at each terminal were two programmers working on the same code. He reports,

> *Having adopted this approach, they were delivering finished and tested code faster than ever . . . The code that came out the back of the two programmer terminals was nearly 100% bug free . . . it was better code, tighter and more efficient, having benefited from the thinking of two bright minds and the steady dialogue between two trusted terminal-mates . . . Two programmers in tandem is not redundancy; it's a direct route to greater efficiency and better quality.* [19]

Lastly, in 1998 Temple University Professor Nosek reported on his study of 15 full-time, experienced programmers working for 45 minutes on a challenging problem, im-

portant to their organization, in their own environment, and with their own equipment. Five worked individually, ten worked collaboratively in five pairs. Conditions and materials used were the same for both the experimental (team) and control (individual) groups. This study provided statistically significant results, using a two-sided t-test. "To the surprise of the managers and participants, all the teams outperformed the individual programmers, enjoyed the problem-solving process more, and had greater confidence in their solutions." The groups completed the task 40% more quickly and effectively by producing better algorithms and code in less time. The majority of the programmers were skeptical of the value of collaboration in working on the same problem and thought it would not be an enjoyable process. However, results show collaboration improved both their performance and their enjoyment of the problem solving process [7].

CHAPTER 3

COLLABORATIVE SOFTWARE PROCESS DEFINITION

As discussed in the previous chapter, the framework of the CSP is modeled after that of the PSP. Because of this, the motivation and rationale behind the development of the Personal Software Process will be briefly discussed. Then, the major elements of the Collaborative Software Process will be defined and described. Lastly, the PSP and the CSP will be compared.

3.1 Process Rationale

The Software Engineering Institute worked with leading software organizations to define the Capability Maturity Model for Software [20] (or CMM). The purpose of the CMM is to provide "an orderly way for organizations to determine the capabilities of their current process and to establish priorities for improvement. It does this by establishing and defining five levels of progressively more-mature process capability [11]." A more mature process is increasingly defined, repeatable and controlled and is more likely to predictably produce high quality software products. In increasing level of maturity, these five levels are: Initial, Repeatable, Defined, Managed and Optimizing.

Each of these levels has key process areas (KPA) defined. The CMM provides goals and example practices for each of these KPAs to guide organizations in achieving higher levels of process maturity. Organizations have internal or external process re-

views to determine their maturity level of their current process and to formulate improvement plans for improving their maturity level. Many times, software organizations that contract programming services are asked to evaluate and disclose their CMM level. CMM level can be an important factor in such an organization's competitive position. Humphrey, the creator of the PSP, was also instrumental in the formulation of the CMM.

Humphrey then essentially brought the philosophy of the CMM/process maturity to the level of the individual engineer by the formulation of the PSP. The PSP specifically addresses many of the KPAs. PSP defines a framework for an individual programmer striving to help their organization achieve a higher level of maturity. Undoubtedly, improvements in personal capability also improve organizational performance.

The PSP follows an evolutionary improvement approach. A student or professional learning to fully integrate the PSP into their process begins at Level 0.0 and progresses in their process maturity seven levels to Level 3.0. Each level incorporates new skills and techniques into their process – skills and techniques that have proven to improve the quality of the software process and to improve the estimating accuracy of the engineer.

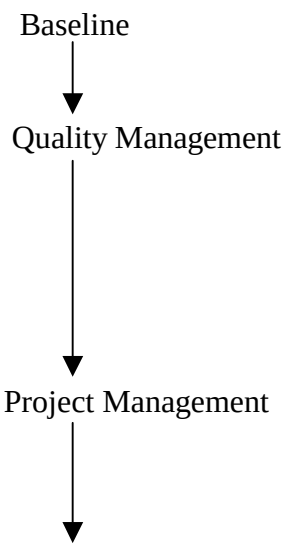
The PSP is defined as a set of 77 process scripts, forms, templates, standards, and checklists. By consistently using this documentation, a software engineer follows a proven, disciplined software development process in which he or she receives measurement-based feedback on their process effectiveness. The scripts enumerate process steps that should be followed. Forms and templates are used to obtain and store necessary data and information from the engineer in a thorough and complete manner. A

coding standard is defined to guide a consistent coding style. Checklists are provided to aid in review processes.

The CSP also incorporates the evolutionary learning approach of PSP and has six levels. These levels are summarized in Figure 1 and defined below. The CSP consists of 45 scripts, forms, templates, standards and checklists, which are documented in Appendix B. Generally, these are based on the PSP [11], but have been adapted for simplicity and to allow for the direction and analysis of a pair of programmers. Additionally, the CSP changes several aspects of PSP, particularly in the analysis and design phases. The CSP also incorporates seven sets of instructions and templates directly, unchanged from the PSP. These are referred to but are not included in this document. As with the PSP, the CSP defines a framework for the collaborative pair to help their organization achieve a higher level of maturity.

Figure 1: CSP Evolutionary Learning Approach

Level	CSP
0.0	Baseline / Current Process
0.1	Coding Standard Size Measurement Process Improvement Plan
1.0	Analysis (Use Case) CRC Card Design Brainstorming Design
1.1	Code Review Design Reviews Testing Measurements
2.0	Size Estimating Resource Estimating
2.1	Task Planning Schedule Planning



3.2 CSP Definition

3.2.1 CSP Level 0: Collaborative Baseline

3.2.1.1 CSP Level 0.0

CSP Level 0.0 does not impose or recommend any additional process steps; the engineers use their "natural" process. In Appendix B, Table 5 (page 106), Table 6 (page 107), and Table 7 (page 108) document Process, Planning, and Development scripts which enumerate steps for the engineer to take. However, these steps are very general and would have to be followed by any engineer who developed software. Examples of these steps are: 1) Produce a design to meet the requirements; 2) Implement the design; 3) Compile the program.

The purpose of this level is to provide baseline measurements from which to compare results of future process improvements. Therefore, the only addition to their "natural" process is to record time and defect data about their development work. Table 11 - Table 14 (pages 113 - 116) are forms and instructions for recording this information. The engineers must do their best to diligently record the amount of time they spent on each phase of the development process and to record information about the defects they remove during their review, compilation and testing phases.

The Postmortem script (Table 8 on page 110) prescribes the completion of the Project Plan. First, prior to beginning development, the pair makes an overall estimate of how long it will take to develop the product. The recorded time and defect data are summarized in the Project Plan (Table 9 on page 111) for use in analyzing the pair's process and for making future estimates. The table easily facilitates the comparison

between how long the pair thought it would take to develop a product and how long it actually took (by phase).

One other important thing happens at this level, particularly if the engineers have never worked in pairs before – they jell as a team. Most programmers have been conditioned to work individually – and switching to collaborative programming is certainly an adjustment. Many engineers venture into their first pair programming experience skeptical that they would actually benefit from collaborative work. They wonder about coordinating schedules, the added communication that will be required, about adjusting to the other's working habits, programming style, and ego, and about disagreeing on aspects of the implementation.

In industry, this adjustment period has historically taken hours or days, depending upon the individuals. In the university experiment run as part of this research, the students generally adjusted after the first assignment, though some reported an even shorter adjustment period. It doesn't take many victorious, clean compiles or declarations of "We just got through our test with no defects!" for the teams to celebrate their union – and to feel as one jelled, collaborative team.

3.2.1.2 CSP Level 0.1

At the CSP Level 0.1, several small process improvements are made. The engineers begin to follow a coding standard. Groups of individuals who follow a coding standard can be expected to have similar coding styles. This is particularly beneficial for collaborative pairs as each takes turns adding to and reviewing their partner's code. It is also advantageous for software maintenance when field support must read and understand

the code of many different programmers. (A sample C++ coding standard is documented in Table 23 on page 128).

Engineers also count and record their number of lines of code as a measure of software size. To no avail, software engineers perpetually debate the best measure of software size. While lines of code may be an imperfect measure of product size it is satisfactory for meeting the goals of the CSP. When used in conjunction with a coding standard, particular collaborative pairs can use the line of code measurement to compare relative size of their various programs.

The Project Plan is updated to incorporate recording the line of code measurement. Additionally, the estimate of development time is entered at the process phase (Planning, Design, Code, Compile, Test, Postmortem) level. Pairs estimate the proportion of development time they expect to spend by phase by reviewing the historical data they began recording while using CSP Level 0. The Project Plan and Postmortem (Table 18 - Table 20 on pages 121 - 124) are adjusted accordingly.

Lastly, after each program, pairs reflect on their process – what went well and what didn't go so well about the software development process they actually used for that program – and record these observations in the Process Improvement Proposal (PIP). The purpose of this document is to impress upon the pair what they should and should not do in the future in order to be most effective. The PIP and the instructions for completing the PIP are in Table 21 and Table 22 (pages 126 - 127).

3.2.2 CSP Level 1: Collaborative Quality Management

Why spend all this time finding and fixing and fighting when you could prevent the incident in the first place?

[P.B Crosby in *Quality is Free* [1]]

Collaborative programmers get used to working in pairs and taking some very basic measurements in CSP Level 0. The attention is turned toward introducing particular activities to improve product quality in Level 1. "The goal of quality management in the PSP is to find and remove all defects before the first compile [21];" CSP shares this noble goal.

3.2.2.1 CSP Level 1.0

In Level 1.0, attention is focused on the first stages of the development process, analysis and design. The analysis phase deals with understanding the problem, goals and constraints of the program. [22] enumerates the goals of performing analysis:

- To understand the problem that the eventual software system, if any, should solve
- To prompt relevant questions about the problem and the system
- To provide a basis for answering questions about specific properties of the problem and system
- To decide what the system should do.
- To decide what the system should not do.
- To ascertain that the system will satisfy the needs of its users, and define customer acceptance criteria
- To provide a basis for the development of the system

In the CSP, analysis is performed through the development of use cases [23] based on the customer requirements. The use cases are documented using the UML Use Case Model [24]. The first step in developing the use cases is to identify the actors, the people or systems that are external to the system but act upon or with the system. Then, the

use cases themselves can be discovered. A use case is a sequence of transactions performed by a system that yields a measurable result of values for a particular actor. A use case typically represents major functionality that is complete from beginning to end. Through the identification of use cases, scenarios, which are used later in the process, are identified. "A use case is an abstraction that describes all possible scenarios involving the described functionality. A scenario is an instance of a use case describing a concrete set of events. . . Scenarios are used as examples for illustrating common cases -- their focus in on understandability. Use cases are used to describe all possible cases -- their focus is on completeness [25]."

In the CSP, each use case is explored by completing a Use Cases Flow of Events template [26] as shown in Table 28 (on page 135) in Appendix B. The completion of the Flow of Events serves to clarify the engineer's thoughts on what the system should and should not do. The Use Case Model and Flow of Events are both very readable and understandable by non-technical customers. They, therefore, can be shown to customers to ascertain that the system meets the customer requirements. The development of these artifacts leads to the early stages of design as relationships between the use cases are explored. Therefore, the goals of analysis, as stated above, are achieved through the creation of the Use Case Model and Use Case Flow of Events. In Appendix C, the requirements for a small program are developed into a Use Case Model and Flow of Events.

The Use Case Flow of Events is also highly beneficial for black box test case development. Engineers can identify many paths through the flow of events and devise test cases to validate the correctness of the program for that set of conditions. The "Alterna-

tive Flows" identified in the Flow of Events is very beneficial for identifying error conditions which must be handled in the program and tested to ensure proper handling.

Once analysis is completed in the CSP, a CRC card brainstorming exercise is held as a predecessor to high-level design. (CRC stands for **C**lass, **R**esponsibility, and **C**ollaborator) The exercise is performed to facilitate the process of identifying the system's objects and their public interfaces [27]. Index cards are used to identify classes, their responsibilities, and which other classes they must collaborate with to perform their services. A format of a typical CRC card is shown below in Figure 2. (Often the class attributes are written on the back of the card.)

The scenarios identified by the use cases are "role played" using the cards – to ensure that the classes perform the necessary services to complete each scenario. It is best to choose a set of use cases that look like they would touch a related set of classes. (Sets of scenarios that touch different sets of classes should have their own CRC card exercise. This segmentation allows for more manageable brainstorming sessions.)

Figure 2: CRC Card Format

Class Name	
Main Class Responsibility (one sentence)	
Responsibilities . . .	Collaborators . . .

A CRC card session proceeds as follows. A particular scenario is chosen from a use case (e.g. one particular flow through the use case flow of events). The engineers role play what the code would need to do in order for that scenario to complete successfully. When a new class would need to be created to fulfill the requirements of the scenario, a blank card is put on the table. The name and purpose of the class is written on the card. Once classes are identified, any responsibility identified as part of the scenarios walkthrough is written down on the class's card under the Responsibility section. If the class must collaborate with another class in order to complete its responsibility, that class is listed across from the responsibility in the Collaborator column. A representative set of scenarios must be role played. For each scenario role play, participants point to or pick up cards that would be used to handle the responsibility if the classes and responsibilities have already been defined and/or initiate the creation of new classes and responsibilities.

It is expected that classes will be identified and later discarded during the course of the brainstorming session. The exercise allows several design alternatives to be on the table at one time. Classes that seem to be uneeded are pushed to the side of the table but not discarded. "An unpopular initial design may turn out to be a popular later design, or perhaps the final design is a small alternation of an initially rejected design. [28]"

Through this process, the engineers ensure that the classes have been well formulated and that they have the necessary behavior-responsibility (via their methods) and knowledge-responsibility (via their attributes) to handle a representative set of scenarios. From the CRC card exercise the high-level/UML class design almost falls out – because the classes have been identified as well as the required methods and attributes.

Formulating the official Object Model and associated interaction diagram is done as a formalization of the CRC card exercise and serves as the high-level design.

Lastly, cyclic development is encouraged. It is recommended that once high-level design and review are completed, the pair break their project into pieces that appear to be between 100 and 300 lines of code. The pair should then iteratively perform low-level design, code, review, compile and test for each of these increments. Future increments should be built upon the growing code base of past increments.

3.2.2.2 CSP Level 1.1

For over 20 years, numerous studies have documented the benefits of reviews and inspections for efficient defect removal (some selected references are [11, 29, 30, 31]). At CSP Level 1.1**,** both design and code reviews are introduced. During these reviews, the pair of programmers examines their own work products.

Even with the non-driver performing constant reviews, the pair still needs to step back from the computer and review their work against prescribed design and code review checklists. Realistically, even with collaborative pairs, some work will be done individually due to illness, time conflict, or by conscious choice. (For example many pair programmers have found that rote, routine coding is more effectively done alone.) Therefore, the CSP has two versions of the design review checklist (Table 37 and Table 38) and two versions of the code review checklist (Table 35 and Table 36) – one version of each for individual work and one version of each for collaborative work. Work done individually must be very carefully checked before being incorporated into the shared code base. Therefore, the checklists for individual work are quite thorough. However,

work performed by both partners does not require as thorough a formal review because the non-driver performs a constant review. Reviews of collaborative work focus on overriding factors like “Does the design cover all items in the specification?” or “Did we completely implement our design?” Reviews of individual work also check for syntax and lower-level logic errors.

Sample checklists can be found in Appendix B. Some items in the design review checklists were taken from [32]. It must, however, be emphasized that these checklists should be considered dynamic for each collaborative pair. If a pair never makes a particular mistake, this item should be taken off the checklist. Other errors the pair is prone to making should be added to the checklist.

Level 1.1 also introduces black box, white box and automated regression testing techniques. Initial black box test cases are written early, in the design stage. The philosophy behind this is that if you diabolically think about “how can I break this code” and write test cases to see if you have or not, you will design and code in order to pass your own test cases. Also, the design phase is a relatively calm, thoughtful phase of development, conducive to thinking clearly and thoroughly about test cases. When a project is in the chaotic throes of testing with a deadline looming, the development of a complete set of test cases is often compromised. For each test case, the Test Case Template (see Table 39 on page 151 in Appendix B) is completed. Additional information about the black box test cases is added, as more implementation issues are resolved.

As is done in Extreme Programming, white box unit test cases are incrementally written and added to an automated regression test suite prior to writing the actual code. Writing unit test cases before coding allows you to verify that you really understand the

requirements and can even help clarify implementation issues. Ron Jeffries, one of the Extreme Programming principles discusses Extreme Testing [33]:

> *We need to be sure of two things: the new capability works, and we haven't broken anything that used to work. And that requires testing. There are two things to be sure of, so Extreme Testing specifies two key actions:*
>
> *1. To be sure that new features work, write Unit Tests for every feature. Write them before you release the code, preferably before you even write it. Save all the unit tests for the whole system.*
>
> *2. To be sure that nothing else is broken, run all the Unit Tests in the entire system before any code is released – and ensure that those tests run at 100 percent!*
>
> *Let me emphasize that last point. Whenever Extreme Programmers release any code at all,* ***every unit test in the entire system must be running at 100%!*** *That shows us not just that the new feature works, but that the changes haven't broken anything anywhere.*

This iterative process of "design-a-little code-a-little test-a-little" allows development to proceed with confidence that code is correct. It also improves defect removal efficiency because if a test case fails, the engineer can be assured that the new code caused the fault. "Software release goes much faster when you run the tests before every release, because if anything breaks you know almost exactly where the problem is. Developers who work with tests get to spend more time working with new code, and less time trying to find obscure bugs in old code [33]."

Appendix E has a sample design and example for the automated regression tester. A philosophy behind the tester is that it is tedious and error-prone to visually inspect program output to see if tests passed. Testing is then accomplished by formulating tests as collections of Boolean expressions and having the test program report a summary of passes and failures. "You can't have comprehensive repeatable tests if you have to

manually check the results. Have a testing facility to set up and run the tests, check the results, and report them [33]."

Additionally, Table 41 on page 153 in Appendix B has a test coverage script that helps with the review of completeness of the both black box and white box test case sets.

The last thing added to the Level 1.1 is measurements, which examine the effectiveness of the quality initiatives of Level 1. The measurements mirror the measurements of the Personal Software Process [11]. Beginning with Level 0.0, engineers record data on the time they spend and the defects they remove. In this level this data is turned into significantly more information in order to provide measurement-based feedback. This information can provide critical feedback so they can effectively critique their own work and adjust their pair-process. Each of the measurements that are introduced in Level 1.1 is briefly explained:

Yield: Yield is the percentage of defects that were in a program during a particular phase that were removed during that phase. A high yield is good; a low yield is poor. The yield measurements demonstrate how good a "filter" for removing defects a particular phase was. Yield can be measured because in the CSP defect recording log engineers record their best guess at the phase the defect was injected and the phase it was removed. Process yield is the percent of total defects that were removed prior to the first compile:

Equation 1: Process Yield

$$\text{Process Yield} = \frac{100 * (\text{defects found before the first compile})}{\text{Total defects found}}$$

With practice and experience, an 80% process yield is an excellent goal to strive for [11].

Cost of Quality (COQ): A measure of the amount of time spent to achieve a quality product. COQ has two components. One component of COQ is failure costs or the cost to diagnose a failure and to make necessary repairs.

Equation 2: Failure Cost of Quality

Failure Cost of Quality = 100 * (compile time + test time) / (total development time)

Another component of COQ is appraisal costs. Appraisal costs are the costs to evaluate a product to determine its quality level and can be calculated:

Equation 3: Appraisal Cost of Quality

Appraisal Cost of Quality = 100 * (design review time + code review time) / (total development time).

These two components are summed to get the Total COQ. The Appraisal to Failure Cost ratio (A/FR ratio) is also calculated from these measures.

Equation 4: Appraisal to Failure Ratio (A/FR)

A/FR = (Appraisal COQ) / (Failure COQ)

The A/FR ratio is a good indication of the "degree to which the process attempts to eliminate defects prior to compiling and test phases [11]." A high A/FR is associated with low test defects.

Defect Removal Efficiency: This is a measure of the number of defects that are removed per hour in each defect removal phase. The measure is used to indicate the relative defect removal efficiency of each phase. It is calculated by dividing the total number of defects found during a phase by the amount of time spent in that phase.

Defect Removal Leverage: This measure is used to directly compare the relative effectiveness of the defect removal phases. It is a ratio of the above Defect Removal Efficiency in any two phases. It is most often the ratio relative to the test phase as an indication of how much more efficient a pre-test phase is at removing defects when comparing with the test phase.

It should be noted that these measurements are tedious to calculate. Tool support to calculate these measurements from raw time and defect data is absolutely essential for data accuracy and in order for the methodology to be practical. Various tools have been developed and used to perform these calculations. The SEI provides an Excel spreadsheet program to track the data and perform the calculations. The students involved in this research used a web-based tool, which stored the data on an NT server. The students could then use the tool in the university laboratories, from their workplace, or

from home and could effortlessly combine their date with their partners when they chose to work separately. The web-based tool was developed as part of this research.

3.2.3 CSP Level 2: Collaborative Project Management

Level 2 is concerned with adding sound project management activities to the collaborative team's process. "Project management includes the oversight activities that ensure the delivery of a high-quality system on time and within budget [25]." The project management techniques of the PSP are essentially unchanged in the CSP. They easily apply to collaborators as well as individuals. However, as discussed in Section 3.3, their position in the evolutionary learning approach has been adjusted.

3.2.3.1 CSP Level 2.0

Often product size and resource estimates are developed via guesses and/or gut feels. However, using the methods of Level 2.0, one can systematically answer the question software development managers perpetually ask, similar to, "Can you be done with this project by the end of March? The customer wants it by March." It raises the engineer's ability to answer this question from a "probably" answer to an answer such as "I can tell you with 90% confidence that this project will be between 4,000 and 4,500 lines of code. Based on my own personal historical data, this should take me about 100 hours – so I feel very comfortable with a March commitment." In short, the method helps engineers make commitments they can meet.

The first step in formulating the commitment is the development of a high-level conceptual design. "This design establishes a preliminary design approach and names the expected product objects and their functions [11]." The idea is not to spend too

much time on the conceptual design – balancing the need to postulate the objects that will be needed without devising the high-level design. It is this conceptual design that will be used to determine the resource estimate/commitment required to build the product. Decisions on whether to proceed or not with the product will be based on these estimates and commitments. Further analysis and design activities will then refine this design if it is decided that the product will be developed.

The PROBE method is used to systematically develop a product size (lines of code) and resource estimate using sound mathematical methods. PROBE stands for PROxy-Based Estimating. First, the PROBE method recognizes the need to start the estimation process with "some proxy that relates product size to the functions the estimator can *visualize* and *describe* [11]." For object-oriented design, the objects identified in the conceptual design are used as the proxy. During early, conceptual design, the engineers can begin to *visualize* the objects that will be included in their design. An estimate of the number of lines of code per object estimate is made via projecting the quantity of methods each class will likely need and personal historical data on object size. This is a typical engineering estimation method in which the sum of estimates of the components is found to be more accurate than a single estimate of the whole.

Statistical linear regression analysis is then performed on the engineer's historical database of past projects to determine the relationships between past estimates, actual size, and actual effort. These relationships and the estimated object size (discussed above) are used to forecast the projected actual size and resource requirements for the current project. Finally, a prediction interval is calculated to give the likely range

around the estimate based on the variance in the historical data. Engineers can choose their external commitment from the range of values in the calculated prediction interval.

A few additional relevant measurements are added to the Project Plan in Level 2.0. First, the LOC/Hour measurement for the current program and all programs to date is added to the summary section. Additionally, the Cost-Performance Index (CPI) is added to indicate the degree to which cost commitments are being met. CPI is the ratio of the planned time to date divided by the actual time to date for all programs. A CPI of 1.0 or greater is desirable. A CPI less than 1.0 indicates that cost commitments are not being met.

As with the measurements of Level 1.1, automation of the PROBE method and the calculation of the additional measurements are essential for making the method practical and accessible to busy software engineers.

3.2.3.2 CSP Level 2.1

Performing the prescribed activities of Level 2.1 gives engineers an orderly plan for performing the required tasks to successfully complete a project and a framework for determining and communicating the status of their work. The engineer implements task and schedule planning and tracking via the earned value method.

> *A particular task's earned value is based on the percentage of the total planned project effort that the task will take. As tasks are completed, the task's planned value becomes earned value for the project. The project's earned value then becomes an indicator of the percentage of completed work. When tracked week by week, the project's earned value can be compared to its planned value to determine status, to estimate rate of progress, and to project the completion date for the project. [21]*

For the Task Planning, the engineer enumerates a list of the tasks needed to complete the project. The engineer then assigns a projection of the amount of time it would

take to complete the task (generally a percentage of the total resource estimate developed in Level 2.0). Each task is assigned an earned value based on the percentage of total time the task is projected to take. The engineer earned this value by completing the task. The engineer can easily communicate their completion status based on the Task Planning results.

The Schedule Planning is used to determine the status of how much time the have spent on the project. The resource estimate from Level 2.0 is divided into time commitments for each week of the project. The time entered beginning in Level 0.0 can be used to compare the time commitment of the Schedule Planning with actual time dedicated to the project.

Note: In the CSP, the PROBE instructions, Size Estimating Template, Task Planning Template, and Schedule Planning Template are identical to that of the PSP. Therefore, these are not included in Appendix B. For these instructions and templates, refer to Appendix C in [11].

3.3 Differences Between CSP and PSP

Obviously the largest difference between PSP and CSP is the incorporation of pair programming. Essentially every script, template, and form has been adjusted to incorporate the work of two and to specifically leverage the power of two working together.

Table 1 below summarizes the differences of the software engineering techniques introduced in each level. Two major differences are noted. First, the Quality Management and Project Management phases are swapped and reordered in the CSP. This was done to place additional focus on quality management early in the process, while accu-

mulating more historical data that can be used for estimation in CSP Level 2. Additionally, cyclic development is encouraged through levels 1 and 2 in the CSP making the equivalent of PSP Level 3 unnecessary.

There are really two levels of cyclic development involved. CSP encourages “micro-iteration” whereby a particular pair of programmers iterates while developing the segment of the project assigned to them. They analyze and develop and review a high-level design for their piece of the program. Then, the engineers are encouraged to divide up their design into their own micro-increments and cycle through low-level design, code, review, compile, and test for each. Another level of cyclic development is “macro-iteration” whereby a whole development team schedules the development of the whole project in large increments. This macro-iteration is not addressed by the CSP. It would need to be addressed by a larger team process, such as the Team Software Process [34].

Table 1: Differences between PSP and CSP Levels

Level	**PSP**	**CSP**
0.0	Baseline / Current Process	Baseline / Current Process
0.1	Coding Standard Size Measurement Process Improvement Proposal	Coding Standard Size Measurement Process Improvement Proposal
1.0	Size Estimating Test Reports	Analysis (Use Cases) CRC Cards Design
1.1	Task Planning Schedule Planning	Code Review Design Reviews Test Case Development Measurements
2.0	Code Review Design Review Measurements I	Size Estimating Resource Estimating
2.1	Design Templates Measurements II	Task Planning Schedule Planning
3.0	Cyclic Development	(Removed)

Additionally, more recent Object-Oriented Analysis and Design techniques were incorporated into the CSP. Use Cases, CRC Cards and class design are introduced in Level 1.0. Analysis was addressed in the PSP in the development of an Operational Scenario Template in which the system's operational behavior was described via scenarios. Use cases are a more recent, thorough, and higher-level version of the Operational Scenario Template. PSP design involved the development of the Functional Specification Template, State Specification Template and the Logic Specification Template. These all involve formal and semi-formal notation. While powerful techniques, practic-

ing programmers generally do not use formal notation, unless mandated by management. These were replaced with higher level UML class diagrams developed with the help of CRC card brainstorming.

Inspired by the automated testing techniques of XP, additional testing focus was incorporated into Level 1.1. Black box test cases are written during the design phase using a Test Case Template. White box and additional black box test cases are written prior to actually coding new functions, and these test cases are added to an automated regression test suite. Overall test coverage is checked against a Test Coverage Checklist.

"Lesson 1 about data collection is you may have to sacrifice some data accuracy to make data collection easier [31]." Many engineers complain about the amount of data they must record as part of PSP. Several fields were removed from the Defect Recording Log to reduce the tedium of entering defect data. Additionally, engineers are not asked to estimate defects prior to code development. Several of the forms were simplified.

CHAPTER 4

QUALITATIVE RESULTS

A formal experiment was run at the University of Utah to validate the effectiveness of the Collaborative Software Process. In the summer of 1999, a web programming class was taught to 20 undergraduates. The students formed ten pairs and worked collaboratively using the CSP for all assignments. The purpose of the class was to pilot the CSP before running a formal experiment.

The official experiment was run in the fall of 1999. The class consisted of 41 juniors and seniors. (It is important to note that by the time these students participated in this class and this experiment, they had significant programming experience in the form of internships and large class projects – such as writing compilers, portions of operating systems, and interpreters.) They learned both the PSP and the CSP and coded in C++, a language they had used for between two and three years. One third of the class worked individually while the rest worked in collaborative pairs. The individuals used the PSP; the pairs used the CSP. Both groups were asked to write the same programs so their results could be directly compared. The students completed six assignments over a period of seven weeks. The first and last assignments were pre-test and post-test elements of the formal experiment in order to study the performance of an individual programmer versus the performance of the same individual as a collaborative programmer. The experimental design and more details about these classes can be found in Appendix A.

The qualitative results discussed in this chapter were obtained through observation, personal experiences, discussions, and from written information obtained from the students involved in the experiment outlined in Appendix A and from professional pair programmers. The results center on explaining why collaborative programming is beneficial (beyond the economic advantages, which will be discussed in Chapter 5) and on sharing success factors for effective collaboration. (Much of this information was reported in [35-37].)

4.1 Why Collaborative Programming is Beneficial

4.1.1 Pair-Pressure

Pair programmers put a positive form of "pair-pressure" on each other. The programmers admit to working harder and smarter on programs because they do not want to let their partner down. Also, when they meet with their partner they both work very intensively because they are highly motivated to complete the task at hand during the session. "Two people working together in a pair treat their shared time as more valuable. They tend to cut phone calls short; they don't check e-mail messages or favorite Web pages; they don't waste each others time. [6]" (Contrast that with the productivity and quality expected from one student who admitted, "When I work on assignments individually, I can watch TV while I work.") Summarized by a pair programmer, "It takes more effort because the pace is forced by the other person all the time; neither person feels they can slack off." As each keeps his or her partner focused and on-task, tremendous productivity gains and quality improvements are realized.

As reported in [36], a class was taught at the University of Utah in which all students programmed collaboratively. These students consistently turned in their assignments on time; each of the ten collaborative groups turned in eight projects and all 80 were on time. Additionally, all projects were of very high quality. The average grade on all 80 assignments was 98%. (A teaching assistant, who had no interest in the research results, did all the project grading.)

This same group of students did not perform so flawlessly on their individual work. The students had one in-class midterm exam, one take-home final exam, and one paper evaluating the collaborative process. The average on these items was 78.1% with a standard deviation of 20.91. Again, the average on the collaborative aspects of the class was 97.9% with a standard deviation of 6.74. The students performed much more consistently and with higher quality in pairs than they did individually – even the less motivated students performed well on the programming projects. Through the students' weekly journal entries, the students communicated that this performance was not due to one person carrying the load of two – except on one of the 80 assignments. In an anonymous survey on the last day of class, the students were queried about the reasons for the performance differences of the projects vs. the exams. Overwhelmingly, the students responded, "It was the pair-pressure – I could not let my partner down."

Another benefit of pair pressure is improved adherence to procedures and standards. Each partner is expecting the other to follow the prescribed development practices. "With your partner watching, though, chances are that even if you feel like blowing off one of these practices, your partner won't . . . the chances of ignoring your commitment

to the rest of the team is much smaller in pairs then it is when you are working alone [5]"

4.1.2 Pair-Think

As reported in Chapter 2, Nick Flor, a Cognitive Science MS student, studied a pair of collaborative programmers. Flor recorded via video and audiotape their exchanges and progress on their task. A subset of these exchanges is discussed in [17] in order to correlate specific behaviors with known distributed cognition theories. One of these theories is "Searching Through Larger Spaces of Alternatives" demonstrates *pair-think*.

> *A system with multiple actors possesses greater potential for the generation of more diverse plans for at least three reasons: (1) the actors bring different prior experiences to the task; (2) they may have different access to task relevant information; (3) they stand in different relationships to the problem by virtue of their functional roles. . . An important consequence of the attempt to share goals and plans is that when they are in conflict, the programmers must overtly negotiate a shared course of action. In doing so, they explore a larger number of alternatives than a single programmer alone might do. This reduces the chances of selecting a bad plan.* [17]

A student pair-programmer confirms Flor's findings, "We often came up with different ideas about how the design should go and the result of arguing over which one was better often led to a truly superior hybrid design."

Flor also reports,

> *Because of the nature of ill-structured tasks, there is often insufficient information for selecting the right plan. Thus there is the potential for an incorrect or less efficient course of action to be adopted. Fortunately, refuted plans do not disappear. The process of negotiating plans distributes them between the actors. If at a later time, it is discovered that the current course of action is wrong, that plan may later be independently adopted by an actor who is not necessarily its originator.* [17]

4.1.3 Pair-Relaying

Literature on collaboration overflows with examples of remarkable achievements in many fields that could have only occurred with collaboration. One author contends we are living in a world in which technological complexity "increases at an accelerating rate [which] offers fewer and fewer arenas in which individual action suffices. [38]." Software has become the driving force behind most new technologies. But the engineering of software is becoming increasingly complicated. A software engineer must balance a variety of competing factors, including functionality, quality, performance, safety, usability, time to market, and cost. Moreover, the size of software systems that are being built is rapidly growing.

Related to pair-think, collaborative teams consistently report that together they can evolve solutions to unruly or seemingly impossible problems. "Problem solving" refers to when the two programmers are puzzled as to why something doesn't work as expected, or simply can't figure out how to go forward. *Pair relaying is* a name for the effect of having two people working to resolve a problem together in the exact manner Wagstaff describes below.

> *There were times we felt that we would have given up except that we "tag teamed." I'd be on the ropes and I'd describe the problem in such a way that he had a valuable insight. Then he'd fight on as long as he could and stop . . . then I'd have an insight . . . and so on. I suppose others would call it brainstorming, but it feels different to me.*
>
> [-David Wagstaff, software engineer, Salt Lake City]

Practitioners describe contributing their knowledge to the best of their abilities, in turn. They share their knowledge and energy (and also brainstorming) in turn, chipping steadily away at the problem, evolving a solution to the problem.

Additionally, pairs report that in their problem solving, they do not spend as much time lost in a particular problem or fix.

One student noted:

> *One problem with single programming is that you can forget what you are doing and easily get wrapped in a few lines of code, losing the big picture. Your partner is able to constantly review what you do, making sure that it is in line with the product design. He/she can also make sure that you are not making the problem too difficult. Many times, these two items alone can waste a lot of time.*

Combining pair-think and pair-relaying is powerful. One student wrote, "I have found that, after working with a partner, if I go back to working alone, it is like part of my mind is gone. I find myself getting confused about things."

4.1.4 Pair-Reviews

Inspections were introduced more than twenty years ago as a cost-effective means of detecting and removing defects from software. Results [29] from empirical studies consistently profess the effectiveness of reviews. Even still, most programmers do not find inspections enjoyable or satisfying. As a result, inspections are often not done if not mandated, and many inspections are held with unprepared inspectors.

> *Despite a consistent stream of positive findings over 20 years, industry adoption of inspection appears to remain quite low, although no definite data exists. For example, an informal USENET survey we conducted found that 80% of 90 respondents practiced inspection irregularly or not at all* [39].

The theory on why inspections are effective is based on the prominent knowledge that the earlier a defect is found in a product, the cheaper it is to fix the defect. Many sources, including [40] state that it is ten times more expensive to remove a defect for each additional process step.

Intuitively, it is easy to understand why this exponential cost increase occurs. In a review, the programmer looks directly at the problem that was just identified and considers alternatives and fixes. Once the product enters test or is delivered to customer(s), the programmer or field maintenance team must work to translate the symptom (e.g. the answer is wrong or the program crashed) back to the problem (which exact line(s) of code caused the symptom). It is easy to see why the translation of the symptom back to the problem would cost exponentially more than direct problem identification.

With pair programming, this problem identification occurs on a minute-by-minute basis. "The human eye has an almost infinite capacity for not seeing what it does not want to see . . . Programmers, if left to their own devices, will ignore the most glaring errors in their output – errors that anyone else can see in an instant [41]." With pair-programming, "four eyeballs are better than two," and a momentous number of defects are prevented, removed right from the start. These continual reviews outperform traditional, formal reviews in their defect removal speed. Additionally, they also eliminate the programmer's distaste for reviews so that effective reviews are *actually* performed.

4.1.5 Debugging by Explaining

Every person has experienced in some context that some problems can be resolved by explaining them to another.

> *. . . effective technique is to explain your code to someone else. This will often cause you to explain the bug to yourself. Sometimes it takes no more than a few sentences, followed by an embarassed "Never mind; I see what's wrong. Sorry to bother you." This works remarkably well; you can even use non-programmers as listeners. One university computer center kept a teddy bear near the help desk. Students with*

> *mysterious bugs were required to explain them to the teddy bear before they could speak to a human counselor. [42]*

Students at the University of Utah, similarly, noted how surprised they were that it helped them to understand things when they had to explain it to another. As one student said,

> *When I explained an idea to my partner, I concentrated on what I was saying, and carefully made things clear and logical because I did not want to confuse my partner and I wanted him to understand what I was talking about. It helped me better understand the problem I was addressing. It also helped me discover some mistakes I had made but did not notice before I talked with my partner.*

4.1.6 Pair-Learning

The continuous reviews of collaborative programming create a unique educational capability, whereby the pairs are endlessly learning from each other. "The process of analyzing and critiquing software artifacts produced by others is a potent method for learning about languages, application domains, and so forth [39]." Earlier, it was stated that the continuous reviews of collaborative programming were more effective than traditional review because of their optimum defect removal efficiency. To further this, the learning that transcends in these continual reviews prevents future defects from ever occurring – and defect prevention is more efficient than any form of defect removal. Says Capers Jones, chairman of Software Productivity Research,

> *It is an interesting fact that formal design and code inspections, which are currently the most effective defect removal technique, also have a major role in defect prevention. Programmers and designers who participate in reviews and inspections tend to avoid making the mistakes which were noted during the inspection sessions.* [43]

Phillip M. Johnson, a professor at the University of Hawaii, refutes traditional inspections heuristic "Raise issues, don't resolve them." He speaks, instead, in favor of

the educational opportunity that abounds in code inspections. "A strong argument can be made that overall software quality is affected far more profoundly by improvements to developer skills, which reduces future defect creation, than by simply removing defects from current individual documents [39]." The continuous reviews of collaborative programming, in which both partners ceaselessly work to identify and resolve problems, affords both optimum defect removal efficiency **and** the development of defect prevention skills.

4.1.6.1 Pair-Learning in the Classroom

Larry Constantine, whose observation of P. J. Plaugher's software company were reported in Chapter 2, noted that ". . . for language learning, there seems to be an optimum number of students per terminal. It's not one . . . one student working alone generally learns the language significantly more slowly than when paired up with a partner [19]." A class taught during Summer Semester at the University of Utah set out to study pair programming in an educational setting in which programming language learning takes place. (Details on this class can be found in Appendix A.) The results of the class, in which collaborative programming proved beneficial to both students and teaching staff, will be discussed.

Despite the fact that some students get better grades than others, classrooms are unique in that skill level and experience between students are relatively equivalent when compared with differences found in industry. As a result, the students have a mutual learning relationship rather than a novice-expert relationship. This relationship proved fruitful for the students. The students learned several web programming languages dur-

ing the (shorter) summer semester. However, the students were able to easily tackle all programming projects, which was very satisfying for them. When one partner did not know/understand something, the other almost always did and could provide immediate assistance. In a survey at the end of the semester, 74% of the students noted, “Between my partner and I, we could figure everything out.”

They also found it a very efficient working arrangement. When they found themselves needing to use new or unfamiliar semantics or syntax, the non-driver has the job of flipping through resource materials. During this time, the driver might make progress on a more familiar area of the code. Together, defect removal was also much more efficient, which significantly reduced the frustration level of debugging they had been accustomed to. Most significantly, 84% of the class agreed with the statement “I learned faster and better because I was always working with a partner.”

Collaboration also makes the instructor feel more positive about the class. Their students are happier, and the assignments are handed in on-time and are of higher quality. There is one additional very positive effect for the teaching staff -- less questions! When one partner did not know/understand something, the other almost always did. Between the two of them, they could tackle anything, which made them much less reliant on the teaching staff. Email questions were almost non-existent. Lab consultation hours were very calm, even the day the projects were due.

Naturally, though, pair programming requires the teaching staff to deal with obvious workload imbalances between the partners that they would not have to deal with if each worked individually. Normal two-person team projects are divided into “my” part and “your” part. However, with collaborative programming, the entire project is “ours.”

Because of this, there was much more of a “collective code ownership” feeling and far less partner problems than have been observed in other classes in which students worked in traditional two-person teams.

(Note: much of the information contained in this Pair-Learning in the Classroom section had been previously reported in [36].)

4.1.6.2 Pair-Learning in the Workplace

In the workplace, skills levels are more variable than in a classroom. The majority of this section will address the knowledge transferability experienced with collaborative programming. This is not to say that the expert does not benefit from the arrangement, even when paired with a novice. Consider this experience of a senior programmer:

> *I was sitting with one of the least-experienced developers, working on some fairly straightforward task. Frankly, I was thinking to myself that with my great skill in Smalltalk, I would soon be teaching this young programmer how it’s really done.*
>
> *We hadn’t been programming more than a few minutes when the youngster asked me why I was doing what I was doing. Sure enough, I was off on a bad track. I went another way. Then the whippersnapper reminded me of the correct method name for whatever I was mistyping at the time. Pretty soon, he was suggesting what I should do next, meanwhile calling out my every formatting error and syntax mistake.*
>
> *I’m not entirely stupid. I noticed very quickly that this most junior of programmers was actually helping me! Me! Can you believe it? Me! That has been my experience every time thereafter, in pair-programming. Having a partner makes me a better programmer.*
>
> [-Ron Jeffries, from [35]]

Learning happens in a very tight apprenticeship mode. From moment to moment, the partners can take turns being the teacher and the taught, the novice and the expert. Even unspoken skills and habits cross partners. [44] discusses apprenticeship case stud-

ies. These studies range from tailors to flag signalmen in the U.S. Navy to butchers in modern supermarkets.

The book points out the importance of the novice working in "line of sight" of the expert. Expertise is transmitted, in part, through the ongoing visual (and auditory) field. They describe successful apprenticeship learning in both tailors and Navy signalmen where “line of sight” is available. The beginner explicitly picks up skills from hearing and/or seeing the expert.

Apprentice butchers, however, do not have line of sight access to their local expert. The beginners are given simple cuts to perform, but do not have a way to learn how to do more difficult cuts, which were being done by the senior butcher in another room. The authors present this as a situation in which apprenticeship learning does not effectively happen.

Most project programming environments match the butcher situation, not the tailor or signalmen situation. The novice programmer generally sits in their workspace working on simple code; the expert sits in their own workspace creating complex code and making architectural decisions. Pair programming is a far superior apprenticeship model (though it has already been stated that the expert, too, learns from the novice.)

In industry, pairs are not usually assigned to each other on a long-term basis. Often, pairs change day-by-day, giving additional opportunity for learning. [45] contains a section called “I heard it through the Pairvine” which discusses this phenomenon – when one person learns a new trick with a tool, or a new innovation or snafu, it tends to spread through the whole group within a couple of days with no deliberate effort.

> *When an important new bit of information is learned by someone on the team, it is like putting a drop of dye in the water. Because of the pairs switching around all the time, the information rapidly diffuses throughout the team just as the dye spreads throughout the pool. Unlike the dye, however, the information becomes richer and more intense as it spreads and is enriched by the experience and insight of everyone on the team.* [5]

4.1.7 Team Building

Programming teams in industry in which pair programming was practices report significantly improved teamwork among the members. If the pair can work together, then they learn ways to communicate more easily and they communicate more often. In many cases, these industrial teams continually rotate partners; two people do not work together for more than a short increment. This increases the overall information flow and team jelling farther.

Anecdotally, pair-programming also tends to improve the team's hustle, as described in [46]:

> *A baseball manager recognizes a nonphysical talent, hustle, as an essential gift of great players and great teams. It is the characteristic of running faster than necessary, moving sooner than necessary, trying harder than necessary. It is essential for great programming teams too. Hustle provides the cushion, the reserve capacity, that enables a team to cope with routine mishaps, to anticipate and forefend minor calamities.*

4.1.8 Project Risk

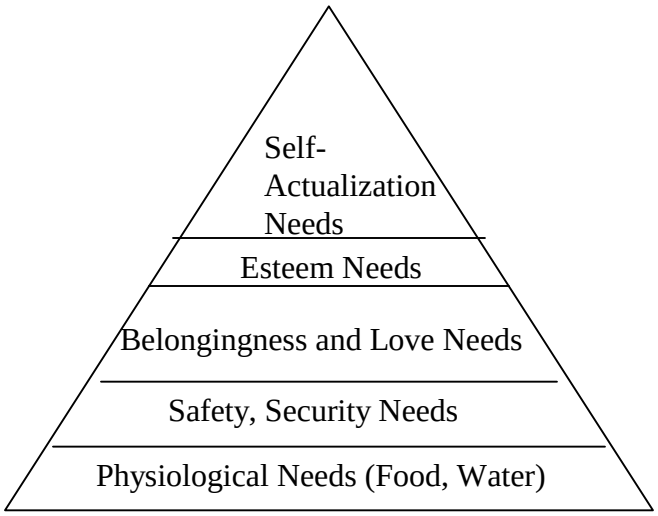
With pair programming, the risk from losing key programmers is reduced, because there are multiple people familiar with each part of the system. If a pair works together consistently, then there are two familiar with this particular area of the program. If the pairs rotate, as discussed above, many people can be familiar with each part. A common informal metric (invented by Jim Coplien of AT&T Bell Labs) is referred to as the

"truck number." "How many or few people would have to be hit by a truck (or quit) before the project is incapacitated?" The worst answer is "one." Having knowledge dispersed across the team increases the truck number, and project safety.

4.1.9 Maslow's Needs Hierarchy

In the early 1950's Abraham Harold Maslow postulated that people will work to satisfy their own needs, but in a hierarchical order of importance. Each lower level must be satisfied at least partially before the person will be motivated to satisfy a higher-level need [47]. Maslow's hierarchy underlies many management approaches to quality motivation [10]. The hierarchy is defined below in Figure 3.

Figure 3: Maslow's Hierarchy of Needs



Contrast individual and collaborative programming and Maslow's assertion that people will work to satisfy their own needs, but in a hierarchical order of importance. If both provide stable employment, they equally satisfy the basic human Physiological, Safety and Security needs at the lowest two levels. However, the social interaction of collaborative programming is attractive to engineers because it better serves the next two higher levels of basic human needs. The Belonging and Love needs causes people to "hunger for relations with people in general – for a place in the group or the family [47]." Additionally, essentially all engineers earn the respect of their partners. Indeed, it has been demonstrated that even novice programmers are able to help expert programmers, giving the thrill of contribution and confidence. This respect helps satisfy the Esteem need, which professes the "need or desire . . . for the esteem of others. The most stable and therefore most healthy self-esteem is based on *deserved* respect from others [47]." Only when the lower four levels are satisfied to a great degree is the fifth and highest level, Self-Actualization, more satisfying. This highest level is the desire for an "individual to do what he or she, individually, is fitted for. Musicians must make music, artists must paint . . ."

Matthias Felleisen of Rice University refers to solo programmers as "lonely macho warriors battling against a sea of bits and bytes." This description reflects that solo programming may satisfy an engineers self-actualization needs, but neglect their more basic needs for belonging and the respect of their partner and teammate.

(Note: much of the information contained in the Pair-Learning in the Workplace, Team Building, and Project Risk sections had been previously reported in [Cockburn, submitted for consideration #58].)

4.2 Success Factors for Effective Collaboration

Most programmers are long conditioned to working alone and often initially resist the transition to pair programming. Ultimately, most make this transition with great success. This purpose of this section is to document proven strategies for aiding programmers in becoming effective pair programmers. (Much of the information in this section has previously been reported in [48].)

4.2.1 Pair-Jelling

The pair must cease considering themselves as a two-programmer team and must start considering themselves as one coherent, intelligent organism working with one mind. Tom DeMarco shares his inspiring view on this type of union.

> *A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts. The production of such a team is greater than that of the same people working in unjelled form. Just as important, the enjoyment that people derive from their work is greater than what you'd expect given the nature of the work itself. In some cases, jelled teams working on assignments that others would declare downright dull have a simply marvelous time. … Once a team begins to jell, the probability of success goes up dramatically. The team can become almost unstoppable, a juggernaut for success* [49].

4.2.2 Project Ownership

In pair programming, *two* programmers are assigned to jointly produce *one* artifact (design, algorithm, code, etc.). The two programmers are jointly responsible for every aspect of this artifact. One person is typing or writing, the other is continually reviewing the work. But, both are equal participants in the process. Both partners own everything.

With pair programming, the two programmers become one. There should be no competition between the two; both must work for a singular purpose, as if the artifact was produced by a singular good mind. Blame for problems or defects should never be placed on either partner. The pair needs to trust each other's judgment and each other's loyalty to the team.

4.2.3 Mutual and Self-Respect

Pair programmers indicate that it is very difficult to work with someone who has a great insecurity or anxiety about their programming skills. They tend to be defensive or do not contribute to the team. Programmers with such insecurity should view pair programming as a means to improve their skill by constantly watching and obtaining feedback from another.

Also, negative thoughts such as "I'm an awesome programmer, and I'm paired up with a total loser" should also be rejected, lest the collaborative relationship be destroyed. None of us, no matter how skilled, is infallible and above the input of another. John von Neumann, the great mathematician and creator of the von Neumann computer architecture, recognized his own inadequacies and continuously asked others to review his work. "And indeed, there can be no doubt of von Neumann's genius. His very ability to realize his human limitation put him head and shoulders above the average programmer today . . . Average people can be trained to accept their humanity -- their inability to function like a machine -- and to value it and work with others so as to keep it under the kind of control needed if programming is to be successful [41]."

4.2.4 Ego-Less Programming

"Ego-less programming," an idea surfaced by Gerald Weinberg in *The Psychology of Computer Programming* [41] a quarter of a century ago, is essential for effective pair programming. According to the pair programming survey (see Appendix D), excess ego can manifest itself in two ways, both damaging the collaborative relationship. First, having a "my way or the highway" attitude can prevent the programmer from considering others ideas. Secondly, excess ego can cause a programmer to be defensive when receiving criticism or to view this criticism as mistrust.

In *The Psychology of Computer Programming [41]*, a true scenario about a programmer seeking review of the code he produced is discussed. On this particular "bad programming" day, this individual ego-lessly laughed because his reviewer found seventeen bugs in thirteen statements. However, after fixing these defects, this code performed flawlessly during test and in production. How different this outcome might have been had this programmer been too proud to accept the input of others or had viewed this input as an indication of his inadequacies. Having another to continuously and objectively review design and coding is a very beneficial aspect of pair programming. "The human eye has an almost infinite capacity for not seeing what it does not want to see . . . Programmers, if left to their own devices, will ignore the most glaring errors in their output -- errors that anyone else can see in an instant [41]."

Conversely, a person who always agrees with their partner lest he or she create tension also minimizes the benefits of collaborative work. For favorable idea exchange, there should be some healthy disagreement/debate. Notably, there is a fine balance between displaying too much and too little ego. Effective pair programmers hone this

balance during an initial adjustment period. Ward Cunningham, one of the XP founders and experienced pair-programmer, reports that this initial adjustment period can take hours or days, depending on the individuals, nature of work and their past experience with pair-programming.

4.2.5 Workspace Layout

In the pair programming survey (see Appendix D), 96% of the programmers agreed that appropriate workspace layout was critical to their success. The programmers must be able to sit side-by-side and program, simultaneously viewing the computer screen and sharing the keyboard and mouse. In

Figure 4 below (from [50]), layouts to the right are preferable to layouts on the left. Effective communication, both within a collaborative pair and within and between collaborative pairs, is paramount. Without much effort, programmers need to see each other, ask each other questions and make decisions on things such as integration issues, lest these questions/issues are not discussed adequately. Programmers also benefit from "accidentally" overhearing other conversations to which they can have vital contributions. Separate offices and cubicles can inhibit this necessary exchange. "If any one thing proves that psychological research has been ignored by working managers, it's the continuing use of half partitions to divide workspace into cubicles. … Like many kings, some managers use divide-and-conquer tactics to rule their subjects, but programmers need contact with other programmers. [41]"

Figure 4: Workspace Layout

4.2.6 Taking Breaks

Because pair programmers do keep each other continuously focused and on-task, it can be very intense and mentally exhausting. Periodically, taking a break is important for maintaining the stamina for another round of productive pair programming. During the break, it is best to disconnect from the task at hand and approach it with freshness when restarting.

CHAPTER 5

QUANTITATIVE RESULTS

This chapter will explore the economics of collaborative programming based on the observations and measurements of the students involved in the experiment at the University of Utah outlined in the previous chapter and in Appendix A. Specifically, comparisons were made between individuals using the PSP and collaborators using the CSP because it has been shown [12, 21] that PSP is a significant improvement over the ad hoc development prevalent in industry. Transitively, it follows that, if CSP is an improvement to PSP, it is an even greater improvement to ad hoc development practices.

Differences between the individual (control) group and the collaborative (experimental) group were examined for statistical significance using the independent-samples t test. This test is used to examine if the mean of a single-variable for subjects in one group differs from that in another group. An independent samples test can be used because the students were placed in identical situations. The only difference between the groups was the variable under study, collaborative work vs. individual work.

In the independent-samples t test, a “p-value” indicates the probability of the difference result being caused by chance. Differences, which had a p-value of less than .05, were deemed statistically significant, indicating that there would be less than a 5% chance the difference would be caused by chance.

The chapter will discuss the effect of collaborative programming on the satisfaction of the programmers. Lastly, some additional, secondary quantitative measurements will be discussed.

5.1 An Economic Evaluation of the Collaborative Software Process

> *. . . economics is primarily a science of choices, and software economics should provide methods and models for analyzing the choices that software projects must make*. [51]

The affordability of pair programming is a key issue. If it is more expensive, managers simply will not permit it. “From a business standpoint, profit is not only an organization's goal, it is necessary for its survival. The ultimate aim of engineering is to create the most income from the least expense, thus maximizing profit. [52].”

Skeptics assume that incorporating pair programming will double code development expenses and critical manpower needs. Along with code development costs, however, other expenses, such as quality assurance and field support costs must also be considered. IBM reported spending about $250 million repairing and reinstalling fixes to 30,000 customer-reported problems [11]. That is over $8,000 for each defect!

In this section, first some basic measurements and observations of the students in the experiment will be reported. Delving into emerging research in Software Engineering Economics, an affordability model for pair programming will be developed using these observed measurements. “Practitioners are most concerned about understanding what aspects of software engineering innovations have worked best and whether they are applicable to their particular situation [53].”

5.1.1 Pair-Quality

> *We have learned to live in a world of mistakes and defective products as if they were necessary to life. It is time we adapt a new philosophy in America.* [W. Edwards Deming]

Product quality is a very important metric. The collaborative pairs and the individual students in the University of Utah experiment completed four programs to the same specifications. The results were compared. The bar chart in Figure 5 below shows the percentage of the instructor's test cases passed, on average, by the two groups. On average, the collaborators' code had about 15% fewer defects than the individuals' code. These results are statistically significant at $p < 0.05$ in all cases except the first program.

Figure 5: Post Development Test Cases Passed

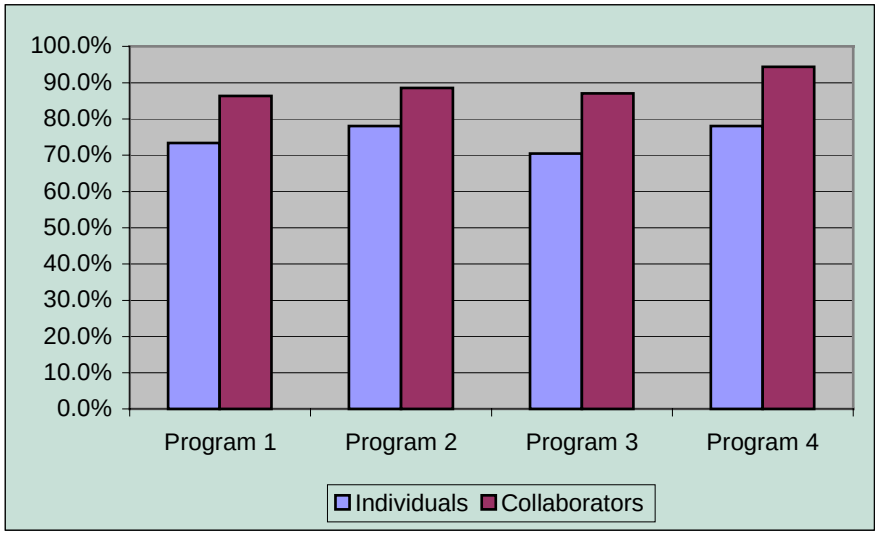
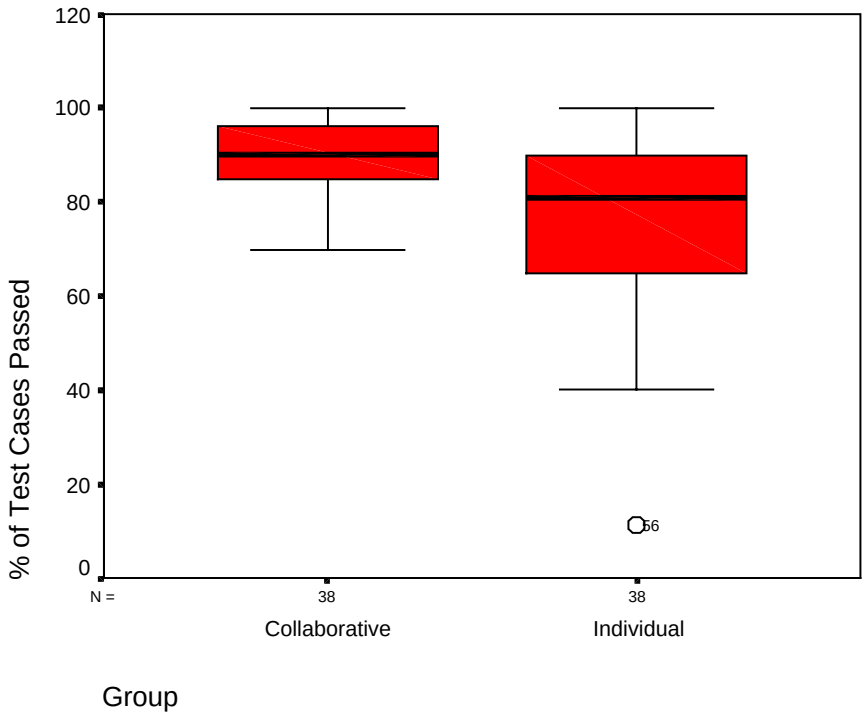


Figure 6 displays a boxplot of the percentage of post development test cases passed for programs 2 through 4. It demonstrates graphically why the quality differences are statistically significant. The horizontal line at the center of each box marks the median (the middle observation when the data values are ordered from smallest to largest) of each sample. The edges of the box mark the 25th and 75th percentiles. The "whiskers," or the vertical lines that extend from the box show the range of values that fall within 1.5 box lengths of the median. The open circle (o) under the individual box indicates an extreme value that is more than 3 box lengths from the median. The number under each box is the sample size.

Figure 6: Pair-Quality Boxplot

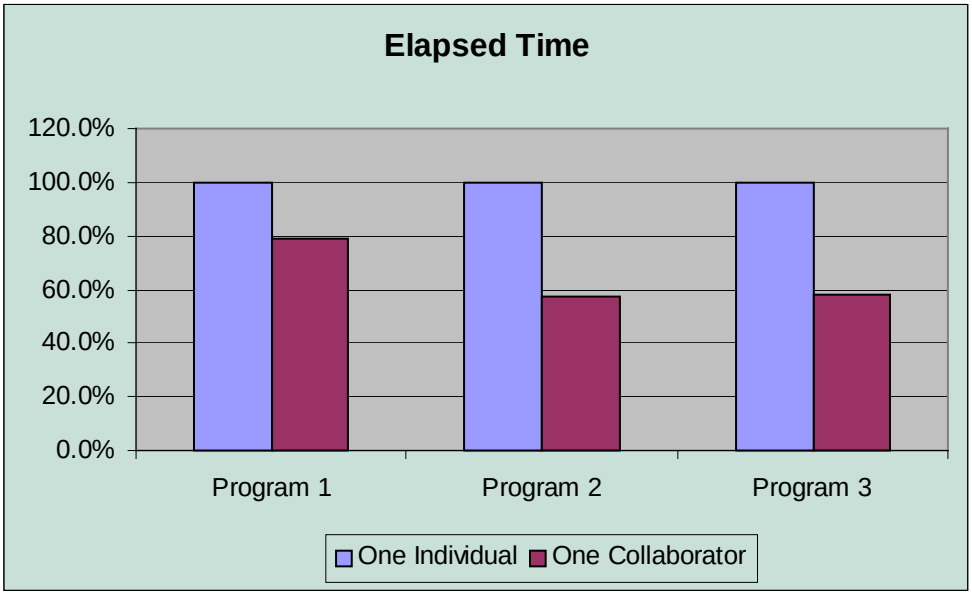


The boxplot in Figure 6 shows several desirable qualities for the quality of collaborative pairs. First, the median value is clearly higher than the median for individuals. In fact, the lower edge of the collaborator box which marks the 25th percentile is above the individual's median – so over 75% of the collaborators achieved scores higher than the individuals. Additionally, the range of values for the collaborators is significantly smaller, indicating greater consistency of high quality, as would be expected by the effects of pair-pressure.

5.1.2 Pair-Time

Many people's gut reaction is to reject the idea of pair-programming because they assume that there will be a 100% programmer-hour increase by putting two programmers on a job that one can do. After the initial adjustment period the total programmer hours spent on each assignment trended downward dramatically as shown below in Figure 7. Together the pairs only spent about 15% more time on the program than the individuals. Additionally, after the first program, the difference between the times for individuals and for the pairs was no longer statistically significant. (As discussed, statistical significance is obtained if $p < .05$. For the difference in time values, $p = .380$, which indicates that there is almost a 40% chance the difference in time values would be observed by chance.)

Figure 7: Elapsed Time



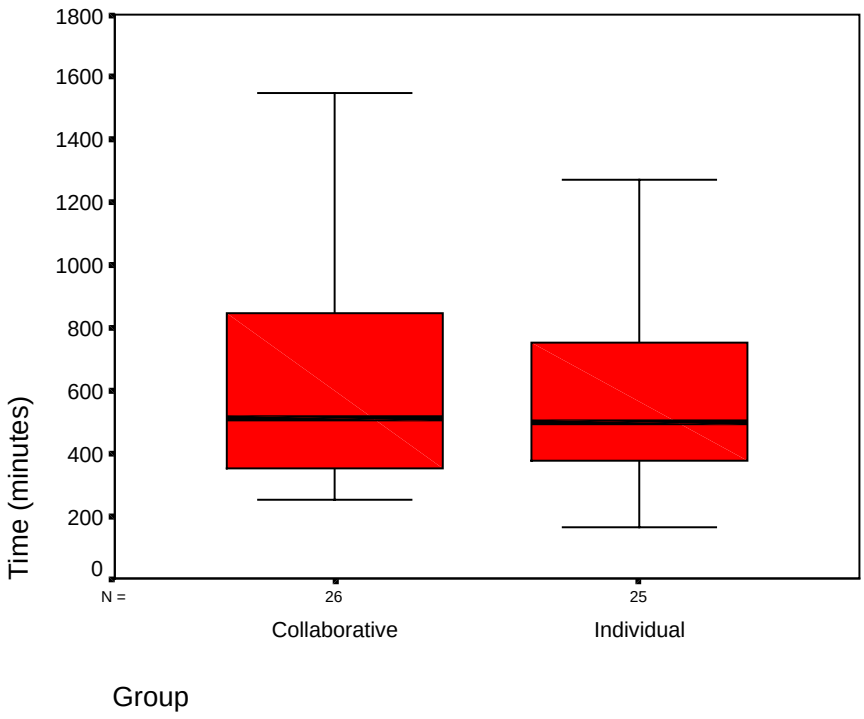
(One important note: Programmer time for program 4 was not included in this analysis. For this particular program, the students reported time measurements with significant variability. Therefore, the instructor distributed an anonymous survey to obtain feedback on this variability from the students. The students reported that it was mid-term week and many of them had a large Operating Systems project due during that week. Many of them said that they recorded their time and defect data for this program significantly less accurately than they had in the past. Therefore, these self-reported measurements were eliminated from the study. The students' quality level for this particular program was included in section 5.1.1 because quality was not a self-recorded measurement.)

Figure 8 below shows the box plot of the time values for programs 2 and 3, after the pair-jelling has occurred. It is easy to see why the time differences are no longer statistically significant. The median values are essentially equal. The range of values for the

collaborators is larger than for the individuals, which drives the 15% increase in the mean value for the collaborators.

These findings that resource requirements do not double with collaborative programming agree with the anecdotes of professional pair-programmers. Professionals who have paired for a year or more consistently describe pair-programming as “more than twice as fast,” implying that increased productivity gains might be realized with more experience. The economic analysis below will assume that pairing requires a 15% development time investment. This assumption, however, is clearly conservative considering the input of long-time pair programmers.

Figure 8: Pair-Time Boxplot



5.1.3 Net Present Value Analysis

"Software quality is an investment that should provide a financial return relative to the initial and ongoing expenditures in the software quality improvement initiatives [54]." [52, 54-56] profess the use of the economic *net present value* (NPV) to evaluate the return on software quality initiatives. NPV is the most widely accepted criterion for project evaluation in corporate finance [57]. A project with positive NPV increases the wealth of the firm. A high NPV is more preferable to a low NPV; a negative NPV indicates an unacceptable investment.

NPV is measured in today's dollars. The time value of money is accounted for by discounting all future cash flows back to the present time under the assumption that a dollar today is worth $(1 + d)^T$ dollars at time T in the future (or a dollar at time T is worth $1/(1 + d)^T$ today). The positive quantity d is referred to as the discount rate, which captures the opportunity cost (e.g. the minimum acceptable return for an investment that the company requires for similar projects) of the underlying investment [58].

A net present value model will first be explained. Then, the model will be used to evaluate the economic advantage of the CSP.

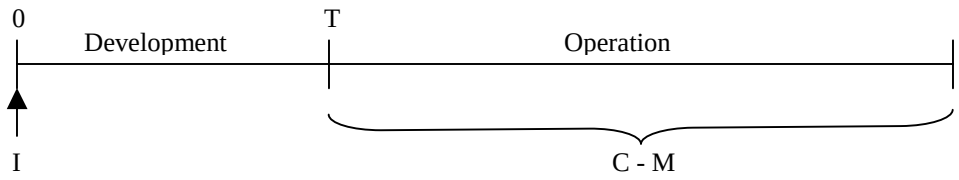
5.1.3.1 General Net Present Value Model

In one economic model [56], Erdogmus considers five determinants to compare the net present value of alternative software development strategies. Consider the timeline and these determinants explained in Figure 9.

The present value of the project is calculated by discounting the net asset value back to time zero (from time T) using a discount rate of d and subtracting the cost of development from the result.

This standard, recognized model will be slightly altered for the analysis of the affordability of CSP based on the personal suggestion of Hakan Erdogmus of the National Research Council of Canada. In order to avoid the choice of an arbitrary Asset Value (C) to represent projected revenues, a Present Value of Costs (PVC) model will be used instead. The calculation of PVC is similar to that of NPV, except that Asset Value (C) is not considered. PVC is then calculated $M/(1+d)^T + I$. The alternative with the lower PVC is superior.

Figure 9: Net Present Value



Development Time (T) or time to market. This is the elapsed time between the commit to invest in the project and the time of the first major positive cash flow from revenues or cost savings.

Development Cost (I) is the total present value at time 0 of all negative cash flows from the time the decision to invest is made to the time the first major positive cash flow.

(Future) Asset Value (C) is the total present value at time T of the positive cash flows that the project is expected to generate post development.

Operation Cost (M) is the total present value at time T of all negative cash flows of the operation phase, mainly maintenance costs.

Discount rate (d) captures the opportunity cost of the underlying investment.

Net Asset Value (C-M) = The net of ongoing post development cash flows.

Net Present Value (NPV) = $(C-M)/(1+d)^T - I$

5.1.3.2 Analyzing the Economic Advantage of CSP Using the Present Value of Costs (PVC) Model

A hypothetical situation will be developed in order to demonstrate the economic advantage of CSP. Consider an application projected to be 50,000 lines of code (LOC). Simultaneously, the application will be developed by an individual programmer and by a collaborative pair. The process of developing the application via the individual will

be considered the *base strategy*. The process of developing the application via the collaborative pair will be considered the *test strategy*.

General Assumptions:

1. An average productivity rate of 25 LOC/hour will be used for the analysis. This was the average productivity rate of 196 engineers who took PSP training [21].
2. The US average defect/KLOC rate is 39 defect/KLOC. This statistic was obtained from Capers Jones [43]. The data comes from companies such as AT&T, Hewlett Packard, IBM, Microsoft, Motorola, Raytheon, and similar companies with formal defect tracking and measurement capabilities.
3. In the US, on average 85% of defects are removed via the development process. 15% of all defects escape to the customer. This statistic was also obtained from Capers Jones [43]. (Together assumptions 2 and 3 indicate that there would be 5.85 defects/KLOC remaining in code. This is consistent, though on the low side, with statistics from the Pentagon and the SEI which state that typical software applications contain 5-15 defects per KLOC [59].)
4. Collaborative pairs spend 15% more time overall than individuals (see section 5.1.2). However, since this work is done in tandem, the collaborators spend 57.5% of the elapsed, “wall clock” time that individuals do.
5. Code produced by collaborative pairs has a 15% lower defect density than code produced by individuals (see section 5.1.1).

6. When software is delivered to customers or users, bugs and defects start being reported back to the development organization. However, the discovery of bugs by users is not instantaneous. Table 2 below enumerates the US average three year discovery rate of initial software defects after release found in [43]:
7. [30] reported statistics from large Bell Northern Research software projects (over 2.5 million lines of code) that show that each defect in software released to customers and subsequently reported as a problem requires an average of 4.5 man-days to repair or an average of 33 hours of subsequent maintenance effort, assuming a 7.5 hour workday. This is consistent with data reported in [11]. Therefore, 33 hours/field defect will be used in the analysis.
8. A reasonable annual discount rate of 10% is used for both the base and test strategy. This equates to 0.80% per month.
9. Software engineers that develop new code cost $50/hour. Field support software engineers cost $40/hour. (This includes salary + benefits.)

In Table 3, the Present Value of Costs is calculated for both the individual and the collaborative programming alternatives.

Table 2: US Average Defect Discovery Rate

	Average
Year 1	57.5%
Year 2	27.5%
Year 3	12.0%
Latent (Not found)	3.0%
Total	100.0%

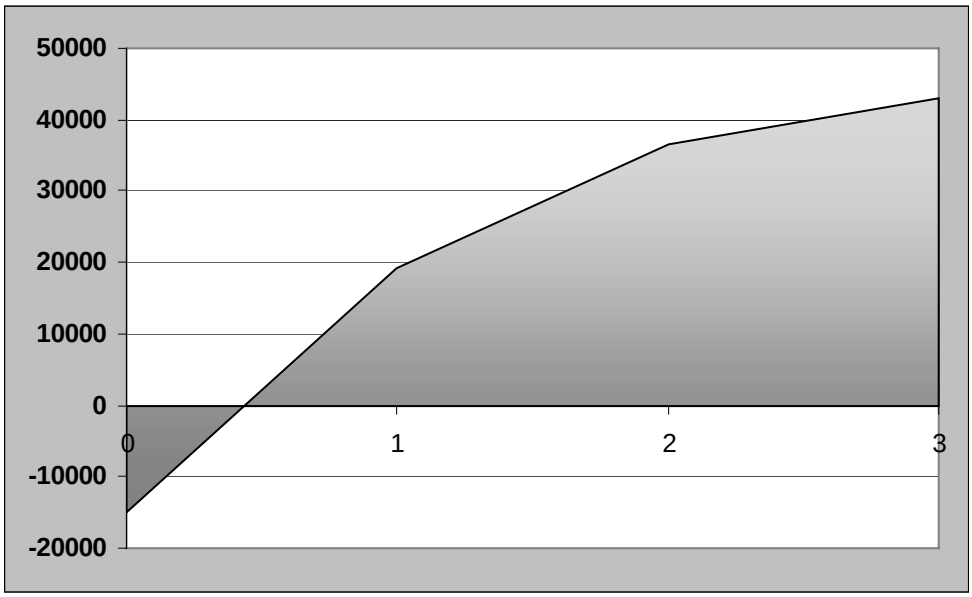
Table 3: Present Value of Costs (PVC) Analysis

	Assumption	**Individual**	**Collaborators**
Engineer Hours	1, 4	2,000 hours	2,300 hours
Development Time (T)	4	2,000 hours (~52 weeks or ~12 months)	1,150 hours (~30 weeks or ~7 months)
Development Cost (I)	9	$100,000	$115,000
Defect in Field (DF)	2, 3, 5, 6	293 Discovery by Year (post development) T + Year 1: 169 T + Year 2: 81 T + Year 3: 35	249 Discovery by Year (post development) T + Year 1: 143 T + Year 2: 68 T + Year 3: 30
Operation Cost(M) (sum of field costs for each defect, discounted back to time T)	DF + 7, 8	$(169*33*40)/1.10 + (81*33*40)/1.10^2 + (35*33*40)/1.10^3 =$ 325,874	$(143*33*40)/1.10 + (68*33*40)/1.10^2 + (30*33*40)/1.10^3 =$ 275,534
Discount Rate (d)	8	10% (or 0.8% monthly)	10% (or 0.8% monthly)
Present Value of Life-time Costs (PVC)		$325874/1.008^{12} + 100{,}000$ = $396,158	$275534/1.008^{7} + 115{,}000$ = $375,586
PV of Cost Savings with CSP relative to PSP ($PVC_{PSP} – PVC_{CSP}$)			$20,572

Because the PVC for the collaborators is less than that of the individuals, the firm's worth would gain more from a collaborative programming strategy. In fact, using the parameters in the example, pair programming could cost as much as 135% of development cost of individuals and the firm would still break even with higher quality. The cost savings of CSP through time can be viewed graphically in Figure 10. Initially, CSP costs more than PSP. Through time, savings are accrued to net a positive investment.

To paraphrase Crosby's words [1] in Chapter 1, "Pair-quality is free . . . Pair-quality is not only free, it is an honest-to-everything profit maker."

Figure 10: Cost Saving of CSP Through Time



5.1.4 Economic Advantage of Cycle Time and Product Quality

Working in tandem, pairs were able to complete their assignments in 58% of the elapsed time and with higher quality than the individuals. In today's competitive market, getting a quality product out as fast as possible is a competitive advantage or can even mean survival. [60] stresses the importance of examining both the technical value and the business value of process improvements. "And, decreased time to market, as well as improved product quality, are perceived as offering business value, too [60]." Intuitively, this is easy to believe.

Erdogmus [56], however, builds an economic model to establish quantitatively our intuition and to confirm the importance of rapid development and superior quality. The model incorporates several lower-level metrics into a Net Present Value Incentive (NPVI) measure. The whole model is defined in Appendix F. Two of the lower-level metrics are Early Entry Advantage (EEA) and Quality/Functionality Advantage (QFA). (EEA considers whether the market is ripe for the end product and that maximum reward is achieved through immediate entry.) If the test strategy has favorable values for these two metrics, as would be expected with collaborative programming, the Asset Value Advantage (AVA) for the test strategy is improved. Ultimately a larger AVA improves the NPVI of the test strategy making it a more desirable alternative.

5.2 Engineer Satisfaction

You know what I like about pair-programming? First, it's something that has shown to help produce quality products. But, it's also something that you can easily add to your process that people actually <u>want</u> to do. It's a conceptually

> *small thing to add . . . And, when times get tough, you wouldn't likely forget to do pair-programming or decide to drop it "just to get done." I just think the idea of working together is a winner.* [Chuck Allison in [35]]

The incorporation of pair-programming into CSP improves engineers' job satisfaction and overall confidence while attaining the quality and cycle time results discussed above. Pair programmers were surveyed six times on whether they enjoyed their job more when pair programming. First, an anonymous survey of professional pair programmers was run on the Internet. (The results of this survey are reported in Appendix D.) Both the summer and fall classes at the University of Utah were surveyed three times. Consistently, over 90% agreed that they enjoyed their job more when pair programming. The results are shown in Figure 11.

The groups were also surveyed on whether working collaboratively made them feel more confident about their work. These results are even more positive. (It is important to note that the fall class was surveyed for the first time before they were instructed on CSP and before they had ever pair-programmed.) The results are shown in Figure 12.

Figure 11: Pair Satisfaction

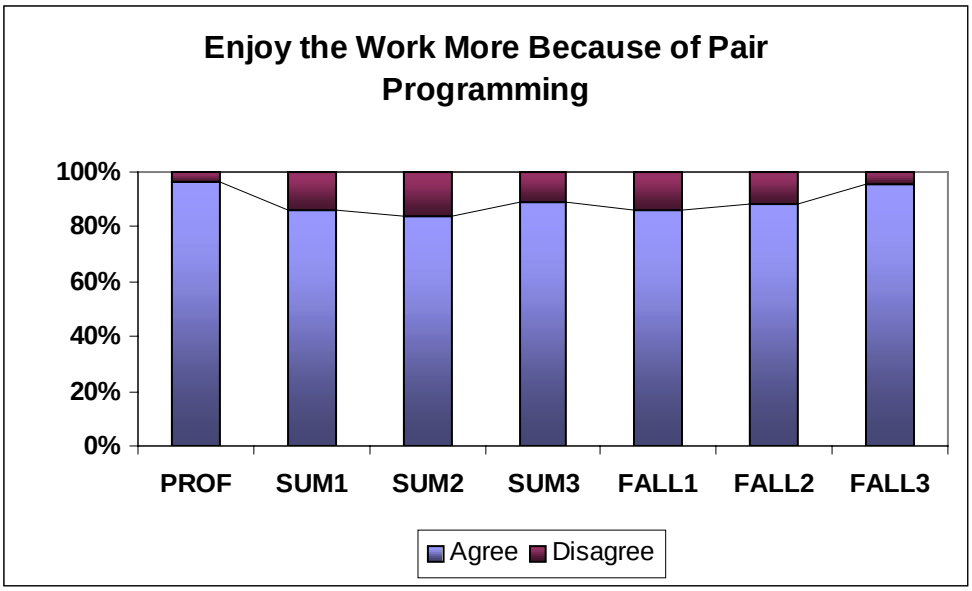
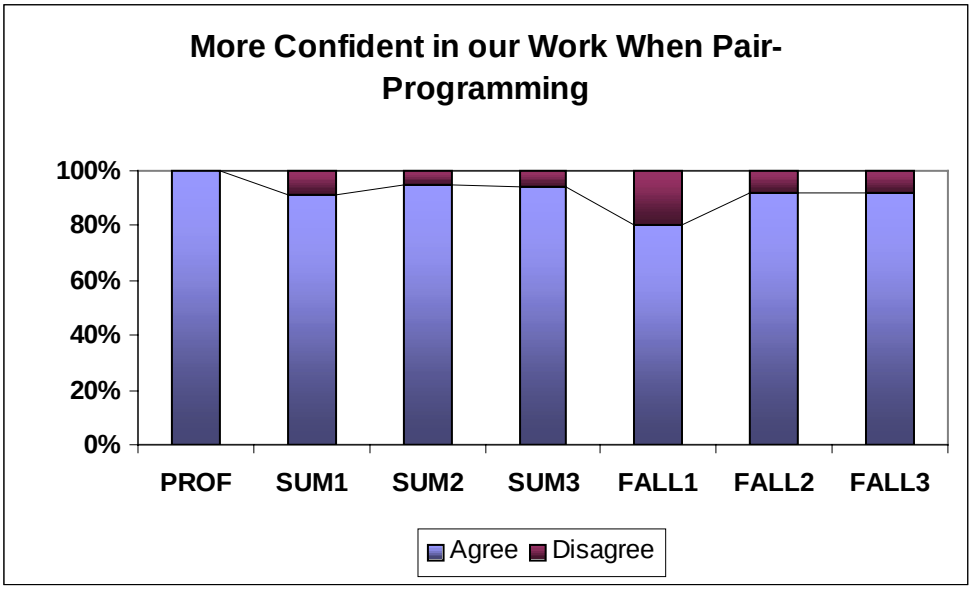


Figure 12: Pair Confidence



These results that show improved satisfaction and confidence cannot be taken lightly, because they could well be the difference between a pair actually following a disciplined process and reverting back to ad hoc procedures in the typically chaotic environment of software development. Often under pressure, engineers are tempted to stop the 'voluntary' defect prevention and efficient defect removal activities and are then faced with an overwhelming amount of 'involuntary' testing and debugging. Consider the following experience of PSP training in the TIS group at Hill Air Force Base. It is important to note that their high CMM maturity level indicates they are already a highly disciplined group, much higher than most software engineering organizations.

> *TIS is a high-maturity organization with a strong history of software process improvement. In March 1995, TIS was assessed as a CMM Level 3 organization, and the assessment conducted in July 1998 rated them at CMM Level 5. This is the first software engineering organization in the Department of Defense (DoD) to receive this rating, and it is one of the few Level 5 software groups in the world . . . During the summer of 1996, TIS introduced the PSP to a small group of software engineers. Although the training was generally well received, use of the PSP in TIS stated to decline as soon as the classes were completed. Soon, none of the engineers who had been instructed in PSP techniques was using them on the job.* [61]"

The SEI is addressing this issue by increasing the awareness of the Team Software Process (TSP) [34] that puts a management structure and awareness around PSP use by engineers. However, the collaborative programming can aid in the long term use of a disciplined process. First, engineers would not likely "forget" to work with their partner (as can often be done with activities such as design or code reviews) when under stressful situations. Indeed, they want to work with their partner; they enjoy working with their partner. Further, pairs consistently report that pair-pressure causes them to follow procedures that they might otherwise discard if they were not working with a partner.

The comparisons made in this dissertation are between individuals following a disciplined process and collaborators following a disciplined process. Actual results might even be better than what has been shown if the individuals revert to an ad hoc process while the collaborators keep each other on the disciplined track.

5.3 Secondary Indications

5.3.1 Collaboration and Teamwork

The students did one four-week project in four-person teams. Seven teams were formed of two collaborative pairs and used the CSP as the underlying process for their code implementation. Three groups were formed of four individuals and used the PSP as the underlying process for their code implementation. The students worked at the CSP2.1/PSP2.1 level. They also used Watts Humphrey's Introductory Team Software Process (TSPi) [34]. Teams using the TSPi use PSP to guide their individual code implementation and use the team structure of TSPi to guide their team coordination activities. The purpose of this phase was to examine the hypothesis that the intercommunication effort associated with code/system integration between programmers on a development team is significantly reduced with the use of pair programming.

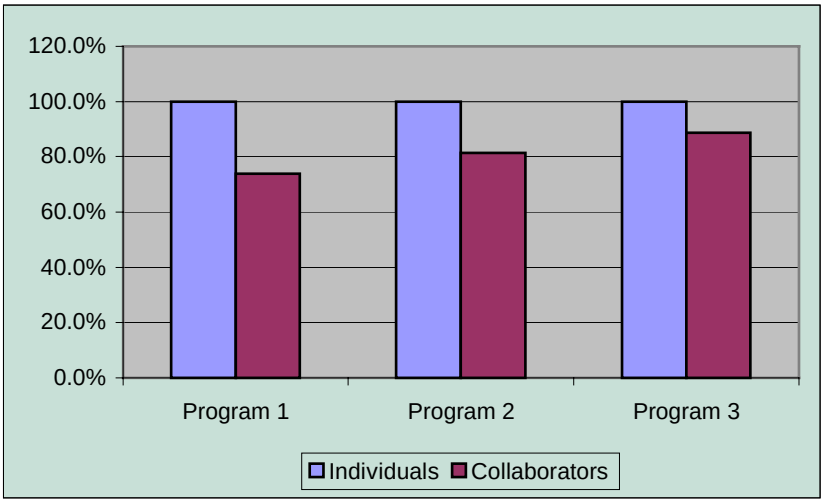
Unfortunately, the sample sizes of this phase were too small and did not yield statistically significant results. However, the results can be discussed for their merit. The groups formed from collaborative pairs spent 28% less time than the groups formed from individuals with a p value of 0.410. The groups formed from individuals actually passed 2% more test cases than the collaborative teams with a p value of 0.521. These results do indicate that collaborative teams are more efficient than teams consisting of

individuals and that collaborative teams can produce code of similar quality to that of individuals in less time. However, because the results are very far from being statistically significant, it is hard to draw any conclusions from the findings. Further investigation is necessary, as is indicated in the Future Work section of this document.

5.3.2 Design Quality

Observations of the students' code showed that the pairs produced superior high-level project designs. The individuals were more likely to produce "blob class [62]" designs -- just to get the job done. The design from the collaborative teams exploited more of the benefits of object-oriented programming. Their classes demonstrated more encapsulation and had more classes with better class-responsibility alignment. The individuals tended to have fewer classes that had many responsibilities. The collaborative designs would, therefore, be easier to implement, enhance and maintain. A confirmation of their superior designs is that the pairs consistently write less code than the individuals to achieve the same result but with higher quality. This could not happen if not for better, simpler, well thought-out designs. The student results are shown in Figure 13 below. It is a well known adage in industry that a good strategy for reducing maintenance costs is to reduce software size. Pair-programming helps in this goal. Also, Microsoft Chief Operating Officer Robert Herbold states that, "Our challenge is to make software simpler. [59]"

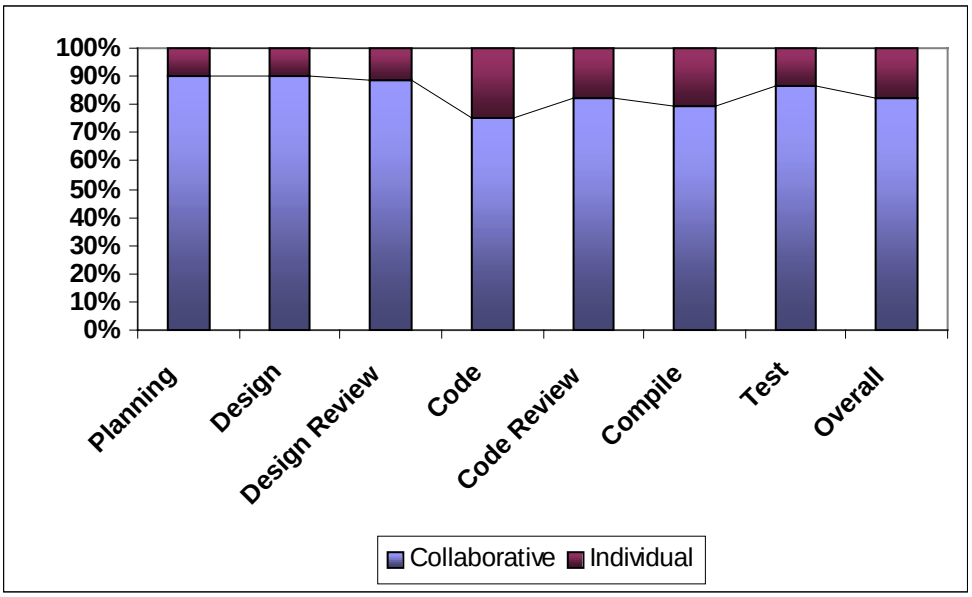
Figure 13: Relative Number of Lines of Code



5.3.3 Collaboration by Phase

Ideally, pair-programmers should work together constantly. However, reality dictates that at times the pair must split – for illness, time conflicts, or even efficiency. Over time, experienced pair programmers have prioritized which parts of the development cycle are most important to work together, which can be done separately, and what to do with the independently developed work when reuniting. This information has been derived from surveys of professional programmers and students and from the self-reporting time records of the students. A summary of the average collaboration by phase records for all programs of the students is shown in Figure 14.

Figure 14: Collaboration by Phase



5.3.3.1 Analysis and design

Unanimously, pair-programmers agree that collaborative analysis and design is critical for their pair success. First, it is important for the pair to collectively agree on the development direction and strategy outlined during these stages. Additionally, it is doubtless that "two brains are better than one" when performing analysis and design. Together, pairs have been found to consider many more possible solutions to a problem and more quickly converge on which is best to implement. Their constant feedback, debate, and idea exchange significantly decreases the probability of proceeding with a bad design. Perhaps, the collaborators can perform tasks that might be just too challenging for one to do alone.

While one partner is busy typing or writing down the design, the other partner can think more strategically about the implications of the design and can perform a continuous design review -- considering whether the design will run into a dead end or if there is a better strategy. Design defects are prevented or removed almost as soon as they hit the paper. A further benefit is the reduction of "design tunnel vision," which occurs when one makes a design decision and sticks with it no matter what. With the partner reviewing and questioning decisions, the chance of exploring good design alternatives is increased.

This investment in dual analyzers and developers is wise. [10] reports that typically no more than 33% of development effort is spent in the pre-coding phases. However, 68% of the testing field errors and more than 83% of the defect removal effort is focused on fixing complex defects that were injected in the design phase.

5.3.3.2 Code Implementation

After developing a quality design, the pair must implement it. Interestingly, programmers view pair-analysis and design as more critical than pair-implementation. Pairs report that they plan to code individually at times. They often deliberately split for the more rote, routine, simple coding of a project. They find performing this type of programming is more effective done individually. It seems that some tasks, such as GUI drawing, are largely detail-oriented in nature. Developers report that having a partner for this work doesn’t help much. Additionally, they do allow themselves to code average complexity modules if the situation, such as time conflicts, dictates – though most immediately feel notably uncomfortable and more error prone. Some, particularly

the Extreme Programmers, profess that any work done individually should be scrapped and redone by the pair. However, most programmers perform a thorough review of the individual work and incorporate it into the project. A small minority integrates individual work without review.

5.3.3.3 Testing

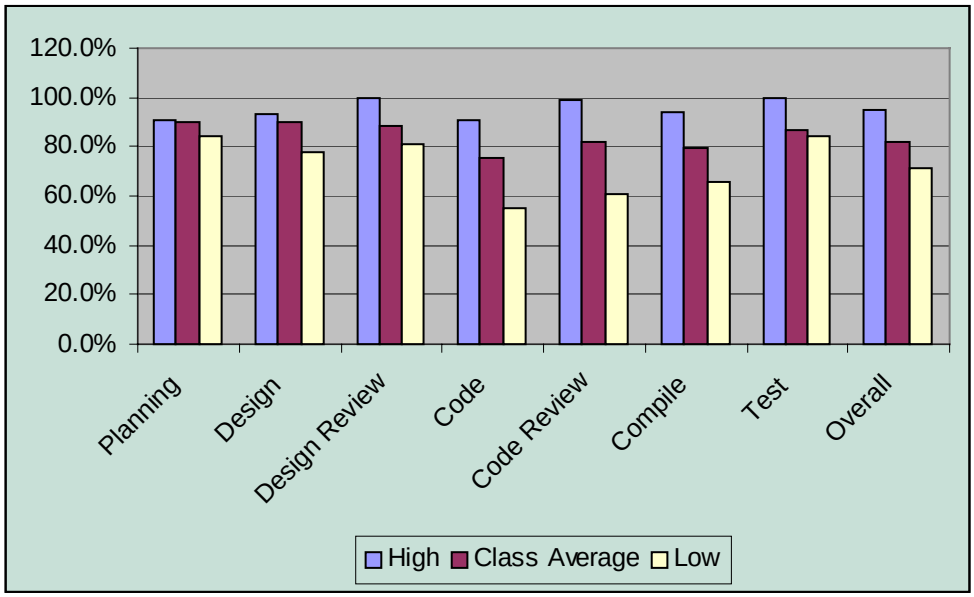
Pairs report that they consistently develop the test cases together. Sometimes, however, they split up to run test cases, often side-by-side at two computers. When defects are uncovered, the pairs usually rejoin to collaborate to find the best solution.

(Much of this information has previously been reported in [35])

5.3.3.4 Collaboration Among the High and Low Academic Performers

High academic performers tend to collaborate more than lower academic performers. Based on their past GPA, members of the class were classified as high (top 25%), middle (mid 50%), and low (bottom 25%) past academic performers. The percentage of collaboration by phase was examined for pairs with at least one high performer and for pairs with at least one low performer. The results are shown in Figure 15. Doubtlessly, the groups with at least one high performer collaborated significantly more than the groups with at least one low performer. In an academic setting, the students must make the effort to coordinate schedules in order to collaborate. The high achievers saw enough value in collaboration to put forth this effort consistently. The lower achievers did not make as much effort to collaborate, though they still did collaborate more than 70% of the time.

Figure 15: Average Collaboration by Phase for Performance Types



5.3.4 Collaboration Perhaps Not for All

All students took the Meyer-Briggs [63, 64] test, which has been used in many studies to indicate personality type. The personality test classifies people into one of sixteen personality types.

The Meyers-Briggs' sixteen personality types are based on combinations of four indicators. The test indicates which of each of these four indicators is stronger. These four indicators are briefly described:

- Extraversion or Introversion: Extraverts (E) are talkative and social. Introverts (I) are quiet and private.
- Sensation or Intuition: Indicates whether a person is more likely to obtain input from external observation (S) or introspective inner feelings (I)

- Thinking or Feeling: Thinkers (T) govern themselves with their heads, their concepts and their percepts. Feelers (F) follow their heart, which means most of what they do is based on emotion or desire.
- Judgment or Perception: People with high judgment (J) make and keep schedules in their daily life. People who rank high in perception (P) prefer to probe for options and not be tied to a schedule.

Almost half (19/41) ranked high in both Introversion (I) and Judgment (J). (In Meyer's Briggs' notation, this makes them IxxJ personality types.) This means many in the class were introverted and followed a schedule daily. These results are not surprising for a class of computer scientists.

However, it is notable that on the first day of class, six of the seven students that indicated they did not want to try collaborative programming were IxxJ personality types. Perhaps they were resistant to needing to constantly communicate with and to coordinate schedules with another. This is not seen as an insurmountable personality type that is a barrier to collaborative programming, though. The other 13 IxxJ students were very successful collaborators.

In Appendix D are the results of a survey of professional collaborative programmers. A question on the survey probed whether the programmer had ever tried and failed to pair with another engineer. Most indicated that they were never unable to collaborate. Excess or too little ego, not personality type, was cited as the main problem in those that did have difficulty.

5.3.5 Gender and Personality-Type Considerations

There were six females and 35 males in the experimental class. As outlined in Appendix A, when student pairs were formed, specific attention was placed to ensure that groups were made up of a diversity of (seven) male-male, (four) male-female, and (one) female-female pairs. Additionally, attention was placed on ensuring a variety of Meyers-Briggs personality type combinations. These factors were carefully considered in order to study possible factors for successful collaboration. However, there were no conclusive findings based on gender or personality types. This is mainly because all pairs were deemed to be successful collaborators. There were no trends of the following:

- groups not getting along
- groups consistently spending more or less time completing their task than the average
- groups consistently achieving far better or worse quality than the average

Performance differences could easily be attributed to past academic performance.

CHAPTER 6

SUMMARY AND CONTRIBUTIONS

Dr. W. Edwards Deming, legendary quality consultant, led sweeping manufacturing quality revolutions in both Japan and the US beginning in the 1950's. His teachings dramatically altered the economy of Japan (creating the opportunity for the US to "catch up"). His Total Quality Management (TQM) practice stressed the importance of studying and understanding in great depth the process of the production or service you are delivering. He defined "Deming's 85/15 rule: 85% of a worker's effectiveness is determined by the system he works within, only 15% by his own skill [9]." Considering these of Deming's philosophies, the merits of the research outlined in this dissertation will be discussed.

6.1 Studying and Understanding the Process

Deming stresses the importance of studying and understanding your process. His message to manufacturers was clear: Design in quality at the beginning of the development process, instead of "testing in" pseudo-quality at the end of the production line [59]. The Collaborative Software Process (CSP) synchronizes with this philosophy. A significant assumption behind the designation of CSP as a high quality process is the use of pair-programming. With pair-programming, two software engineers work side-by-side at one computer – together producing one design, algorithm, code, or test arti-

fact. At any one time, one of the engineers is the “driver” who is actively creating and recording the artifact. The other partner is constantly observing, critiquing, strategizing on, and improving upon the work of the driver. The engineers periodically switch roles. Both are continuous, active participants in their joint creation.

Requirements analysis begins with the development of use classes, which are thoroughly explored through the development of the Use Case Flow of Events. The scenarios, which emerge through the development of use cases, are used in the CRC Card brainstorming session. The goal of this brainstorming session is the interactive development of a high-level class diagram. The pair then reviews the design.

Once the design is documented and reviewed, the collaborative pair begins the iterative process of developing a high quality implementation. The high-level design is broken into smaller increments. Together they iteratively perform low-level design and review, create test cases and code, perform a code review, compile and execute test cases.

Throughout all this process, the pair is recording information about the amount of time they spend on various stages of the process and about the defects they find and remove from their product. By following defined procedures, this data is turned into valuable information the pair can used to evaluate the effectiveness of their process and to adjust their joint process accordingly. Using this information as well as qualitative knowledge of how their process went, the pair documents what worked and what did not work about what they did in order to perform continual process improvement.

In analyzing Deming’s directive “understand your process for delivering a quality product” CSP excels. The Personal Software Process (PSP) has been shown to be a sig-

nificant improvement over mainstream, ad hoc software development. When compared quantitatively, the CSP demonstrates a marked improvement over the PSP for delivering high quality products. The incorporation of pair-programming paves the way for continually improving defect prevention and extremely efficient defect removal. The process incorporates a systematic, continual evaluation of the overall process effectiveness.

6.2 The System the Engineer Works In

Deming asserts that 85% of a worker's effectiveness is determined by the system he works within, only 15% by his own skill. A defined, quality process, such as the CSP, provides a disciplined system for the engineer to work in. However, the collaboration of the CSP maximizes the performance of the engineer beyond his or her own skill. In fact, the collaboration of the CSP has been shown to improve the engineer's own skill through various factions of pair-learning – through apprenticeship and through the continual design and code reviews that take place.

Collaboration has been shown to improve the engineers' joint ability to derive the best solutions and to evolve solutions to unruly or seemingly impossible problems. Theories from the science of Distributed Cognition support the notion that different skills and perspectives each of the pair bring to the task and their desire to succeed at a common goal causes them to explore a larger number of alternatives and to negotiate the best course of action. Pairs are also observed to attack hard problems with a "tag-team" approach whereby each partner, in turn, incrementally contributes to the solution.

Collaboration also improves the system in which the engineers work. Pairs tend to positively pressure their partners into a higher level of performance and into consistently following the prescribed process. In industry, the engineers generally pair with different partners on a regular basis. This has been shown to reduce the risk to the project of several individuals being the primary technological assets of the project; the untimely loss (through job change or accident) of any of these individuals could devastate the project. Additionally, pair rotation has been shown to greatly improve teamwork and communication among the team. Lastly, almost unanimously, pair programmers claim to be happier and more confident on the job when pair programming.

6.3 Summary of Contributions

The two primary contributions of this research are the Collaborative Software Process and the quantification of the benefits of collaborative programming. Mature engineering disciplines generally follow proven, documented procedures to reliably produce high quality products. As Software Engineering strives to mature, proven processes are necessary. The Collaborative Software Process makes a contribution. This research has defined and validated the effectiveness of a disciplined process for a collaborative pair of software engineers.

The quantitative study that validated the effectiveness of CSP also legitimized the practice of pair-programming. In the experiment, the independent variable between the control group and the experimental group was the use of pair-programming. Therefore, the increased quality demonstrated by the pairs can be attributed to the collaboration. Additionally, the pairs spent only statistically insignificant more time working on their

programs. Therefore, the quality gains can be realized with little or no increase in development time. Factoring in field support cost savings of increased quality and the benefits of reduced cycle time, the collaborative teams are less expensive overall. Additionally, many quantitative benefits to the software firm and to the individual are realized with pair programming. In short, all paths point to pair-programming.

One professional programmer reported that, despite many successes, pair-programmers were ordered to work individually at his workplace. His management simply could not believe that it was cost effective. The results of this research have already been used by many professional programmers who want to justify continuing and to justify initiating the prevalent use of pair programming as they strive to delight their customers with high quality products on schedule.

CHAPTER 7

FUTURE WORK

These findings have spawned many more research ideas:

1. **Industrial Validation.** The validation of the CSP was a carefully planned empirical study. An important consideration in empirical research design is external validity, the ability of the experimental results to apply to the world outside the research situation. The results outlined in the dissertation are conclusive and can be meaningfully applied to professional programmers because the research studied the interactions between and efficiencies of two programmers working collaboratively. These issues would not be complicated by the complexity or scale found in industry. Indeed, several professional programming organizations have begun to justify pair-programming from the publications resulting from this research. Nonetheless, formal re-validation of the results with professional programmers in an industrial setting would be beneficial.
2. **Brook's Law.** Over 25 years ago, Frederick Brooks taught us that *adding manpower to a late project makes it later* due in a large part to added communication costs. Pairing programmers cuts the necessary communication paths in half. It would be interesting to study the effects of collaborative programming in a larger team setting to examine the communication efficiencies.

3. **Distributed Cognition.** Many of the observations and findings on collaborative programming can be supported by theories in the distributed cognition area of cognitive science. Investigating these ties further would be interesting research.
4. **Distributed Collaboration.** An exciting area of Computer Science is Distributed Collaboration. Many tools are being developed to support collaboration between team members who are physically separated. Further study could analyze whether the efficiencies and gains seen with physically co-located programmers can be realized with distributed pairs.
5. **Pair-Learning**. "Traditionally, collaboration in the classroom . . . has been taboo, condemned as a form of cheating. Yet what we discover . . . is that collaboration can only make our classrooms happier and more productive [38]." An unexpected result of the experiment was the observations of the intense benefits of pair-learning and pair-programming for students learning a new programming language. Students are happier and learn faster. Pairs continuously teach each other. Students no longer look solely to the teaching staff for technical help, and therefore the workload of the teaching staff is reduced. It would be useful to quantify these benefits in order to justify the benefits of pair-learning to other instructors. Indeed, Larry Constantine, who's observation of P. J. Plaugher's software company were reported in the Related Works chapter, noted that ". . . for language learning, there seems to be an optimum number of students per terminal. It's not one . . . one student working alone generally learns the language significantly more slowly than when paired up with a partner [19]."

6. **eXtreme Programming.** The eXtreme Programming methodology employs many techniques counter to currently accepted software engineering practice. Therefore, isolating which factor to attribute its reported success is difficult. This dissertation research established that employment of pair-programming contributes to their success. Additional research could help dissect additional contributing factors.
7. **Economic Net Present Value Analysis.** Section 5.1.4 and Appendix F utilize an economic model [56], which incorporates various lower-level metrics into Net Present Value Analysis. Most of this model is concretely defined in mathematical formulae. However, one of the inputs, Quality/Function Advantage (QFA), is still theoretical. Research into defining a concrete formula for QFA would complete the model. The model could then be utilized for a complete, concrete evaluation of the Net Present Value of software process alternatives that improve the quality of software.

APPENDIX A

EXPERIMENTAL DESIGN

The validation of the Collaborative Software Process was based on an empirical study of students at the University of Utah. The details of this experiment were submitted to the Institutional Review Board (IRB) at the University of Utah. The role of the IRB is to determine if they believe the rights of the students will be violated in any way by their participation in an experiment. The IRB deemed that this study was exempt from their surveillance. In order to be declared exempt, an experiment must be conducted in an established educational setting and involve normal educational practices in order to evaluate or compare regular or special educational instructional strategies, curricula or methods.

The study was based on two courses taught in Summer Semester 1999 and Fall Semester 1999. The Summer class was an exploratory, preparatory class in which CSP was initially used and reviewed. Based on student feedback, CSP was revised for the Fall class. The Fall class was the primary experimental class through which the majority of the quantitative evidence was obtained. A Windows NT data collection and analysis web application was developed as part of this research. The application was used to accurately obtain data from and provide feedback to the students, as easily as possible for the students. The details of both classes will be examined in this section.

Summer 1999

The class, Collaborative Development of Active Server Pages, consisted of 20 juniors and seniors. The students were very familiar with programming, but not with the Active Server Pages (ASP) web programming languages learned and used in the class. One class period per week was spent learning the Collaborative Software Process. The other class period each week was spent learning the web programming languages. The students applied their newly acquired CSP knowledge and practices when developing the class assigned web programming projects.

The majority of the students had only used WYSIWYG web page editors prior to taking the class. During the eleven-week semester, the students learned advanced HTML, JavaScript, VBScript, Active Server Page Scripting, Microsoft Access/SQL and some ActiveX commands. In many cases, the students would need to intertwine statements from all these languages in one program listing – some of the content running on the browser and some running on the NT server, adding to the overall complexity of the program. Upon course completion, the students were all writing web scripts that had significant dynamic content that accessed and updated a Microsoft Access database – applications similar (though smaller) to what you would find on a typical e-commerce web site.

Each student was paired with another student to work with for the entire semester. At the start of the class, the students were asked whom they wanted to work with and whom they did not want to work with. Of the ten collaborative pairs, eight pairs were mutually chosen in that each student had asked to work with their partner. The last two pairs were assigned because the students did not express a partner preference. Tests

were, however, taken individually. They understood that they were not to break the class project into two pieces and integrate later. Instead they were to work together (almost) all the time on one product. These requirements were stated in the course announcement and were re-stated at the start of the class. The students received instruction in effective pair-programming and read a paper [48] which helped prepare them for their collaborative experience. Most skeptically, but enthusiastically, embarked on making the transition from solo to collaborative programming.

The students kept a password protected web-page journal during the class in which they recorded their impressions of using CSP each week. Each week they were given a different set of questions to answer in their journal. Some example questions are listed below:

1. It has been said among teachers, “You do not know it unless you can teach it.” Do you find any value to yourself in explaining your work to your partner?
2. Do you feel like you have learned anything about Active Server Pages programming just by reading your partner’s code?
3. What was the biggest hurdle you have had to overcome as a collaborative programmer?
4. What kinds of things does the non-driver do as he/she observes?
5. Which development phases have you tried to work together the most?
6. If you work separately, what do you do with the separate work when you get back together?

7. Which development phases have you found it is OK to work separately at times on?
8. What do you think is the biggest advantage of collaborative programming?
9. What do you think is the biggest problem with collaborative programming?

Additionally, three times throughout the semester, the students completed anonymous surveys on their collaborative experience. Lastly, as part of the final exam, the students wrote a letter objectively giving advice to future collaborative programmers. These observations and critiques were used to update and enhance the CSP process prior to a structured experiment and were reported in several papers [36, Williams, submitted to IEEE Software #45, Cockburn, 2000 #58] and throughout this document.

Fall 1999

The class, Senior Software Engineering (CS4510), consisted of 41 juniors and seniors. The students learned of the experiment during the first class. They had to be informed that it is an experiment because, as outlined below, some students completed class programming projects individually and some worked in pairs. Additionally, they were strongly encouraged to report all data accurately during the semester because of the importance of the outcome. Generally, the students responded very favorably to being part of an experiment that could drastically change the way software development could be performed in the future.

On the first day of class, the students were asked if they preferred to work collaboratively or individually, whom they wanted to work with, and whom they did not want to work with. Additionally, the students took a Meyers Briggs personality test [63]. The students were also classified as “High” (top 25%), “Average,” or “Low” (bottom 25%)

academic performers based on their GPA. The GPA was not self-reported; academic records were reviewed.

Using this information, the twenty-eight students were then assigned to Group C (Collaborative) and thirteen to Group I (Individual). (Several students dropped the class after the initial assignment. These numbers reflect the students who completed the class.) Group C students completed their collaborative assignments using the CSP. Group I students completed all assignments using a modified version of PSP. The PSP was modified from that defined in [11] in order to parallel the CSP (e.g. use cases). (Differences between PSP and CSP are outlined in Chapter 3.) The students were assured that grades would be curved, as necessary, independently for each of the groups to ensure that neither group would get an advantage in being academically successful in the class.

The GPA was used to ensure that the groups are academically equivalent. Ultimately (after the students had dropped out), Group C consisted of 7 high performers, 16 average performers, and 5 low performers. Group I consisted of 5 high performers, 5 average performers, and 3 low performers. The students were also grouped to ensure there was a sufficient spread of high-high, high-average, high-low, average-average, average-low, and low-low pair grouping. This was done in order to study the possible relationship between previous academic performance and successful collaboration.

Of the fourteen collaborative pairs, thirteen pairs were mutually chosen in that each student had asked to work with their partner. The last pair was assigned because the students did not express a partner preference.

The students from both groups received instruction in effective pair-programming and were given a paper [48] and several of the “letters to a future collaborative programmer” written by the Summer Semester class. These helped prepare them for their collaborative experience.

The majority of the students were familiar with the Personal Software Process (PSP), on which CSP was based, because they had been instructed on it in their CS1 and CS2 class using the *Introduction to the Personal Software Process* book [40]. The CS4510 used the more advanced PSP book, *A Discipline for Software Engineering* [11]. The PSP is documented through many process scripts, templates and forms. The CSP, modeled on the PSP, also uses this documentation framework. Between the two processes, some of the documentation artifacts are very similar; others might be significantly different. In the cases where a particular artifact differs between the CSP and the PSP, both were taught to the entire class. For example, there was a class dedicated to the code review sub-process. The procedures for doing code review alone (PSP) and for doing code review collaboratively (CSP) were both be discussed and contrasted. All aspects of the development cycles for both PSP and CSP were taught to all students.

The experiment proceeded in phases as defined below:

Pre-treatment: Each student completed one program individually, using PSP Level 0, which is essentially their current process with the addition of tracking the amount of time they spend on the program and the defects they remove during their process. This phase got them used to the data entry procedures. The data was used as a "pre-treatment" baseline for all students. The purpose of the pre-treatment was to determine if there were any significant performance changes for individuals when they worked in-

dividually versus when they worked collaboratively. No such changes were consistently observed. Performance in the class was very coherent with the student's past academic performance.

Treatment: Four assignment cycles were completed during the treatment phase of the experiment. During each cycle, the individuals completed one assignment and each collaborative team completed two assignments. During this phase, the students progressed from PSP/CSP0.1 to PSP/CSP2.0.

Post-treatment: Each student completed one program individually. The data was used as a "post-treatment" measure for all students. As with the pre-treatment, the purpose of the post-treatment was to determine if there were any significant performance changes for individuals when they worked individually versus when they worked collaboratively. No such changes were consistently observed. Performance in the class was very coherent with the student's past academic performance. Therefore, no additional findings were gained by the post-treatment.

Team: The students did one four-week project in four-person teams. Seven Group C teams consisted of two collaborative pairs and used the CSP as the underlying process for their code implementation. Three Group I students consisted of four individuals and used the PSP as the underlying process for their code implementation. The students worked at the CSP2.1/PSP2.1 level. They also used Watts Humphrey's Introductory Team Software Process (TSPi) [34]. Teams using the TSPi use PSP to guide their individual code implementation and use the team structure of TSPi to guide their team coordination activities.

The purpose of this phase was to examine the hypothesis that the intercommunication effort associated with code/system integration between programmers on a development team is significantly reduced with the use of pair programming. Unfortunately, the sample sizes of this phase were too small and did not yield statistically significant results.

It must be noted that in both the summer and the fall classes, specific measures were taken to ensure that the pairs worked together consistently each week. In the summer class, one day each week the formal Active Server Page instruction was followed by exercises in which the pairs worked together. In the fall class, one class period each week was allotted for the students to work on their projects. Additionally, the students were required to attend two hours of office hours with their partners each week where they also worked on their projects. It is critical for **student** pairing success to establish these regular meeting times, lest the students get too involved in other classes and their jobs and never get together. During these regular meeting time, the pairs jelled or bonded and were much more likely to establish additional meeting times to complete their work.

Experiment Validity

Specific details of this empirical study have been designed to adhere to principles of good research study design, as outlined in [65].

A common research study threat is caused by the Hawthorne effect. "The Hawthorne effect refers to a change in sensitivity, performance, or both by the subjects that may occur merely as a function of being in an investigation . . . The Hawthorne effect be-

comes a threat to internal validity when one group receives such a "special" treatment and another does not, thereby introducing a systematic difference between groups in addition to the experimental variable [65]." Since both groups will receive the same information about the study and in all lecture materials, the Hawthorne effect should not pose a threat to this study.

Factors to ensure internal validity were carefully considered. Drew defines internal validity as:

> *The technical soundness of a study. A study is internally valid or has high internal validity when all the potential factors that might influence the data are controlled except the one under study. This would mean that the concept of control had been successfully implemented. If, for example, two instructional methods were being compared, internal validity would require that all differences between groups (e.g. intelligence, age) be removed except the differences in the instructional method, which is the experimental variable* [65].

In this case, the experimental variable is the act of solo programming vs the act of collaborative programming. Efforts have been made to remove other differences between the groups. GPA statistics were analyzed to balance the potential for success of both groups. All students received the same information. Software processes, such as the CSP and PSP, define steps for developing software to achieve predictable results. This process structure also improves the internal validity of the study because all the students should be using the same defined, repeatable process to develop their assignments.

Factors affecting external validity were also carefully considered to ensure the research and the experiment results will be considered viable by researchers and practitioners. Drew defines external validity as:

> *The generalizability of results from a given study. External validity involves how well the results of a particular study apply to the world outside the research situation. If a study is externally valid or has considerable external validity, one can expect that the results are generalizable to a considerable degree* [65].

Often empirical software engineering studies involving students are not highly regarded research because it is not viewed that projects done in a semester need deal with issues of scope or scale that often complicate real, industrial projections. Even Watts Humphrey – working in an industrial organization, did his initial PSP studies on students, because he could not find any real project that would risk its success on a new process. However, he says "You can apply PSP principles to almost any software-engineering task because its structure is simple and independent of technology -- it prescribes no specific languages, tools or design methods." His study, involving students, was highly regarded, respectable research. CSP, though on a slightly larger scale because it involves two programmers, can be considered likewise. Also, an experiment was performed on seniors at Carnegie Mellon involving communication metrics for software development. "Such a test-bed represents an ideal environment for empirical software engineering, providing sufficient realism while allowing for controlled observation of important project parameters [66]." This empirical study of this dissertation involves the interactions between and efficiencies of two programmers working collaboratively. Issues of complexity and scale are not inhibitors to the external validity of a study of CSP with students at the University of Utah.

APPENDIX B

THE COLLABORATIVE SOFTWARE PROCESS (CSP) DOCUMENTATION

Table 4: CSP Documentation Cross-Reference

	Table Number						
Process Level	**All**	**CSP0**	**CSP0.1**	**CSP1**	**CSP1.1**	**CSP2**	**CSP2.1**
Process Scripts and Summaries							
Process Script		4	14	23	29	41	45
Planning Script		5	15	24	24	42	46
Development Script		6	16	25	30	30	47
Postmortem Script		7	17	26	31	31	48
Project Plan Summary and Instructions		8 9	18 19	18 19	32 33	43 44	43 44
PROBE Estimating Script	See [11] Appendix C Table C36					X	X
Forms, Templates, Standards and Instructions							
Time Recording Log	10 11	X	X	X	X	X	X
Defect Recording Log	12 13	X	X	X	X	X	X
Process Improvement Proposal	20 21		X	X	X	X	X
Coding Standard	22		X	X	X	X	X
Use Case Flow of Events and Instructions	27 28			X	X	X	X
Individual Code Review Checklist	34				X	X	X
Collaborative	35				X	X	X

	Table Number						
Process Level	**All**	**CSP0**	**CSP0.1**	**CSP1**	**CSP1.1**	**CSP2**	**CSP2.1**
Code Review Checklist							
Individual Design Review Checklist	36				X	X	X
Collaborative Design Review Checklist	37				X	X	X
Test Case Template and Instructions	38 39				X	X	X
Test Coverage Checklist	40				X	X	X
Size Estimating Template	See [11] Appendix C Tables C39 and C40					X	X
Task Planning Template	See [11] Appendix C Tables C47 and C48						X
Schedule Planning Template	See [11] Appendix C Tables C49 and C50						X

Note: An X indicates that the form, template or script indicated in the “All” column is appropriate at that level. (Conversely, a blank square indicates that the form is not used at that level.)

Table 5: CSP0 Process Script

Phase Number	**Purpose**	**To guide you in collaboratively developing module-level programs**
	Entry criteria	• Problem description • Empty Project Plan Summary form • Empty Time and Defect Recording Logs • Stop watch (optional)
1	Planning	• Produce or obtain a requirement statement • Estimate the required development time of both partners • Enter the plan data in the Project Plan Summary form • Record the time spent in the Time Recording Log for Planning
2	Development	• Design the program • Implement the design • Compile the program and fix and log all defects found • Test the program and fix and log all defects found • Record the time spent in these activities in the Time Recording Log in the appropriate phase
3	Postmortem	• Complete the Project Plan Summary form with actual time, defect, and size data
	Exit Criteria	• A thoroughly tested program • Completed Project Plan Summary with estimated and actual data • Completed Defect and Time Recording Logs

Table 6: CSP0 Planning Script

Phase Number	**Purpose**	**To guide the CSP planning process**
	Entry criteria	• Problem description • Empty Project Plan Summary form • Empty Time Recording Log
1	Program Requirements	• Produce or obtain a requirements statement for the program • Ensure the requirements statement is clear and unambiguous • Resolve any questions
2	Estimate Resources	• Make your best estimate of the time (for both partners) required to develop this program
	Exit Criteria	• A documented requirements statement • Estimated development time data entered in the Project Plan Summary • Actual time spent planning entered in the Time Recording Log

Table 7: CSP0 Development Script

Phase Number	**Purpose**	**To guide the development of small programs**
	Entry criteria	• Requirements statement • Project Plan Summary with planning completed • Time and Defect Recording Logs with planning completed
Note: The terms ***driver*** and ***non-driver*** are used below. The driver is the partner who has control of the recording medium (ex: paper, computer keyboard) and is recording the design or implementing code or fixing code. The non-driver is the other partner who is actively observing the driver -- identifying defects, giving suggestions, etc. *When a partner is working alone, he or she is considered the driver, and no one is filling the non-driver role.*		
1	Design	• Review the requirements and produce a design to meet them via discussions between partners. • The driver records the design in pre-determined format/on pre-determined medium. • The non-driver observes to ensure the design is being recorded efficiently and effectively meets the requirements. The non-driver identifies defects and gives suggestions for alternative designs. • Periodically, switch drivers. • Record design time in the Time Recording Log
2	Code	• Implement the design. • The driver implements the design by typing code via the keyboard. • The non-driver is observes to ensure the code properly implements the design, identifying defects whenever necessary and giving suggestions for alternative implementations. • Periodically, switch drivers. • Record any requirements or design defects in the Defect Recording Log • Record coding time in the Time Recording Log
3	Compile	• Compile the program until error-free. • Both partners identify and discuss all defects found and the possible implications of these defects elsewhere in the code. • The driver implements the code changes by fixing code via the keyboard. • The non-driver observes to ensure the fix is properly

		implemented, identifying erroneous fix implementations. • Periodically, switch drivers. • Record all defects found in Defect Recording Log • Record compile time (until program compiles error-free) in the Time Recording Log
4	Test	• Test until all tests cases run <u>without error</u> • Both partners identify and discuss all defects found and the possible implications of these defects elsewhere in the code. • The driver implements the code changes by fixing code via the keyboard. • The non-driver observes to ensure the fix is properly implemented, identifying erroneous fix implementations. • Periodically, switch drivers. • Record defects in the Defect Recording Log • Record test time (until all test cases run error-free) in the Time Recording Log
	Exit Criteria	• A thoroughly tested program • Completed Defect Recording Log • Completed Time Recording Log

Table 8: CSP0 Postmortem Script

Phase Number	**Purpose**	**To guide the CSP postmortem process**
	Entry criteria	• Problem description and requirements statement. • Project Plan Summary with planned development time. • Completed Time Recording Log • Completed Defect Recording Log • A tested and running program
1	Defects Injected	• Determine from the Defect Recording Log the number of defects injected in each phase. • Enter this number under Defects Injected -- Actual on the Project Plan Summary
2	Defects Removed	• Determine from the Defect Recording Log the number of defects removed in each phase. • Enter this number under Defects Removed -- Actual on the Project Plan Summary
3	Time	• Review the completed Time Recording Log • Enter the total time spent in each phase under Actual on the Project Plan Summary
	Exit Criteria	• A fully tested program • Completed Project Plan Summary Form • Completed Defect Recording Log and Time Recording Log

Table 9: CSP0 Project Plan Summary

Student	_____	Date	_____
Program	_____	Program #	_____
Instructor	_____	Language	_____

Time in Phase (min.)	**Plan**	**Total Actual**	**To Date %**	**Individual**	**Collaborative**
Planning	_____	_____	_____	_____	_____
Design	_____	_____	_____	_____	_____
Code	_____	_____	_____	_____	_____
Compile	_____	_____	_____	_____	_____
Test	_____	_____	_____	_____	_____
Postmortem	_____	_____	_____	_____	_____
Total	_____	_____	_____	_____	_____

Defects Injected	**Actual**	**Individual**	**Collaborative**
Planning	_____	_____	_____
Design	_____	_____	_____
Code	_____	_____	_____
Compile	_____	_____	_____
Test	_____	_____	_____
Total	_____	_____	_____

Defects Removed	**Actual**	**Individual**	**Collaborative**
Planning	_____	_____	_____
Design	_____	_____	_____
Code	_____	_____	_____
Compile	_____	_____	_____
Test	_____	_____	_____
Total	_____	_____	_____

Table 10: CSP0 Project Plan Summary Instructions

Purpose	**This form holds the estimated and actual project data in a convenient and readily retrievable form**
Header	Enter the following: • Your name and today's date. • The program name and number. • The instructor's name. • The language you used to write the program.
Time in Phase	• Under Plan, enter your original estimate of the total development time. • Under Actual, enter the total actual time in minutes spent in each development phase (should be the sum of the Individual and Collaborative time). • Under Individual, enter the total actual time spent by any partner individually • Under Collaborative, enter the total actual time spent the partners collaboratively • Under To Date %, enter the percentage of Total time (versus your plan) in each phase.
Defects Injected	• Under Actual, enter the number of defects injected in each phase (should be the sum of the Individual and Collaborative defects). • Under Individual, enter the total defects time injected by phase when a partner was working individually • Under Individual, enter the total defects time injected by phase when the partners were working collaboratively
Defects Removed	• Under Actual, enter the number of defects removed in each phase (should be the sum of the Individual and Collaborative defects). • Under Individual, enter the total defects time removed by phase when a partner was working individually • Under Individual, enter the total defects time removed by phase when the partners were working collaboratively

Table 11: Time Recording Log

Student ______________________ Date ______________________

Instructor ______________________ Class ______________________

Program

Date	Start	Stop	Interrupt Time	Delta Time	Phase	Collab or Indiv	Comment

Table 12: Time Recording Log Instructions

Purpose	This form is for recording the time spent in each project phase. These data are used to complete the Project Plan Summary
General	• Record all the time you spend on the project. • Record the time in minutes. • Be as accurate as possible.
Header	Enter the following: • Your name • Today's date • The instructor's name • The number of the program
Date	Enter the date when the work was performed.
Example	10/18
Start	Enter the time when you start working on a task.
Example	8:20
Stop	Enter the time when you stop working on that task.
Example	10:56
Interruption Time	Record any interruption time that was not spent working on the task and the reason for the interruption. If you have several interruptions, enter their total time.
Example	37 min -- took a break
Delta Time	Enter the clock time you actually spent working on the task, less the interruption time.
Example	From 8:20 to 10:56, less 37 minutes, or 119 minutes.
Phase	Enter the name of the development phase being worked on.
Example	Planning, Design, Design Review, Code, Code Review, Compile, Test
Collab or Indiv	Enter C if the work was performed collaboratively. Enter I if the work was done individually.
Comments	Enter any other pertinent comments that may later remind you of any unusual circumstances regarding this activity.
Example	"Had a compiler problem, had to get help."
Important	It is important to record all worked time. If you forget to record the starting, stopping, or interruption time for a task, promptly enter your best estimate of the time.

Table 13: Defect Recording Log

Student _____ Date _____
Instructor _____ Class _____
Program

Date	Def. Num.	Phase Injected	Phase Removed	Fix Time	Collab or In-div	Description

Table 14: Defect Recording Log Instructions

Purpose	This form is for recording each defect you find and fix. These data are used to complete the Project Plan Summary
General	• Record all the defects you find in review, compile and test. • Record each defect separately and completely. • Be as accurate as possible.
Header	Enter the following: • Your name • Today's date • The instructor's name • The number of the program
Number	Enter the defect number. For each program, this should be a sequential number starting with 1.
Phase Injected	Enter the phase during which this defect was injected. Use your best judgment.
Phase Removed	Enter the phase during which this defect was removed. This will generally be the phase during which you found the defect.
Fix Time	Enter your best judgment of the time you took to fix the defect. This time can be determined by using a stop watch or your judgment.
Collab or Indiv	Using your best judgment, enter C if you believe the defect was injected during collaborative work. Enter I if you believe the defect was injected during individual work.
Description	Write a succinct description of the defect that is clear enough to later remind you about the error and help you to remember why you made it.

Table 15: CSP0.1 Process Script

Phase Number	**Purpose**	**To guide you in collaboratively developing module-level programs**
	Entry criteria	• Problem description • Empty Project Plan Summary form • Empty Time and Defect Recording Logs • Stop watch (optional)
1	Planning	• Produce or obtain a requirement statement • ***Estimate the total new and changed LOC required*** • Estimate the required development time of both partners • Enter the plan data in the Project Plan Summary form • Record the time spent in the Time Recording Log for Planning
2	Development	• Design the program • Implement the design • Compile the program and fix and log all defects found • Test the program and fix and log all defects found • Record the time spent in these activities in the Time Recording Log in the appropriate phase
3	Postmortem	• Complete the Project Plan Summary form with actual time, defect, and size data
	Exit Criteria	• A thoroughly tested program • Completed Project Plan Summary with estimated and actual data • ***Completed Process Improvement Proposal (PIP) form*** • Completed Defect and Time Recording Logs

Table 16: CSP0.1 Planning Script

Phase Number	**Purpose**	**To guide the CSP planning process**
	Entry criteria	• Problem description • Empty Project Plan Summary form • Empty Time Recording Log
1	Program Requirements	• Produce or obtain a requirements statement for the program • Ensure the requirements statement is clear and unambiguous • Resolve any questions
2	Size Estimate	• ***Make your best estimate of the total new and changed LOC required to develop this program***
3	Resource Estimate	• Make your best estimate of the time (for both partners) required to develop this program • ***Make your best estimate of the total new and changed LOC required to develop this program***
	Exit Criteria	• A documented requirements statement • Estimated development time ***and program size*** data entered in the Project Plan Summary • Actual time spent planning entered in the Time Recording Log

Table 17: CSP0.1 Development Script

Phase Number	**Purpose**	**To guide the development of small programs**
	Entry criteria	• Requirements statement • Project Plan Summary with planning completed • Time and Defect Recording Logs with planning completed • ***Coding Standard***.
Note: The terms ***driver*** and ***non-driver*** are used below. The driver is the partner who has control of the recording medium (ex: paper, computer keyboard) and is recording the design or implementing code or fixing code. The non-driver is the other partner who is actively observing the driver -- identifying defects, giving suggestions, etc. *When a partner is working alone, he or she is considered the driver, and no one is filling the non-driver role.*		
1	Design	• Review the requirements and produce a design to meet them via discussions between partners. • The driver records the design in pre-determined format/on pre-determined medium. • The non-driver observes to ensure the design is being recorded efficiently and effectively meets the requirements. The non-driver identifies defects and gives suggestions for alternative designs. • Periodically, switch drivers. • Record design time in the Time Recording Log
2	Code	• Implement the design ***following the Coding Standard***. • The driver implements the design by typing code via the keyboard. • The non-driver is observes to ensure the code properly implements the design, ***and conforms to the Coding Standard,*** identifying defects whenever necessary and giving suggestions for alternative implementations. • Periodically, switch drivers. • Record any requirements or design defects in the Defect Recording Log • Record coding time in the Time Recording Log
3	Compile	• Compile the program until error-free. • Both partners identify and discuss all defects found and the possible implications of these defects elsewhere in the code. • The driver implements the code changes by fixing code via the keyboard.

		• The non-driver observes to ensure the fix is properly implemented, identifying erroneous fix implementations. • Periodically, switch drivers. • Record all defects found in Defect Recording Log • Record compile time (until program compiles error-free) in the Time Recording Log
4	Test	• Test until all tests cases run <u>without error</u> • Both partners identify and discuss all defects found and the possible implications of these defects elsewhere in the code. • The driver implements the code changes by fixing code via the keyboard. • The non-driver observes to ensure the fix is properly implemented, identifying erroneous fix implementations. • Periodically, switch drivers. • Record defects in the Defect Recording Log • Record test time (until all test cases run error-free) in the Time Recording Log
	Exit Criteria	• A thoroughly tested program ***that conforms to the Coding Standard*** • Completed Defect Recording Log • Completed Time Recording Log

Table 18: CSP0.1 Postmortem Script

Phase Number	**Purpose**	**To guide the CSP postmortem process**
	Entry criteria	• Problem description and requirements statement. • Project Plan Summary with ***planned program size and*** planned development time. • Completed Time Recording Log • Completed Defect Recording Log • A tested and running program ***that conforms to the Coding Standard***
1	Defects Injected	• Determine from the Defect Recording Log the number of defects injected in each phase. • Enter this number under Defects Injected -- Actual on the Project Plan Summary
2	Defects Removed	• Determine from the Defect Recording Log the number of defects removed in each phase. • Enter this number under Defects Removed -- Actual on the Project Plan Summary
3	***Size***	• ***Count the LOC in the completed program.*** • ***Determine the base, reused, deleted, modified, added, total, total new and changed, and new reused LOC*** • ***Enter these data on the Project Plan Summary.***
4	Time	• Review the completed Time Recording Log • Enter the total time spent in each phase under Actual on the Project Plan Summary
	Exit Criteria	• A fully tested program ***that conforms to the Coding Standard*** • Completed Project Plan Summary Form • ***Completed PIP form describing process problems, improvement suggestions, and what went well.*** • Completed Defect Recording Log and Time Recording Log

Table 19: CSP0.1 and CSP 1.0 Project Plan Summary

Student	_____	Date	_____
Program	_____	Program #	_____
Instructor	_____	Language	_____

Program Size (LOC)	**Plan**	**Actual**	**To Date**
Base(B)		_____	
Deleted(D)		_____	
Modified(M)		_____	
Added(A) (T - B + D - R)		_____	
Reused(R)		_____	
Total New and Changed(N) (A + M)	_____	_____	_____
Total LOC (T)		_____	_____
Total New Reused		_____	_____

Time in Phase (min.)	**Plan**	**Total Actual**	**To Date %**	**Individual**	**Collaborative**
Planning	_____	_____	_____	_____	_____
Design	_____	_____	_____	_____	_____
Code	_____	_____	_____	_____	_____
Compile	_____	_____	_____	_____	_____
Test	_____	_____	_____	_____	_____
Postmortem	_____	_____	_____	_____	_____
Total	_____	_____	_____	_____	_____

Defects Injected	**Actual**	**Individual**	**Collaborative**
Planning	_____	_____	_____
Design	_____	_____	_____
Code	_____	_____	_____
Compile	_____	_____	_____
Test	_____	_____	_____
Total	_____	_____	_____

Defects Removed	**Actual**	**Individual**	**Collaborative**
Planning	_____	_____	_____
Design	_____	_____	_____
Code	_____	_____	_____
Compile	_____	_____	_____
Test	_____	_____	_____
Total	_____	_____	_____

Table 20: CSP0.1 and CSP 1.0 Project Plan Summary Instructions

Purpose	**This form holds the estimated and actual project data in a convenient and readily retrievable form**
Header	Enter the following: • Your name and today's date. • The program name and number. • The instructor's name. • The language you used to write the program.
Program Size (LOC)	***Prior to Development:*** • ***If you are modifying or enhancing an existing program, count that program's LOC and enter it under Base – Actual*** • ***Using your best judgment, estimate the new and changed LOC you expect to develop*** ***After Development:*** • ***If the base LOC (B) has changed, enter the new value*** • ***Measure the total program size and enter it under Total LOC (T) – Actual*** • ***Review your source code and determine the actual LOC that were deleted (D), modified (M), or reused (R). Enter these in the appropriate Actual row.*** • ***Calculate the LOC of added code as A = T – B + D – R*** • ***Calculate the total new and changed LOC as N = A + M.***
Time in Phase	• Under Plan, enter your original estimate of the total development time and ***the time required by phase***. • Under Actual, enter the total actual time in minutes spent in each development phase (should be the sum of the Individual and Collaborative time). • Under Individual, enter the total actual time spent by any partner individually • Under Collaborative, enter the total actual time spent the partners collaboratively • Under To Date %, enter the percentage of Total time (versus your plan) in each phase.
Defects Injected	• Under Actual, enter the number of defects injected in each phase (should be the sum of the Individual and Collaborative defects). • Under Individual, enter the total defects time injected by phase when a partner was working individually • Under Individual, enter the total defects time injected by phase when the partners were working collaboratively
Defects Removed	• Under Actual, enter the number of defects removed in each phase (should be the sum of the Individual and Collaborative defects).

	• Under Individual, enter the total defects time removed by phase when a partner was working individually • Under Individual, enter the total defects time removed by phase when the partners were working collaboratively

Table 21: Process Improvement Proposal (PIP)

Student _____ Date _____
Instructor _____ Program # _____
Process CSP Level

PIP Number	Problem Description and Proposed Solution

Describe what worked about your process during this program.

Notes and Comments

Table 22: Process Improvement Proposal (PIP) Instructions

Purpose	• To provide a way to record process problems and improvement ideas • To provide an orderly record of your process improvement ideas for use in later process improvement • To provide an orderly record of what you found beneficial during this program cycle so these things can be continued, as appropriate, in future cycles
General	Use the PIP form as follows: • To record process improvement ideas as they occur to you • To record beneficial process steps as you complete them • To record lessons learned and unusual conditions Keep PIP forms on hand while using the CSP • Record process problems even without proposed solutions. • Retain the PIPs for use in future process improvements
Header	• Enter your name, the date, the instructor's name, and the program number or other project designation. • Enter the name of the process you are using (such as CSP0.1)
Problem and Proposed Solution	Number the problems in each form in the left hand column. Start with number 1 on each PIP. Describe the problem as clearly as possible: • The difficulty encountered • The impact on the product, the process, and you. • Describe the proposed process improvement as explicitly as possible. • Where possible, reference the specific process element and the words or entries to be changed. • If you feel a proposed improvement is particularly important, describe its priority and explain why.
What Worked	Briefly describe process steps that proved beneficial to the project outcome. Be sure to describe when a process step prescribed in the CSP was altered, which yielded beneficial results.
Notes and Comments	For each product, complete at least one PIP form with overall comments about the process: • Record the process lessons learned. • Note any conditions you need to remember to later determine why the process worked particularly well or poorly.

Table 23: C++ Coding Standard

Purpose	To guide the development of C++ programs

Program Headers	//
	**********
	*
	// Program Assignment: the Program Number
	// Name(s): your names
	// Date: the date the program development STARTED
	// Description: a short description of the program function
	// **********
	**

Identifiers	Use descriptive names for all variables, function names, constants, and other identifiers. Avoid abbreviations or single-letter variables

Identifier Example	int number_of_students; // This is GOOD
	float x4, j, ftave; // These are BAD

Comments	• Document the code so that the reader can understand its operation. The more self-documenting your code is via meaningful variable names and proper spacing, the less comments will be needed to understand its purpose • Comments should explain both the purpose and behavior of the code, particularly at the beginning of function declarations in the header file. • Comment variable declarations to indicate their purpose.
Good Comment	if (record_count) > limit) // have all the records been processed?
Useless Comment	if (record_count) > limit) // check if record_count greater than limit
Blank Spaces	• Write programs with sufficient spacing so that they do not appear crowded. • Separate every program construct with at least one space.
Indenting	• Indent every level of brace from the previous one. • Open and close braces should be on lines by themselves and aligned with each other.

Indenting Example	while (miss_distance > threshold)
	{
	success_code = move_robot(target_location);
	if (success_code == MOVE_FAILED)
	{
	cout << "The robot move has failed." << endl;
	}

	}
Capitalization	• Capitalize all #define's • Lowercase all other identifers and reserved words. • Messages being output to the user can be mixed-case as to make a clean user presentation.
Capitalization Example	#define DEFAULT-NUMBER-OF-STUDENTS 15 int class_size = DEFAULT-NUMBER-OF-STUDENTS
Class Declaration and Definition	• All class data members must be private. • Public "getter" and "setter" accessor methods should be implemented to allow access to private data members, as appropriate.
Instance Variables	• There is a unique copy of each ariable for each instance of a class. Since every instance has its own copy, append "my" to the variable name. • Variable names should indicate the purpose of the variable. • The first letter of every word (except "my") is capitalized. • Examples: myAccount, myLastName, myMiddleInitial
Static/Class Variables	• There is one copy of static variables that is shared by all instances of a class. Append "our" to the beginning of each static variable name to remind yourself every time they are used that they are shared. • Variable names should indicate the purpose of the variable. • The first letter of every word (except "our") is capitalized. • Examples: ourNumberOfInstances, ourTotalCost, ourFileHandle

Class Declaration Example	class Student
	{
	public:
	  string getFirstName();
	  void setFirstName(string name);
	  string getLastName();
	  void setLastName(string name);
	  int getNumStudents();
	  void setNumStudents(int number);
	private:
	  string myFirstName;
	  string myLastName;
	  static int ourNumStudents;
	};

Table 24: CSP1.0 Process Script

Phase Number	Purpose	To guide you in collaboratively developing module-level programs
	Entry criteria	• Problem description • Empty Project Plan Summary form • Empty Time and Defect Recording Logs • Stop watch (optional)
1	Planning	• Produce or obtain a requirement statement • ***Analyze the requirements statement via the development of a thorough set of use cases.*** • Estimate the total new and changed LOC required • Estimate the required development time of both partners • Enter the plan data in the Project Plan Summary form • Record the time spent in the Time Recording Log for Planning
2	Development	• ***Perform a CRC card exercise in order to develop a preliminary high-level design.*** • Design the program ***Perform the steps below iteratively:*** • Implement the design • Compile the program and fix and log all defects found • Test the program and fix and log all defects found • Record the time spent in these activities in the Time Recording Log in the appropriate phase
3	Postmortem	• Complete the Project Plan Summary form with actual time, defect, and size data
	Exit Criteria	• A thoroughly tested program • Completed Project Plan Summary with estimated and actual data • Completed Process Improvement Proposal (PIP) form • Completed Defect and Time Recording Logs

Table 25: CSP1.0 and CSP 1.1 Planning Script

Phase Number	**Purpose**	**To guide the CSP planning process**
	Entry criteria	• Problem description • Empty Project Plan Summary form • Empty Time Recording Log
1	Program Requirements	• Produce or obtain a requirements statement for the program • Ensure the requirements statement is clear and unambiguous • ***Analyze the program requirements by producing a comprehensive set of Use Cases for the set of requirements. Complete the Use Case Flow-of-Events template for each use case.*** • Resolve any questions
2	Size Estimate	• Make your best estimate of the total new and changed LOC required to develop this program
3	Resource Estimate	• Make your best estimate of the time (for both partners) required to develop this program • Make your best estimate of the total new and changed LOC required to develop this program
	Exit Criteria	• A documented requirements statement • ***Completed Use Case Flow-of-Events templates for each use case.*** • Estimated development time and program size data entered in the Project Plan Summary • Actual time spent planning entered in the Time Recording Log

Table 26: CSP1.0 Development Script

Phase Number	**Purpose**	**To guide the development of small programs**
	Entry criteria	• Requirements statement • Project Plan Summary with planning completed • Time and Defect Recording Logs with planning completed • Coding Standard.
Note: The terms ***driver*** and ***non-driver*** are used below. The driver is the partner who has control of the recording medium (ex: paper, computer keyboard) and is recording the design or implementing code or fixing code. The non-driver is the other partner who is actively observing the driver -- identifying defects, giving suggestions, etc. *When a partner is working alone, he or she is considered the driver, and no one is filling the non-driver role.*		
1	Design	• Review the requirements and • Produce a design to meet the requirements by ***performing a CRC card exercise with partners and/or members of the product team.*** • ***Include in your design a class diagram that lists the properties and methods of each class and demonstrates which other classes the class is dependent upon for services/information.*** • The driver records the design in pre-determined format/on pre-determined medium. • The non-driver observes to ensure the design is being recorded efficiently and effectively meets the requirements. The non-driver identifies defects and gives suggestions for alternative designs. • Periodically, switch drivers. • Record design time in the Time Recording Log
Perform Code, Compile and Test (below) iteratively. Choose an element of the design and code it, compile it and test it before choosing another element of the design to implement.		
2	Code	• Implement the design following the Coding Standard. • The driver implements the design by typing code via the keyboard. • The non-driver is observes to ensure the code properly implements the design, and conforms to the Coding Standard, identifying defects whenever necessary and giving suggestions for alternative implementations. • Periodically, switch drivers. • Record any requirements or design defects in the De-

		fect Recording Log • Record coding time in the Time Recording Log
3	Compile	• Compile the program until error-free. • Both partners identify and discuss all defects found and the possible implications of these defects elsewhere in the code. • The driver implements the code changes by fixing code via the keyboard. • The non-driver observes to ensure the fix is properly implemented, identifying erroneous fix implementations. • Periodically, switch drivers. • Record all defects found in Defect Recording Log • Record compile time (until program compiles error-free) in the Time Recording Log
4	Test	• Test until all tests cases run without error • Both partners identify and discuss all defects found and the possible implications of these defects elsewhere in the code. • The driver implements the code changes by fixing code via the keyboard. • The non-driver observes to ensure the fix is properly implemented, identifying erroneous fix implementations. • Periodically, switch drivers. • Record defects in the Defect Recording Log • Record test time (until all test cases run error-free) in the Time Recording Log
	Exit Criteria	• A thoroughly tested program that conforms to the Coding Standard • Completed Defect Recording Log • Completed Time Recording Log

Table 27: CSP1.0 Postmortem Script

Phase Number	**Purpose**	**To guide the CSP postmortem process**
	Entry criteria	• Problem description and requirements statement. • Project Plan Summary with planned program size and planned development time. • Completed Time Recording Log • Completed Defect Recording Log • A tested and running program that conforms to the Coding Standard
1	Defects Injected	• Determine from the Defect Recording Log the number of defects injected in each phase. • Enter this number under Defects Injected -- Actual on the Project Plan Summary
2	Defects Removed	• Determine from the Defect Recording Log the number of defects removed in each phase. • Enter this number under Defects Removed -- Actual on the Project Plan Summary
3	Size	• Count the LOC in the completed program. • Determine the base, reused, deleted, modified, added, total, total new and changed, and new reused LOC • Enter these data on the Project Plan Summary.
4	Time	• Review the completed Time Recording Log • Enter the total time spent in each phase under Actual on the Project Plan Summary
	Exit Criteria	• A fully tested program that conforms to the Coding Standard • ***Completed Use Case Flow-of-Events templates*** • Completed Project Plan Summary Form • Completed PIP form describing process problems, improvement suggestions, and what went well. • Completed Defect Recording Log and Time Recording Log

Table 28: Use Case Flow of Event Template

Student _____ Date _____
Instructor Program #

X	**Flow of Events for the <name> Use Case**
X.1	**Preconditions**
X.2	**Main Flow**
X.3	**Sub-flows (if applicable)**
X.4	**Alternative Flows (if applicable)**

Table 29: Use Case Flow of Events Template Instructions

Purpose	• To systematically develop a description of the events needed to accomplish the required behavior of the use case. • This flow of events should enumerate what the system **should do**, not how the system should do it. • The use case should be written in the language of the domain so that it can be easily read by a customer.
Header	• Enter your name, the date, the instructor's name, and the program number or other project designation.
Flow of Events Number and Name	• Assign each use case has its own number, starting with number 1. Place this number in the Flow of Events everywhere an X appears in the template. • Assign each use case a short name, which is indicative of the purpose of the use case. Example: **1.0** Flow of Events for the **Customer Transaction** Use Case
Preconditions	• Enumerate any data that is needed by the use case • Enumerate any flows or subflows that must execute in another use case before this use case can begin.
Main Flow	• Enumerate the normal sequence of events or the basic start-to-finish path an actor will follow under normal conditions.
Sub-flows	• Further describe/breakdown the sequence of events in the Main Flow. • Only break the Main Flow into Sub-flows if the complexity warrants the breakdown. Resist temptation to develop pseudocode of the implementation in the subflow.
Alternative Flows	• Enumerate infrequently used paths through a scenario, exceptions, and error conditions.

Table 30: CSP1.1 Process Script

Phase Number	Purpose	To guide you in collaboratively developing module-level programs
	Entry criteria	• Problem description • Empty Project Plan Summary form • Empty Time and Defect Recording Logs • Stop watch (optional)
1	Planning	• Produce or obtain a requirement statement • Analyze the requirements statement via the development of a thorough set of use cases. • Estimate the total new and changed LOC required • Estimate the required development time of both partners • Enter the plan data in the Project Plan Summary form • Record the time spent in the Time Recording Log for Planning
2	Development	• Perform a CRC card exercise in order to develop a preliminary high-level design. • Design the program • ***Review the design and fix and log all defects found.*** Perform the steps below iteratively: • Implement the design • ***Review the code and fix and log all defects found.*** • Compile the program and fix and log all defects found • Test the program and fix and log all defects found • Record the time spent in these activities in the Time Recording Log in the appropriate phase
3	Postmortem	• Complete the Project Plan Summary form with actual time, defect, and size data
	Exit Criteria	• A thoroughly tested program • Completed Project Plan Summary with estimated and actual data • ***Completed Design Review and Code Review Checklists.*** • Completed Process Improvement Proposal (PIP) form • Completed Defect and Time Recording Logs

Table 31: CSP1.1 and CSP2.0 Development Script

Phase Number	**Purpose**	**To guide the development of small programs**
	Entry criteria	• Requirements statement • Project Plan Summary with planning completed • Time and Defect Recording Logs with planning completed • Coding Standard.
Note: The terms ***driver*** and ***non-driver*** are used below. The driver is the partner who has control of the recording medium (ex: paper, computer keyboard) and is recording the design or implementing code or fixing code. The non-driver is the other partner who is actively observing the driver -- identifying defects, giving suggestions, etc. *When a partner is working alone, he or she is considered the driver, and no one is filling the non-driver role.*		
1	Design	• Review the requirements and • Produce a design to meet the requirements by performing a CRC card exercise with partners and/or members of the product team. • Include in your design a class diagram that lists the properties and methods of each class and demonstrates which other classes the class is dependent upon for services/information. • The driver records the design in pre-determined format/on pre-determined medium. • The non-driver observes to ensure the design is being recorded efficiently and effectively meets the requirements. The non-driver identifies defects and gives suggestions for alternative designs. • Periodically, switch drivers. • Record design time in the Time Recording Log
2	***Design Review***	• ***Follow the Design Review Checklist and review the design.*** • ***Fix all defects found.*** • ***Record defects in Defect Recording Log*** • ***Record Design Review time in Time Recording Log***
3	***Prepare Test Cases***	• ***Prepare a preliminary set of test cases using the Test Case Template. The test case should validate that all requirements have been properly implemented and possible error conditions have been properly handled. (Details that are not yet know can be completed after code development.) Use the Unit Test Checklist to ensure test coverage.***

		• ***Fix any design defects surfaced by writing the test cases. Record these defects in the Defect Recording Log.*** • ***Record test development time as Testing time in the Time Recording Log.***
Perform Code, Compile and Test (below) iteratively. Choose an element of the design and code it, compile it and test it before choosing another element of the design to implement.		
4	Code	• Implement the design following the Coding Standard. • The driver implements the design by typing code via the keyboard. • The non-driver is observes to ensure the code properly implements the design, and conforms to the Coding Standard, identifying defects whenever necessary and giving suggestions for alternative implementations. • Periodically, switch drivers. • Record any requirements or design defects in the Defect Recording Log • Record coding time in the Time Recording Log
5	***Code Review***	• ***Using the Code Review Checklist and review the code.*** • ***Fix all defects found.*** • ***Record defects in the Defect Recording Log*** • ***Record Code Review time in Time Recording Log.***
6	Compile	• Compile the program <u>until error-free.</u> • Both partners identify and discuss all defects found and the possible implications of these defects elsewhere in the code. • The driver implements the code changes by fixing code via the keyboard. • The non-driver observes to ensure the fix is properly implemented, identifying erroneous fix implementations. • Periodically, switch drivers. • Record all defects found in Defect Recording Log • Record compile time (until program compiles error-free) in the Time Recording Log
7	Test	• ***Develop additional test cases using the Test Case Template. Complete any additional, new information on previously developed test cases. Use the Unit Test Checklist to ensure test coverage.*** • ***Add new test cases to an ever-enlarging set of regression tests.*** • Test until all tests cases **(including all regression tests)** run <u>without error</u>

		• ***Record the results running test case on the Test Case Template***. • Both partners identify and discuss all defects found and the possible implications of these defects elsewhere in the code. • The driver implements the code changes by fixing code via the keyboard. • The non-driver observes to ensure the fix is properly implemented, identifying erroneous fix implementations. • Periodically, switch drivers. • Record defects in the Defect Recording Log • Record test time (until all test cases run error-free) in the Time Recording Log
	Exit Criteria	• A thoroughly tested program that conforms to the Coding Standard • ***Completed Design Review and Code Review Checklists.*** • ***Completed Unit Test Checklists*** • Completed Defect Recording Log • Completed Time Recording Log

Table 32: CSP1.1 and CSP2.0 Postmortem Script

Phase Number	**Purpose**	**To guide the CSP postmortem process**
	Entry criteria	• Problem description and requirements statement • Project Plan Summary with planned program size and planned development time • ***Completed Design Review and Code Review Check-lists*** • ***Completed Unit Test Checklists*** • Completed Time Recording Log • Completed Defect Recording Log • A tested and running program that conforms to the Coding Standard
1	Defects In-jected	• Determine from the Defect Recording Log the number of defects injected in each phase. • Enter this number under Defects Injected -- Actual on the Project Plan Summary
2	Defects Re-moved	• Determine from the Defect Recording Log the number of defects removed in each phase. • Enter this number under Defects Removed -- Actual on the Project Plan Summary
3	Size	• Count the LOC in the completed program. • Determine the base, reused, deleted, modified, added, total, total new and changed, and new reused LOC • Enter these data on the Project Plan Summary.
4	Time	• Review the completed Time Recording Log • Enter the total time spent in each phase under Actual on the Project Plan Summary
	Exit Criteria	• A fully tested program that conforms to the Coding Standard • Completed Use Case Flow-of-Events templates • Completed Project Plan Summary Form • Completed PIP form describing process problems, im-provement suggestions, and what went well. • Completed Defect Recording Log and Time Recording Log

Table 33: CSP1.1 Project Plan Summary

Student	_____	Date	_____
Program	_____	Program #	_____
Instructor	_____	Language	_____

Summary	**This Program**	**Programs to Date**
Defects/ KLOC	_____	_____
Yield %	_____	_____
% Appraisal COQ	_____	_____
% Failure COQ	_____	_____
COQ A/F Ratio	_____	_____

Program Size (LOC)	**Plan**	**Actual**	**To Date**
Base(B)		_____	
Deleted(D)		_____	
Modified(M)		_____	
Added(A) **(T - B + D - R)**		_____	
Reused(R)		_____	
Total New and Changed(N) **(A + M)**	_____	_____	_____
Total LOC (T)		_____	_____
Total New Reused		_____	_____

Time in Phase (min.)	**Plan**	**Total Actual**	**To Date %**	**Individual**	**Collaborative**
Planning	_____	_____	_____	_____	_____
Design	_____	_____	_____	_____	_____
Code	_____	_____	_____	_____	_____
Compile	_____	_____	_____	_____	_____
Test	_____	_____	_____	_____	_____
Postmortem	_____	_____	_____	_____	_____
Total	_____	_____	_____	_____	_____

Defects Injected	**Actual**	**Individual**	**Collaborative**
Planning			
Design			
Code			
Compile			
Test			
Total			

Defects Removed	**Actual**	**Individual**	**Collaborative**
Planning			
Design			
Code			
Compile			
Test			
Total			

Defect Removal Efficiency (Defects/Hour)	**This Program**	**Programs to Date**
Design Review		
Code Review		
Compile		
Test		

Defect Removal Leverage (vs Test)		
Design Review		
Code Review		
Compile		

Table 34: CSP1.1 Project Plan Summary Instructions

Purpose	**This form holds the estimated and actual project data in a convenient and readily retrievable form**
Header	Enter the following: • Your name and today's date • The program name and number • The instructor's name • The language you used to write the program
Summary	• ***Enter the actual and to date defect data*** • ***Enter the actual and to date yield*** • ***Enter the actual and to date Appraisal Cost of Quality: the percentage of development time spent in compile and test*** • ***Enter the actual and to date Failure Cost Of Quality: the percentage of development time spent in compile and test*** • ***Enter the A/F Ratio: the ratio of Appraisal COQ divided by Failure COQ***
Program Size (LOC)	Prior to Development: • If you are modifying or enhancing an existing program, count that program's LOC and enter it under Base – Actual • Using your best judgment, estimate the new and changed LOC you expect to develop After Development: • If the base LOC (B) has changed, enter the new value • Measure the total program size and enter it under Total LOC (T) – Actual • Review your source code and determine the actual LOC that were deleted (D), modified (M), or reused (R). Enter these in the appropriate Actual row. • Calculate the LOC of added code as A = T – B + D – R • Calculate the total new and changed LOC as N = A + M.
Time in Phase	• Under Plan, enter your original estimate of the total development time and the time required by phase. • Under Actual, enter the total actual time in minutes spent in each development phase (should be the sum of the Individual and Collaborative time). • Under Individual, enter the total actual time spent by any partner individually • Under Collaborative, enter the total actual time spent the partners collaboratively • Under To Date %, enter the percentage of Total time (versus your plan) in each phase.

Defects Injected	• Under Actual, enter the number of defects injected in each phase (should be the sum of the Individual and Collaborative defects). • Under Individual, enter the total defects time injected by phase when a partner was working individually • Under Individual, enter the total defects time injected by phase when the partners were working collaboratively
Defects Removed	• Under Actual, enter the number of defects removed in each phase (should be the sum of the Individual and Collaborative defects). • Under Individual, enter the total defects time removed by phase when a partner was working individually • Under Individual, enter the total defects time removed by phase when the partners were working collaboratively
Defect Removal Efficiency	• ***Under This Program, enter the actual efficiencies (defects/hour) achieved for this program*** • ***Under Programs To Date, enter the actual efficiencies (defects/hour) achieved for all programs to date***
Defect Removal Leverage	• ***Under This Program, enter the actual leverage (defects/hour of this phase divided by defects/hour test) achieved for this program*** • ***Under Programs To Date, enter the actual leverage (defects/hour of this phase divided by defects/hour test) achieved for all programs to date***

Table 35: Individual Code Review Checklist

Purpose	To guide you in conducting an effective code review
General	• As you complete each review step, check off that item. • Complete the checklist for one program unit before you start to the review the next.
Complete	Verify that the code is a complete and correct implementation of the design.
Standards	Ensure the code conforms to the C++ coding standards
Include	Verify that all includes are complete
Line-by-line check	Check every line of code for: • instruction syntax • proper punctuation
Initialization	Check variable and parameter initialization: • At program initiation • At start of every loop • At function entry
Calls	Check function call formats: • Pointers • Parameters • Arrays • Use of &
{} pairs	Ensure the {} are proper and matched
Logic Operators	• Verify the proper use of ==, =, ‖ and so on • Check every logic test for proper ( )
Classes and Functions	• Class declarations end with ; • Ensure all functions are defined before they are used or properly defined in a .h file • The scope resolution operator :: is used properly in class functions definitions.
Names	Check name spelling and use: • Is it consistent? • Is it within the declared scope? • Do all structures and classes use '.' or '->' references properly?
Pointers	Check that: • pointers are initialized to NULL • pointers are declared only after new • new pointers are always deleted after use
Output Format	Check the output format: • Line stepping is proper • Spacing is proper
File Open and	Verify that all files are:

Close	• properly declared • properly opened • properly closed

Table 36: Collaborative Code Review Checklist

Purpose	To guide you in conducting an effective code review
General	• As you complete each review step, check off that item. • Complete the checklist for one program unit before you start to the review the next.
Complete	Verify that the code is a complete and correct implementation of the design.
Standards	Ensure the code conforms to the C++ coding standards

Table 37: Individual Design Review Checklist

Purpose	To guide you through an effective Design Review
General	• As you complete each review step, check off that item • Complete the checklist for one program unit before you start to review the next
Completeness	Ensure that the requirements and specifications are completely and correctly covered by the design: • All specified outputs are produced • All needed inputs are furnished • All required includes are stated
Class Design	• All data members are private with public getters/setters where necessary and prudent • Data Connectedness: Can you traverse the network of collaborations between the classes to gather all the information you need to deliver the services based on a representative set of scenarios? • Abstraction: Does the name of each class convey its abstractions? Does the abstraction have a natural meaning and use in the domain? • Responsibility Alignment: Do the name, main responsibility statement data and functions in each class align?
Logic	• All program sequences are in the proper order • Recursion unwinds properly and terminates • All loops are properly initiated, incremented and terminated
Modularity	• Ensure the proper use of functions to modularize program steps. • Ease of reading • Repetitive steps
Special Cases	Check all special cases: • Ensure proper operation with empty, full, minimum, maximum, negative, and zero values for all variables • Protect against out-of-limits, overflow, underflow conditions • Ensure “impossible” conditions are absolutely impossible • Handle all incorrect input conditions

Table 38: Collaborative Design Review Checklist

Purpose	To guide you through an effective Design Review
General	• As you complete each review step, check off that item • Complete the checklist for one program unit before you start to review the next
Completeness	Ensure that the requirements and specifications are completely and correctly covered by the design: • All specified outputs are produced • All needed inputs are furnished • All required includes are stated
Class Design	• All data members are private with public getters/setters where necessary and prudent • Data Connectedness: Can you traverse the network of collaborations between the classes to gather all the information you need to deliver the services based on a representative set of scenarios? • Abstraction: Does the name of each class convey its abstractions? Does the abstraction have a natural meaning and use in the domain? • Responsibility Alignment: Do the name, main responsibility statement data and functions in each class align?
Special Cases	Check all special cases: • Ensure proper operation with empty, full, minimum, maximum, negative, and zero values for all variables • Protect against out-of-limits, overflow, underflow conditions • Ensure “impossible” conditions are absolutely impossible • Handle all incorrect input conditions

Table 39: Test Case Template

Student _____ Date _____
Instructor Program #

Test Case Number	Test Objective	Test Description	Expected Results	Actual Results

Table 40: Test Case Template Instructions

Purpose	• To systematically document test cases necessary to thoroughly validate the desired behaviors of the program
Header	• Enter your name, the date, the instructor's name, and the program number or other project designation.
Test Number	Identify each test case with a unique number.
Test Objective	• Briefly describe the objective for running the test case • Example: linear regression with normal input
Test Description	• Describe each test's data and procedures in sufficient detail to permit it to be run or re-run by someone other than yourself • At this time, it is not acceptable to write, "run linear regression with normal input." You must indicate specific function calls or data that should be input into the program to force the conditions you want to test.
Expected Results	• List the <u>exact</u> results the test should produce if it runs properly.

Actual Results	• List the results that were actually produced when the test is run.
	• When the same test is run multiple times while fixing multiple defects, note the results of each test.
	• Example:
	  o Run 1:
	  o Run 2:
	  o Run 3:

Table 41: Test Coverage Checklist

Purpose	To guide you in reviewing the completeness of your test cases
Black Box Testing	
Complete	Does each requirement have it's own test case?

Equivalence Class Partitioning	Have you developed an equivalence class representing the set of valid or invalid input conditions for each test case:
	• If the input for the test case:
	  • can be a range of values, try one valid input value and two different invalid input values
	  • must be a specific value, try the valid input value and two different invalid input values
	  • must be any of a set of values, try one valid value and one invalid value
	  • is a boolean, try both true and false

Boundary Value Analysis	Have you performed boundary analysis on the input conditions for each test case If the input for the test case: • can be a range of values from a to b, try a, b, a-1, and b+1 (if integers -- otherwise slightly less than a an slightly more than b) • must be any of a set of values, test with the min of the set, the max of the set, the min-1 and the max+1 • is a boolean, try both true and false
Scenario Testing	Do you have test cases that run through a representative set of customer scenarios?
Data	Do you have test cases that check for the wrong kind of data -- for example a negative price?
White Box Testing	

Basis Path Testing	Has each line of code been executed with at least one test case?
	• Draw the flowgraph of a module.
	• Compute the minimum number of tests necessary to exercise each line of code by calculating the cyclomatic complexity V(G) using any one of the formulas below
	  o V(G) = the number of regions in the graph OR
	  o V(G) = E - N + 2 (where E= number of edges and N = number of nodes) OR
	  o V(G) = P + 1 (where P = number of predicate nodes)
	Ensure test cases are written to execute each line of code

Loop Testing	Where n is the maximum number of allowable passes through the loop, write test cases to: • Skip the loop entirely • Make only one pass through the loop • Make two passes through the loop • Make m passes through the loop, where m < n • Make n-1, n, and n+1 passes through the loop
File Interface	Have you checked for: • proper file attributes • opening and closing files • eof handling
Error Handling	Have you tried error handling routines? • Are error descriptions meaningful? • Do error descriptions match the error conditions? • Are there any spelling mistakes in messages?
Object Oriented Testing	If you have a class hierarchy, have you tested each of the inherited methods in the context of each inherited class?
Misc	• Have you exercised any possible underflow or overflow conditions? • Has a list of common errors been used to write test cases to detect errors that have occurred frequently in the past • Do the test cases make hand checks easy?

Table 42: CSP2.0 Process Script

Phase Number	**Purpose**	**To guide you in collaboratively developing module-level programs**
	Entry criteria	• Problem description • Empty Project Plan Summary form • ***Empty PROBE Size Estimating Template***. • ***Historical estimate and actual size and time data.*** • Empty Time and Defect Recording Logs • Stop watch (optional)
1	Planning	• Produce or obtain a requirement statement • Analyze the requirements statement via the development of a thorough set of use cases. • ***Use the PROBE method to*** estimate the total new and changed LOC required • ***Use the PROBE method to*** estimate the required development time of both partners • Enter the plan data in the Project Plan Summary form • Record the time spent in the Time Recording Log for Planning
2	Development	• Perform a CRC card exercise in order to develop a preliminary high-level design. • Design the program • Review the design and fix and log all defects found. Perform the steps below iteratively: • Implement the design • Review the code and fix and log all defects found. • Compile the program and fix and log all defects found • Test the program and fix and log all defects found • Record the time spent in these activities in the Time Recording Log in the appropriate phase
3	Postmortem	• Complete the Project Plan Summary form with actual time, defect, and size data
	Exit Criteria	• A thoroughly tested program • Completed Project Plan Summary with estimated and actual data • ***Completed PROBE worksheet.*** • Completed Design Review and Code Review Checklists. • Completed Process Improvement Proposal (PIP) form • Completed Defect and Time Recording Logs

Table 43: CSP2.0 Planning Script

Phase Number	**Purpose**	**To guide the CSP planning process**
	Entry criteria	• Problem description • Empty Project Plan Summary form • Empty PROBE Size Estimating Template • Historical estimated and actual size and resource data • Empty Time Recording Log
1	Program Requirements	• Produce or obtain a requirements statement for the program • Ensure the requirements statement is clear and unambiguous • ***Analyze the program requirements by producing a comprehensive set of Use Cases for the set of requirements. Complete the Use Case Flow-of-Events template for each use case.*** • Resolve any questions
2	Size Estimate	• ***Produce a program conceptual design***. • ***Use the PROBE method to*** estimate of the total new and changed LOC required to develop this program • ***Estimate the base, added, deleted, modified, and reused LOC*** • ***Complete the Size Estimating Template and the Project Plan Summary***
3	Resource Estimate	• ***Based on the time required per LOC on previous programs,*** estimate of the time (for both partners) required to develop this program • Make your best estimate of the total new and changed LOC required to develop this program
	Exit Criteria	• A documented requirements statement • Completed Use Case Flow-of-Events templates for each use case. • ***The program conceptual design*** • ***Completed Size Estimating Template*** • Estimated development time and program size data entered in the Project Plan Summary • Actual time spent planning entered in the Time Recording Log

Table 44: CSP2.0 and CSP2.1 Project Plan Summary

Student	_____	Date	_____
Program	_____	Program #	_____
Instructor	_____	Language	_____

Summary	**This Program**	**Programs to Date**
LOC/Hour	_____	_____
Planned Time	_____	_____
Actual Time	_____	_____
CPI (Cost-Performance Index)	_____	_____
Defects/ KLOC	_____	_____
Yield %	_____	_____
% Appraisal COQ	_____	_____
% Failure COQ	_____	_____
COQ A/F Ratio	_____	_____

Program Size (LOC)	**Plan**	**Actual**	**To Date**
Base(B)		_____	
Deleted(D)		_____	
Modified(M)		_____	
Added(A) (T - B + D - R)		_____	
Reused(R)		_____	_____
Total New and Changed(N) (A + M)	_____	_____	_____
Total LOC (T)		_____	_____
Total New Reused		_____	_____

Time in Phase (min.)	**Plan**	**Total Actual**	**To Date %**	**Individual**	**Collaborative**
Planning	_____	_____	_____	_____	

	_____	_____	_____	_____	_____
Design	_____	_____	_____	_____	_____
Code	_____	_____	_____	_____	_____
Compile	_____	_____	_____	_____	_____
Test	_____	_____	_____	_____	_____
Postmortem	_____	_____	_____	_____	_____
Total	_____	_____	_____	_____	_____

Defects Injected	**Actual**	**Individual**	**Collaborative**
Planning	_____	_____	_____
Design	_____	_____	_____
Code	_____	_____	_____
Compile	_____	_____	_____
Test	_____	_____	_____
Total	_____	_____	_____

Defects Removed	**Actual**	**Individual**	**Collaborative**
Planning	_____	_____	_____
Design	_____	_____	_____
Code	_____	_____	_____
Compile	_____	_____	_____
Test	_____	_____	_____
Total	_____	_____	_____

Defect Removal Efficiency (Defects/Hour)	**This Program**	**Programs to Date**
Design Review	_____	_____
Code Review	_____	_____
Compile	_____	_____
Test	_____	_____

Defect Removal Leverage (vs Test)		
Design Review	_____	_____

Code Review		
Compile		

Table 45: CSP2.0 and CSP2.1 Project Plan Summary Instructions

Purpose	**This form holds the estimated and actual project data in a convenient and readily retrievable form**
Header	Enter the following: • Your name and today's date • The program name and number • The instructor's name • The language you used to write the program
Summary	• ***Enter the new and changed LOC per hour for this program and for all programs developed to date.*** • Enter the actual and to date defect data • Enter the actual and to date yield • Enter the actual and to date Appraisal Cost of Quality: the percentage of development time spent in compile and test • Enter the actual and to date Failure Cost Of Quality: the percentage of development time spent in compile and test • Enter the A/F Ratio: the ratio of Appraisal COQ divided by Failure COQ
Program Size (LOC)	Prior to Development: • If you are modifying or enhancing an existing program, count that program's LOC and enter it under Base – Actual • ***From the Size Estimating Template, enter estimated object LOC (E) under plan.*** • ***Enter the estimated new and changed LOC (N) from the Size Estimating Template.*** • ***Estimate the numbers of deleted (D) and reused (R) LOC and combine with the measured base (B) LOC so that*** $T = N + B - M - D + R$ After Development: • If the base LOC (B) has changed, enter the new value • Measure the total program size and enter it under Total LOC (T) – Actual • Review your source code and determine the actual LOC that were deleted (D), modified (M), or reused (R). Enter these in the appropriate Actual row. • Calculate the LOC of added code as $A = T - B + D - R$ • Calculate the total new and changed LOC as $N = A + M$.
Time in Phase	• ***Under Plan, enter estimated total time from the Size Estimating Template and time by phase.*** • Under Actual, enter the total actual time in minutes spent in each development phase (should be the sum of the Individual and Col-

	laborative time). • Under Individual, enter the total actual time spent by any partner individually • Under Collaborative, enter the total actual time spent the partners collaboratively • Under To Date %, enter the percentage of Total time (versus your plan) in each phase.
Defects Injected	• Under Actual, enter the number of defects injected in each phase (should be the sum of the Individual and Collaborative defects). • Under Individual, enter the total defects time injected by phase when a partner was working individually • Under Individual, enter the total defects time injected by phase when the partners were working collaboratively
Defects Removed	• Under Actual, enter the number of defects removed in each phase (should be the sum of the Individual and Collaborative defects). • Under Individual, enter the total defects time removed by phase when a partner was working individually • Under Individual, enter the total defects time removed by phase when the partners were working collaboratively
Defect Removal Efficiency	• Under This Program, enter the actual efficiencies (defects/hour) achieved for this program • Under Programs To Date, enter the actual efficiencies (defects/hour) achieved for all programs to date
Defect Removal Leverage	• Under This Program, enter the actual leverage (defects/hour of this phase divided by defects/hour test) achieved for this program • Under Programs To Date, enter the actual leverage (defects/hour of this phase divided by defects/hour test) achieved for all programs to date

Table 46: CSP2.1 Process Script

Phase Number	Purpose	To guide you in collaboratively developing module-level programs
	Entry criteria	• Problem description • Empty Project Plan Summary form • Empty PROBE Size Estimating Template**.** • Historical estimate and actual size and time data. • Empty Time and Defect Recording Logs • Stop watch (optional)
1	Planning	• Produce or obtain a requirement statement • Analyze the requirements statement via the development of a thorough set of use cases. • Use the PROBE method to estimate the total new and changed LOC required • Use the PROBE method to estimate the required development time of both partners • ***Complete a Task Planning Template.*** • ***Complete a Schedule Planning Template.*** • Enter the plan data in the Project Plan Summary form • Record the time spent in the Time Recording Log for Planning
2	Development	• Perform a CRC card exercise in order to develop a preliminary high-level design. • Design the program • Review the design and fix and log all defects found. Perform the steps below iteratively: • Implement the design • Review the code and fix and log all defects found. • Compile the program and fix and log all defects found • Test the program and fix and log all defects found • Record the time spent in these activities in the Time Recording Log in the appropriate phase
3	Postmortem	• Complete the Project Plan Summary form with actual time, defect, and size data
	Exit Criteria	• A thoroughly tested program • Completed Project Plan Summary with estimated and actual data • Completed PROBE worksheet. • Completed Design Review and Code Review Checklists.

		• Completed Process Improvement Proposal (PIP) form • Completed Defect and Time Recording Logs

Table 47: CSP2.1 Planning Script

Phase Number	**Purpose**	**To guide the CSP planning process**
	Entry criteria	• Problem description • Empty Project Plan Summary form • Empty PROBE Size Estimating, ***Task Planning, and Schedule Planning*** Template • Historical estimated and actual size and resource data • Empty Time Recording Log
1	Program Requirements	• Produce or obtain a requirements statement for the program • Ensure the requirements statement is clear and unambiguous • Analyze the program requirements by producing a comprehensive set of Use Cases for the set of requirements. Complete the Use Case Flow-of-Events template for each use case. • Resolve any questions
2	Size Estimate	• Produce a program conceptual design. • Use the PROBE method to estimate of the total new and changed LOC required to develop this program • Estimate the base, added, deleted, modified, and reused LOC • Complete the Size Estimating Template and the Project Plan Summary
3	Resource Estimate	• Based on the time required per LOC on previous programs, estimate of the time (for both partners) required to develop this program • Make your best estimate of the total new and changed LOC required to develop this program
4	***Task and Schedule Planning***	• ***For projects requiring several days or more of work, complete the Task Planning and Schedule Planning Templates.***
	Exit Criteria	• A documented requirements statement • Completed Use Case Flow-of-Events templates for each use case. • The program conceptual design • Completed Size Estimating Template • ***For projects requiring several days or more of work, complete the Task Planning and Schedule Planning***

		Templates • Estimated development time and program size data entered in the Project Plan Summary • Actual time spent planning entered in the Time Recording Log

Table 48: CSP2.1 Development Script

Phase Number	**Purpose**	**To guide the development of small programs**
	Entry criteria	• Requirements statement • Project Plan Summary with planning completed • ***For projects of several days' duration or more, completed Task Planning and Schedule Planning Templates*** • Time and Defect Recording Logs with planning completed • Coding Standard.
Note: The terms ***driver*** and ***non-driver*** are used below. The driver is the partner who has control of the recording medium (ex: paper, computer keyboard) and is recording the design or implementing code or fixing code. The non-driver is the other partner who is actively observing the driver -- identifying defects, giving suggestions, etc. *When a partner is working alone, he or she is considered the driver, and no one is filling the non-driver role.*		
1	Design	• Review the requirements and • Produce a design to meet the requirements by performing a CRC card exercise with partners and/or members of the product team. • Include in your design a class diagram that lists the properties and methods of each class and demonstrates which other classes the class is dependent upon for services/information. • The driver records the design in pre-determined format/on pre-determined medium. • The non-driver observes to ensure the design is being recorded efficiently and effectively meets the requirements. The non-driver identifies defects and gives suggestions for alternative designs. • Periodically, switch drivers. • Record design time in the Time Recording Log
2	Design Review	• Follow the Design Review Checklist and review the design. • Fix all defects found. • Record defects in Defect Recording Log • Record Design Review time in Time Recording Log
3	Prepare Test Cases	• Prepare a preliminary set of test cases using the Test Case Template. The test case should validate that all requirements have been properly implemented and pos-

		sible error conditions have been properly handled. (Details that are not yet know can be completed after code development.) Use the Unit Test Checklist to ensure test coverage. • Fix any design defects surfaced by writing the test cases. Record these defects in the Defect Recording Log. • Record test development time as Testing time in the Time Recording Log.
Perform Code, Compile and Test (below) iteratively. Choose an element of the design and code it, compile it and test it before choosing another element of the design to implement.		
4	Code	• Implement the design following the Coding Standard. • The driver implements the design by typing code via the keyboard. • The non-driver is observes to ensure the code properly implements the design, and conforms to the Coding Standard, identifying defects whenever necessary and giving suggestions for alternative implementations. • Periodically, switch drivers. • Record any requirements or design defects in the Defect Recording Log • Record coding time in the Time Recording Log
5	Code Review	• Using the Code Review Checklist and review the code. • Fix all defects found. • Record defects in the Defect Recording Log • Record Code Review time in Time Recording Log.
6	Compile	• Compile the program until error-free. • Both partners identify and discuss all defects found and the possible implications of these defects elsewhere in the code. • The driver implements the code changes by fixing code via the keyboard. • The non-driver observes to ensure the fix is properly implemented, identifying erroneous fix implementations. • Periodically, switch drivers. • Record all defects found in Defect Recording Log • Record compile time (until program compiles error-free) in the Time Recording Log
7	Test	• Develop additional test cases using the Test Case Template. Complete any additional, new information on previously developed test cases. Use the Unit Test Checklist to ensure test coverage.

		• Add new test cases to an ever-enlarging set of regression tests. • Test until all tests cases (including all regression tests) run <u>without error</u> • Record the results running test case on the Test Case Template. • Both partners identify and discuss all defects found and the possible implications of these defects elsewhere in the code. • The driver implements the code changes by fixing code via the keyboard. • The non-driver observes to ensure the fix is properly implemented, identifying erroneous fix implementations. • Periodically, switch drivers. • Record defects in the Defect Recording Log • Record test time (until all test cases run error-free) in the Time Recording Log
	Exit Criteria	• A thoroughly tested program that conforms to the Coding Standard • Completed Design Review and Code Review Checklists. • Completed Unit Test Checklists • Completed Defect Recording Log • Completed Time Recording Log

Table 49: CSP2.1 Postmortem Script

Phase Number	**Purpose**	**To guide the CSP postmortem process**
	Entry criteria	• Problem description and requirements statement • Project Plan Summary with planned program size and planned development time • ***For projects of several days' duration or more, completed Task Planning and Schedule Planning Templates*** • Completed Design Review and Code Review Checklists • Completed Unit Test Checklists • Completed Time Recording Log • Completed Defect Recording Log • A tested and running program that conforms to the Coding Standard
1	Defects Injected	• Determine from the Defect Recording Log the number of defects injected in each phase. • Enter this number under Defects Injected -- Actual on the Project Plan Summary
2	Defects Removed	• Determine from the Defect Recording Log the number of defects removed in each phase. • Enter this number under Defects Removed -- Actual on the Project Plan Summary
3	Size	• Count the LOC in the completed program. • Determine the base, reused, deleted, modified, added, total, total new and changed, and new reused LOC • Enter these data on the Project Plan Summary.
4	Time	• Review the completed Time Recording Log • Enter the total time spent in each phase under Actual on the Project Plan Summary
	Exit Criteria	• A fully tested program that conforms to the Coding Standard • Completed Use Case Flow-of-Events templates • Completed Project Plan Summary Form • Completed PIP form describing process problems, improvement suggestions, and what went well. • Completed Defect Recording Log and Time Recording Log

APPENDIX C

USE CASE/FLOW OF EVENTS EXAMPLE

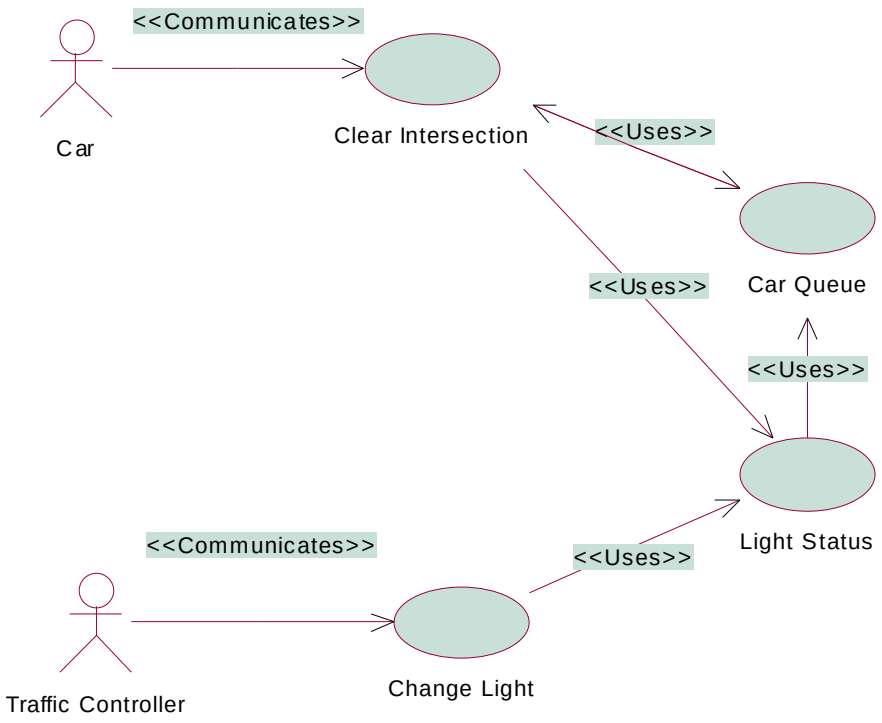
Program Specifications:

This program is a simulation of automobile traffic flow/traffic signals at a typical intersection. Traffic flows in both directions on each of the cross streets. Cars form into eight different queues at the intersection:

Abbreviation	Traffic Flow
N	from the north, headed straight south
NL	from the north, headed east (left at intersection)
E	from the east, headed straight west
EL	from the east, headed south (left at intersection)
S	from the south, headed straight north
SL	from the south, headed west (left at intersection)
W	from the west, headed straight east
WL	from the west, headed north (left at intersection)

- When a car enters the intersection, if the queue there is empty and the light is green, they can clear the intersection. Else, they join the appropriate queue
- When the system is initiated, the traffic signal allows traffic to flow from NL and SL. Next it allows traffic to flow from N and S. Then, it allows traffic flow from EL and WL. Lastly, it allows traffic to flow from E and W -- then starts again with NL and SL and so forth.
- When a signal light changes to green it can allows cars to pass through the intersection or get out of the queue.

UML Use Case Diagram:



Use Case Flow of Events:

1	**Flow of Events for the Clear Intersection Use Case**
1.1	**Preconditions**
	The Initialize sub-flow of the Change Light use case, the Initialize sub-flow of the Light Status use case, and the Initialize sub-flow of the Car Queue use case must be executed before this use case begins.
1.2	**Main Flow**
	This use cases begins when a car enters the intersection. The car checks its status (S-1). The use case ends when the car clears the intersections (S-4).
1.3	**Sub-flows**
	S-1: Check Status Car checks status (S-2, S-3). If the light is green and the queue is empty, the car clears the intersection (S-4). Otherwise, it joins a queue (S-5).
	S-2: Check Light Execute the Report Status sub-flow of the Light Status use case. Send a message indicating if the light is green or red.
	S-3: Check Queue Execute the Report Status sub-flow of the Car Queue use case. Send a message indicating if the queue is empty or not.
	S-4: Go The car clears the intersection and the use case ends.
	S-5: Join a Queue Send a message to the Add to Queue sub-flow of the Car Queue use case.
1.4	**Alternative Flows**
	None.

2	**Flow of Events for the Change Light Use Case**
2.1	**Preconditions**
	None.
2.2	**Main Flow**
	The traffic lights are initialized (S-1). The lights change (S-2) when the Traffic Light Controller (actor) advances the lights.
2.3	**Sub-flows**
	S-1: Initialize The traffic lights are initialized with all lights red except NL and SL. NL and SL are green.
	S-2: Advance Lights Lights are advanced in the following order: When the system is initialized, the traffic signal allows traffic to flow from NL and SL. Next, it allows traffic to flow from N and S. Then it allows traffic flow from EL and WL. Lastly, it allows traffic to flow from E and W – then starts again with NL and SL, and so forth. When a light is changed (from green to red or from red to green), a message is sent to the Update Status sub-flow of the Light Status use case.
2.4	**Alternative Flows**
	None.

3	**Flow of Events for the Light Status Use Case**
3.1	**Preconditions**
	None.
3.2	**Main Flow**
	Update (S-1) and report (S-2) the status of a traffic light color.
3.3	**Sub-flows**
	S-1: Update status. Change the color of the lights. If the light is turned to green, send a message to the Release from Queue subflow of the Car Queue use case.
	S-2: Report Status Send a message indicating the color of the traffic light.
3.4	**Alternative Flows**
	None.

4	**Flow of Events for the Car Queue Use Case**
4.1	**Preconditions**
	None.
4.2	**Main Flow**
	This use case begins by initializing the queue (S-1). Cars may be added to the queue (S-2) or released from the queue (S-3).
4.3	**Sub-flows**
	S-1: Initialize The queue is initialized with zero cars.
	S-2: Add to queue Receive a message from the Join a Queue sub-flow of the Clear Intersection use case. Add car to queue.
	S-3: Release from queue Release cars from queue. Send a message to the Go sub-flow of the Clear Intersection use case.
4.4	**Alternative Flows**
	None.

APPENDIX D

PAIR PROGRAMMING QUESTIONNAIRE

43 Respondents

DEMOGRAPHICS:

How long have you been a programmer in industry/research?

7%	Less than 1 year
17%	1 - 5 years
76%	More than 5 years

How long have you been with your current employer?

30%	Less than 1 year
45%	1- 5 years
25%	More than 5 years

How long have you been pair programming?

46%	Less than 1 year
23%	1 - 2 years
31%	More than 2 years

WORKING ALONE?

When pair programming, do you believe the two programmers should EVER work separately?

74%	Yes
26%	No

If yes,

When do you like to work separately (check all that apply)

3%	During design
16%	When thinking about a tough problem
19%	Tackling a new domain or language issue
50%	During experimental prototyping
59%	Partner's sick or busy
53%	Other (please explain below)

COMMENT:

- Experiment with a new approach and prove it to yourself before showing to partner
- Doing simple, well-defined, rote programming (like wiring entry fields to the GUI)
- Thinking about deep-concentration, logical problems
- Adding to test cases or refactoring test-only code
- Architectural thoughts, for me, are best done alone, at night, in bed. And pairing doesn't work well when making documents. Otherwise, pair all the time.

After working independently, when you get back together with your partner what do you do with the work that was done independently? (choose one)

22% Scrap and re-write
66% Review and incorporate it
6% Incorporate (no review)
6% Other (please comment)

What's the maximum amount of time a pair should be able to work independently and still be considered "pair programming"?

21% 0-10% of the time
12% 10-20% of the time
21% 20-30% of the time
12% 30-40% of the time
15% 40-50% of the time
7% 50-70% of the time
12% 70-80% of the time
0% 80-100% of the time

ROLE OF PERSON NOT TYPING

What's the role of the person not typing?

93% Perform continuous code review
86% Perform continuous design review
12% Work on next increment/project
42% Think about next increment/project
16% Look out the window/anything
26% Other (please comment below)

COMMENTS:

- Stop the other person from deviating from the process/from the assigned task
- Provide strategic viewpoint
- Reminders of method names or to test

- Ask "what the heck is that??" / ask general questions about what the typist is doing
- Perform continuous analysis review "Is that really what they're asking for?"
- Sometimes teach the other person (if you're more experienced in something)
- The person not typing must be as active and engaged and the person typing
- Peer-social effects: the act of having a (friendly) person nearby puts the typing programmer into more of a 'team' mode – the typist is more likely to behave as expected rather than as she/her personally desires
- Take notes about thoughts, goals, and TO DO lists
- Think about what is being put in and see how it fits into the design
- Pull up useful documentation to help with current work.
- Suggest alternatives

REVIEWS

When pair programming, do you do a design review?

67% Yes, we review the design while we create the design
7% Yes, we review once we are done with the design
5% No
21% Other (please comment below)

COMMENTS:

- Sometimes review with technical stakeholders
- CRC Cards with others
- When someone feel particularly uncomfortable with a certain part.
- "While we are creating and also at the end. This is the power of pair programming."
- Design reviews and code reviews are continuous. The design sits next to the pair.
- We try to hold design reviews off until refactoring.

If you do a design review, do you use a pre-defined design checklist?

9% Yes
91% No

When pair programming, do you do a code review?

63% Yes, we review the code while we create the code
5% Yes, we review together once we are done with the code, but before we compile
2% Yes, we review together once we are done with the code, but after we compile

2% Yes, we review with a larger development group
9% No
19% Other (please comment below)

COMMENTS:

- "I answer no because pair programming is, by its nature, code review as it happens. It's not a separate process. It actually is better than a code review."
- "If you code together, you automatically have a very simple form of code review."
- Review with architect
- “We regard reviews as a way to establish project standards. It’s less Your code is not to spec, fix it” and more “Here’s a neat thing we did. We think others should do the same thing.”

If you do a code review, do you use a pre-defined code review checklist?

21% Yes
79% No

MISC

Rank how strongly you agree with these statements.
(SA = Strongly agree; A = Agree; D = Disagree; SD = Strongly disagree)

These factors are critical for my success in pair programming:

	SA	**A**	**D**	**SD**
The physical layout of our work-space allows us to both see the screen and to share the keyboard.	58%	38%	2%	2%
Management support.	35%	44%	16%	5%
My partner must buy in to the pair programming concept.	65%	26%	9%	
My partner must be able to practice ego-less programming.	44%	40%	16%	

Rank how strongly you agree with these statements.
(SA = Strongly agree; A = Agree; D = Disagree; SD = Strongly disagree)

	SA	**A**	**D**	**SD**
When I pair program, I am more confident in our solution than I am when I program	69%	27%	4%	

alone.				
When I pair program, I enjoy my job more than when I program alone.	47%	47%	4%	2%

JUST DIDN'T WORK OUT

Was there ever someone with whom you simply couldn't pair program? If so, please comment on why below.

- Person took any comments as mistrust
- Person with large ego/always thought he was right
- Person always agrees (there needs to be some disagreement)
- His/her way or the highway
- Person with great insecurity or anxiety about their skills
- Overly introverted
- You need to be able to trust the other person's judgement

Are there physical or environmental conditions under which pair programming did not work for you? If so, please comment.

- Sitting too far away from own phone/email
- Computer in the corner
- Too much noise
- Open cubicles make it difficult to talk without disturbing others
- 21” monitors are very helpful; LCD projectors are even better
- “Doesn’t really work using NetMeeting. It seems you really need to be physically present.”
- Remotely via dial-up

GENERAL COMMENTS

- More productive, more demanding, more fun, you move much faster
- "The best thing about Pair Programming for me is the continuous discussion gave me training in formulating the thoughts I have about design and programming, thereby helped me reflect over them and made me a better designer/programmer.
- You must be compatible with the other person
- Will not work if neither partner is experienced or a believer in pair programming
- Not easily initially / takes time to incorporate
- "I strongly feel pair programming is the primary reason our team has been successful. It has given us a very high level of code quality (almost to the point of 0 defects). The only code we have ever had errors in was

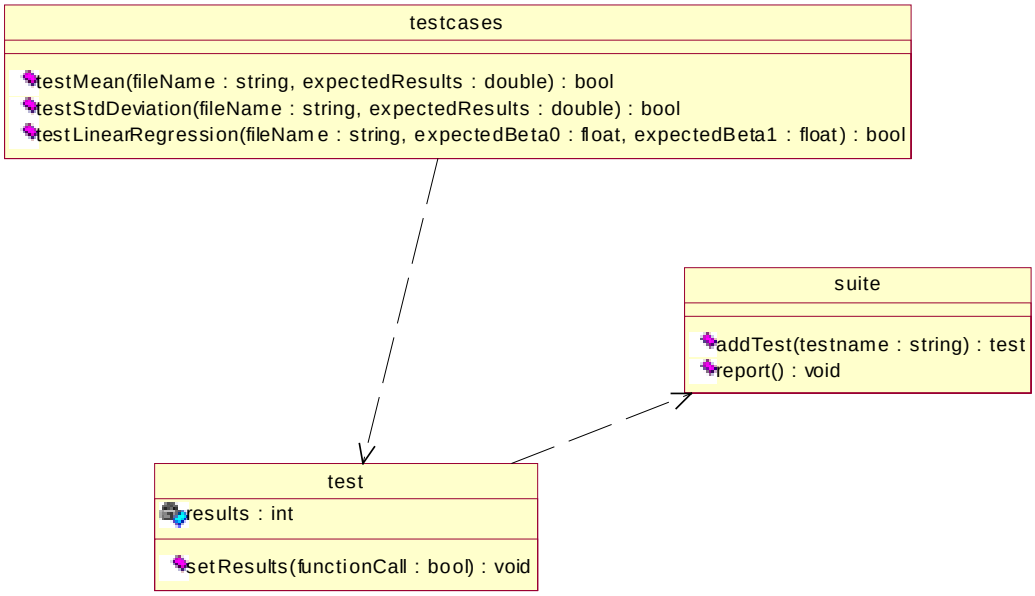
code that wasn't pair programmed . . . we should really question a situation where it isn't utilized."

- "It is a powerful technique as there are 2 brains concentrating on the same problem all the time. It forced one to concentrate fully on the problem at hand."
- Important for the less experienced to be given tasks to do him/herself so they can feel important
- Works well for domain knowledge transfer from one person to another as well as for transferring good programming practice
- "It takes more effort because the pace is forced by the other person all the time. Neither person feels they can slack off."
- “My highest marks, and shiniest projects as an undergraduate were produced with partners. I used to call it the Batman and Robin model, since I would work so much better regardless of the abilities of my partner. I have partnered with people who were much less locally knowledgeable, those who were pretty similar to me, and those who impressed me. In all cases, we got more done than we would have expected to get done alone.”
- “In my 20+ years in the industry, I know one thing for CERTAIN, pairing works! Better code, happier team, more productivity.”
- “You have to try it to believe it – but when you do, it’s very hard to go back.”

APPENDIX E

AUTOMATED REGRESSION TESTER

Sample Regression Tester Class Diagram:



The testcases class contains a library of test functions. Each function is sent parameters to force certain input conditions and to receive expected results. Each function returns a boolean value indicating whether the actual test obtained the expected results.

The test class has an integer attribute, which stores the status of the test case: Not Run, Pass or Fail. The setResults method is used to set the value of this attribute.

The suite class contains a collection of test class instances; test classes instances are added to the suite via the addTest method. The suite report method produces a summary report.

Below is a sample automated test case program.

```
int main() {

  Test* testcase;

  Suite s("Linear Regression");
  testcase = s.addTest("Normal Input");

  testcase.setResults(testLinearRegression("input1.txt", -22.55, 1.72));

   . . .

  s.report();

  return 0;

}
```

The Suite **report()** function will print the name of the test suite. Then it will go through all the test cases that have been added in the test suite and whether or not it has been passed. Lastly, it will print a summary. For example:

```
Suite:  Linear Regression
====================

Test:  Normal Input          Passed
Test:  Alpha Input           Failed
Test:  Three Numbers         Not Run

====================

1 Passed
1 Failed
1 Not Run
```

APPENDIX F

BREAKDOWN OF NPV INCENTIVE INTO LOWER-LEVEL METRICS

(Reprinted from [56], see next page for abbreviation definitions)
A: Test Strategy
B: Base Strategy

<table>
<tr><td colspan="7">NPVI
Net Present Value Incentive
$(NPV_A – NPV_B)/TPS = (PVI * NAV_B + DCI*I_B)/(NAV_B+I_B)$</td></tr>
<tr><td colspan="6">PVI
Present Value Incentive
$(PV_A – PV_B)/NAV_B = (e^{NAVA}/(1+d)^{\beta} – 1)$
where $\beta= T_B(1-1/e^{DTA})$</td><td rowspan="2">DCI
Development Cost Incentive
$(I_B – I_A)/I_B = 1-1/e^{DCA}$</td></tr>
<tr><td rowspan="5">DTA
Development Time Advantage
$\log T_B – \log T_A$</td><td colspan="5">NAVA
Net Asset Value Advantage
$\log NAV_A – \log NAV_B = \log (e^{AVA}C_B – M_B/e^{OCA}) – \log NAV_B$</td></tr>
<tr><td colspan="4">AVA
Asset Value Advantage
$\log C_A – \log C_B = EEA + PCA + QFA + TVA$</td><td rowspan="3">OCA
Operation Cost Advantage
$\log M_B – \log M_A$</td><td rowspan="3">DCA Development Cost Advantage
$\log I_B – \log I_A$</td></tr>
<tr><td>EEA
Early Entry Advantage
MEEA (1-1/(1 + DTA))</td><td>PCA
Product Cost Advantage</td><td>TVA
Termination Value Advantage</td><td>QFA Quality/ Functionality Advantage</td></tr>
<tr><td>MEEA
Max(EEA)</td><td></td><td></td><td></td></tr>
</table>

Development Time	T
Development Cost	I
(Future) Asset Value	C
Operation Cost	M
Product Risk	d
Total Project Scale (Estimated product revenue volume)	TPS
Product Cost Advantage (Relative contribution of direct product savings to the asset value of the test strategy)	PCA
Quality/Function Advantage (Relative contribution to the asset value of the test strategy of the ability to control the quality and functionality of the end system)	QFA
Termination Value Advantage (Relative contribution of termination value to the asset value of the test strategy. Termination value includes the value of reusable software salvaged upon project termination.)	TVA

REFERENCES

[1] P. B. Crosby, *Quality is Free: The Art of Making Quality Certain*. New York, New York: McGraw-Hill Book Company, 1979.

[2] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[3] W. W. Gibbs, "Software's Chronic Crisis," in *Scientific American*, pp. 86-95, Sept. 1994.

[4] S. L. Pfleeger, *Software Engineering: Theory and Practice*. Upper Saddle River, NJ: Prentice Hall, 1998.

[5] K. Beck, *Extreme Programming Explained: Embrace Change*. Reading, Massachusetts: Addison-Wesley, 2000.

[6] Wiki, "Programming In Pairs," in *Portland Pattern Repository*, http://c2.com/cgi/wiki?ProgrammingInPairs., 1999.

[7] J. T. Nosek, "The Case for Collaborative Programming," in *Communications of the ACM*, pp. 105-108, March 1998.

[8] R. L. Baber, "Comparison of Electrical "Engineering" of Heaviside's Times and Software "Engineering" of our Times," *IEEE Annals of the History of Computing*, vol. 19, pp. 5-17, 1997.

[9] W. E. Deming, *Out of the Crisis*. Cambridge, MA: MIT Press, 1986.

[10] J. M. Juran and F. M. Gryna, *Juran's Quality Control Handbook*, Fourth ed. New York, New York: McGraw-Hill Book Company, 1988.

[11] W. S. Humphrey, *A Discipline for Software Engineering*: Addison Wesley Longman, Inc, 1995.

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

[13] A. Anderson, Beattie, Ralph, Beck, Kent et al., "Chrysler Goes to "Extremes"," in *Distributed Computing*, pp. 24-28, Oct. 1998.

[14] Wiki, "Extreme Programming Roadmap," in *Portland Pattern Repository*, http://c2.com/cgi/wiki?ExtremeProgramming, 1999.

[15] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Reading, Massachusetts: Addison-Wesley, 1999.

[16] G. Salomon, *Distributed Cognitions: Psychological and educational considerations*. Cambridge: Cambridge University Press, 1993.

[17] N. V. Flor and E. L. Hutchins, "Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Main-

tenance," presented at Empirical Studies of Programmers: Fourth Workshop, 1991.

[18] J. O. Coplien, "A Development Process Generative Pattern Language," in *Pattern Languages of Program Design*, James O. Coplien and Douglas C. Schmidt, Ed. Reading, MA: Addison-Wesley, 1995, pp. 183-237.

[19] L. L. Constantine, *Constantine on Peopleware*. Englewood Cliffs, NJ: Yourdon Press, 1995.

[20] M. C. Paulk, B. Curtis, and M. B. Chrisis, "Capability Maturity Model for Software Version 1.1," Software Engineering Institute CMU/SEI-93-TR, February 24, 1993.

[21] W. Hayes and J. W. Over, "The Personal Software Process: An Empirical Study of the Impact of PSP on Individual Engineers," Software Engineering Institute, Pittsburgh, PA CMU/SEI-97-TR-001, December 1997.

[22] B. Meyer, *Object-Oriented Software Construction*, Second Edition ed. Upper Saddle River, New Jersey: Prentice Hall, 1997.

[23] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Wokingham, England: Addison-Wesley, 1992.

[24] D. Rosenberg and K. Scott, *Use Case Driven Object Modeling with UML: A Practical Approach*. Reading, Massachusetts: Addison-Wesley, 1999.

[25] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*. Upper Saddle River, NJ: Prentice Hall, 2000.

[26] T. Quatrani, *Visual Modeling with Rational Rose and UML*. Reading, Massachusetts: Addison Wesley, 1998.

[27] D. Bellin and S. S. Simone, *The CRC Card Book*. Reading, Massachusetts: Addison-Wesley, 1997.

[28] A. Cockburn, "Using CRC Cards," Humans and Technology TR.99.01, Salt Lake City, UT , March 11, 1999.

[29] M. E. Fagan, "Advances in software inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, pp. 182-211, 1976.

[30] G. W. Russell, "Experience with Inspection in Ultralarge-Scale Developments," *IEEE Software*, pp. 25-31, Jan. 1991.

[31] E. F. Weller, "Lessons from Three Years of Inspection Data," *IEEE Software*, pp. 38-45, Sept. 1993.

[32] A. Cockburn, "Object-Oriented Analysis and Design, Part 2," in *C/C++ Users Journal*, June 1998.

[33] R. E. Jeffries, "Extreme Testing," in *Software Testing and Quality Engineering*, vol. 1, 1999, pp. 22-27.

[34] W. S. Humphrey, *Introduction to the Team Software Process*. Reading, Massachusetts: Addison Wesley, 2000.

[35] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair-Programming," in *IEEE Software*, submitted for consideration.

[36] L. Williams and R. R. Kessler, “The Effects of "Pair-Pressure" and "Pair-Learning" on Software Engineering Education,” presented at Conference on Software Engineering Education and Training, Austin, TX, 2000.

[37] A. Cockburn and L. Williams, “The Costs and Benefits of Pair Programming,” presented at eXtreme Programming and Flexible Processes in Software Engineering -- XP2000, Cagliari, Sardinia, Italy, 2000.

[38] W. Bennis, Biederman, Patricia Ward, *Organizing Genius: The Secrets of Creative Collaboration*: Addison-Wesley Publishing Company, Inc., 1997.

[39] P. M. Johnson, “Reengineering Inspection: The Future of Formal Technical Review,” in *Communications of the ACM*, vol. 41, 1998, pp. 49-52.

[40] W. S. Humphrey, *Introduction to the Personal Software Process*: Addison-Wesley, 1997.

[41] G. M. Weinberg, *The Psychology of Computer Programming Silver Anniversary Edition*. New York: Dorset House Publishing, 1998.

[42] B. W. Kernighan and R. Pike, *The Practice of Programming*. Reading, Massachusetts: Addison-Wesley, 1999.

[43] C. Jones, *Software Quality: Analysis and Guidelines for Success*. Boston, MA: International Thomson Computer Press, 1997.

[44] J. Lave and E. Wenger, *Situated Learning: Legitimate peripheral participation*. New York, NY: Cambridge University Press, 1991.

[45] W. Cunningham and K. Auer, *Extreme Programing Applied: Playing to Win!*: Addison Wesley, in preparation.

[46] F. P. J. Brooks, *The Mythical Man-Moth*: Addison-Wesley Publishing Company, 1975.

[47] A. H. Maslow, *Motivation and Personality*. New York, NY: Harper and Bros., 1954.

[48] L. Williams and R. Kessler, “All I Ever Needed to Know About Pair Programming I Learned in Kindergarten,” in *Communications of the ACM*, May 2000.

[49] T. DeMarco and T. Lister, *Peopleware*. New York: Dorset House Publishers, 1977.

[50] Wiki, “Pair Programming Facilities,” in *Portland Pattern Repository*, http://c2.com/cgi/wiki?PairProgrammingFacilities, 1999.

[51] L. S. Levy, *Taming the Tiger: Software Engineering and Software Economics*. New York: Springer-Verlag, 1987.

[52] S. Tockey, “A Missing Link in Software Engineering,” in *IEEE Software*, pp. 31-36, November/December 1997.

[53] C. F. Kemerer, “Progress, Obstacles, and Opportunities in Software Engineering Economics,” in *Communications of the ACM*, vol. 41, 1998, pp. 63-66.

[54] S. A. Slaughter, D. E. Harter, and M. S. Drishnan, “Evaluating the Cost of Software Quality,” in *Communications of the ACM*, vol. 41, 1998, pp. 67-73.

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

[56] H. Erdogmus, “Comparative evaluation of software development strategies based on Net Present Value,” presented at International Conference on Software

Engineering Workshop on Economics-Driven Software Engineering, California, 1999.

[57] S. A. Ross, *Fundamentals of Corporate Finance*: Irwin/McGraw-Hill, 1996.

[58] H. Erdogmus and J. Vandergraaf, "Quantitative Approaches for Assessing the Value of COTS-centric Development," presented at Sixth International Symposium on Software Metrics, Boca Raton, FL, 1999.

[59] N. Gross, M. Stepanek, O. Port, and J. Carey, "Software Hell," in *Business Week*, pp. 104-118, Dec. 6, 1999.

[60] J. Favaro and S. L. Pfleeger, "Making software development investment decisions," *Software Engineering Notes*, vol. 23, pp. 69-74, 1998.

[61] D. Webb, Humphrey, Watts, "Using the TSP on the TaskView Project," *Crosstalk*, February 1999.

[62] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowgray, *AntiPatterns*. New York: Wiley Computer Publishing, 1998.

[63] D. Keirsey, "The Keirsey Temperament Sorter II," http://www.keirsey.com, 2000.

[64] D. Keirsey, *Please Understand Me II*. Del Mar, CA: Prometheus Nemesis Book Company, 1998.

[65] C. Drew, Hardman, Michael L. and Hart, Ann Weaver, *Designing and Conducting Research: Inquiry in Education and Social Science*. Needham Heights, Massachusetts: Simon and Schuster Company, 1996.

[66] A. H. Dutoit, Bruegge, Bernd, "Communication Metrics for Software Development," *IEEE Transactions on Software Engineering*, pp. 615-628, Aug. 1998.