5-2003

Pair Programming to Facilitate the Training of Newly-Hired Programmers

Mark Anthony Poff

Pair Programming to Facilitate the Training of Newly-Hired Programmers

by

Mark Anthony Poff

A thesis submitted to
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Software Engineering

Melbourne, Florida
May 2003

We the undersigned committee
hereby approve the attached thesis

Pair Programming to Facilitate the Training of Newly-Hired Programmers

by
Mark Antony Poff

David W. Clay, M.A., M.S. Assistant Professor Computer Science Principal Advisor	Arthur F. Dickinson, Ph.D. Assistant Director of Computer Programs Committee Member
John F. Clark, Ph.D., P.E. Professor Director, Graduate Studies Space Systems Committee Member	William D. Shoaff, Ph.D. Assistant Professor / Head Computer Science

Abstract

Pair Programming to Facilitate the Training

of Newly-Hired Programmers

by

Mark Anthony Poff

Principal Advisor: David W. Clay, M.A., M.S.

Pair programming is touted by some software professionals as a style which allows developers to produce superior code in less time, and with fewer defects, than code produced by individuals. However, the acclaim for the pair methodology is not universal; some see it as a waste of effort which produces marginal improvements. Reported experiments which obtain quantitative results have typically been performed in an educational environment, and may not reflect actual workplace conditions. This thesis reports on an experiment using pair programming in an industrial setting. Its goal is to determine if this programming style can be used to increase the technical and environmental knowledge of newly-hired programmers, and verify the claims stated above.

Table of Contents

List of Figures

List of Acronyms and Abbreviations

ANSI:	American National Standards Institute
App SW:	Payload Checkout Application Software Group
FEP:	Front-End Processor
GUI:	Graphical User Interface
ISS:	International Space Station
KSC:	Kennedy Space Center
MDM:	Multiplexer-Demultiplexer
MPLM:	MultiPurpose Pressurized Logistics Module
NASA:	National Aeronautics and Space Administration
PP:	Pair Programming
ProgA:	Programmer “A”
ProgB:	Programmer “B”
PUI:	Program-Unique Identifier
PSP:	Personal Software Process
SRD:	Software Requirements Document
SSPF:	Space Station Processing Facility
TCMS:	Test Control and Monitor System
UML:	Unified Modeling Language
XP:	Extreme Programming

Acknowledgements

- My thesis Advisor, Professor David Clay, for his advice, guidance, and suggestions in my graduate studies.

- My two co-workers, anonymous in this thesis, who agreed to be the “guinea pigs” for the pair programming experiment. More cooperative subjects could not have been found.

- The Boeing Company, Kennedy Space Center Division, for its support in this pursuit.

Chapter 1: Introduction

1.1 Problem Description

Software development organizations are staffed to meet anticipated need. When planned projects exceed the current work force capability, additional programmers are hired - hopefully far enough in advance to be trained and ready to be productive when the time comes. If the new programmers are not hired with enough lead time to allow sufficient training, they may be of little use when their contributions are required. Thus, identification of effective learning curve accelerator strategies is important.

At the Kennedy Space Center, a group exists which produces application software for the pre-flight testing of space shuttle payloads. Many of these payloads require custom checkout software to verify their functionality prior to launch. Pre-flight checkout of the payloads is critical - once in orbit, little can be done if they do not perform as designed.

Working at KSC is a unique experience - the terminology used, the environment, and the checkout software design are foreign to incoming experienced programmers, and even more so to neophytes. Historically, most newcomers to the application software group require a year of acclimation and training before they can be independently productive. This lead time is higher for inexperienced programmers. The group is staffed based on the anticipated shuttle payload flight schedule. Hiring cannot be done until money is allocated, and money is not allocated until a payload is officially placed on an upcoming flight. Unfortunately, the certainty of the shuttle flight schedule rarely exceeds one year.

Therein lies the problem - the need for programming staff cannot be projected much more than a year in advance, and new-hire training requires at least that long. Budget allocations encourage hiring lower paid - i.e., inexperienced - programmers who require even more instruction.

1.2 Attempted Solutions to the Problem

The application software group has, over the past years, attempted a variety of ways to reduce the acclimatization time for newcomers to the organization. A primary area of instruction involves the hardware for which checkout applications are written. It is believed that the better the programmer understands the payload systems, the better the programmer will understand the purpose of an application being designed to test those systems. The result will be software which fully meets payload checkout requirements. Initial attempts to instill this knowledge had new-hires attending the same training sessions as did system engineers. This proved to be of minimal use, as the programmers didn't have the hardware or electrical background to understand much of what was presented in these sessions. Simplified training was developed which gave an elementary education on payload systems. These sessions are of greater benefit, but still seem to do little to increase early productivity. It is apparent that lack of environmental knowledge is not the only issue.

Another focus is on the skills of the programmers themselves. Many of the newly hired individuals come to the organization directly from college. It had commonly been assumed that someone with an impressive G.P.A. would have no problem mastering the ANSI C code and associated GUIs developed by the group. This has been found to be false. Individuals might have taken a wide range of computer-related college courses, but still have problems understanding the group's software development methodology. Familiarity with programming languages does not necessarily mean new programmers can produce a good software design. Additional formal training is not an option - not only is the training budget non-existent, but no courses exist which describe the unique framework of payload checkout applications.

Mentoring is the most common training method - assigning experienced programmers to train inexperienced ones. This works to a degree, but is practiced unevenly. Senior programmers are often too busy for the effort involved, and some people are simply not very effective at mentoring. Some programmers resent being mentors, feeling it isn't part of their job description. As a result, some new programmers receive needed assistance, while others do not.

1.3 A New Approach

A possible improvement to the understanding and experience bottleneck incorporates pair programming into the training process. In the pair programming style, two individuals work closely together on a single software artifact. This differs from teaming and/or mentoring in that each member of the pair is very involved in, and knowledgeable of, what the other is doing at all times.

The majority of literature on pair programming lauds it as having superior benefits when compared to individual programming, primarily in the quality of the software artifacts produced. The man-hours required for programming in pairs is not reported to be significantly greater than that required for programming individually, especially if the artifacts produced contain fewer defects and thus require less rework. Further, individuals of *any* skill mix will supposedly provide superior products to equally talented (or untalented) programmers working alone.

However, software professionals disagree when it comes to the pairing of novice programmers. Some feel the learning curves of the novices will be significantly decreased, and they will produce surprisingly good code. Others feel the effort will be wasted; a case of “the blind leading the blind”, and the outcome will be well-written bad software. A number of factors might influence this range of positive and negative opinions. The primary objective of this thesis will be to research the benefits of pairing newly-hired programmers.

1.3.1 An Experiment in Pair Programming

The experiment to be performed will involve individuals with little professional software development experience. Two programmers will be selected to create the GUIs for a software application, including the imbedded GUI dynamics (code that drives the user interface objects). The pair will be provided with the GUI requirements, and will be responsible for developing an appropriate number of user interface displays to meet those requirements.

The individuals will be given instruction on the techniques of pair programming and asked to work collaboratively for some portion of the development. The time spent in pair programming will be at least 33% of the time, and as much as 100% of the time - this will be a choice left to the pair. When the programmers are not working together, they will work on the project individually. The pair will review all individual work during the next collaboration.

Mentoring and instruction on GUI development will be provided at a level commensurate with that historically given to solo programmers of a similar skill level. Additional mentoring and instruction on the best practices of pair programming will be provided to the two on a regular basis.

Prior to beginning the experiment, estimates will be made (based on historical group data) as to the time required to produce the assigned software, and the number of defects that will be uncovered during pre-release test. Two sets of estimates will be made: one for a solo programmer, and another for pair programmers. The programmers will gather data to allow a comparison of the estimates to the actual result. To accomplish this, the pair will be directed to keep metric data during the life of this experiment, including:

- Time spent on artifact production as a pair.
- Time spent on artifact production alone.
- Coding / design errors detected during GUI functional test.

Interviews will be conducted with the pair on a regular basis (approximately every week) to capture their impressions of the pair programming experiment. This will provide a qualitative assessment of the experience, with an emphasis on skill and knowledge development.

It is theorized that the outcome of the experiment will demonstrate the following:

1. The pair will require more “man-hours” to produce the assignment than would a sole programmer of a similar experience level.
2. The artifacts produced by the pair will contain fewer defects than anticipated for an individual.
3. Inexperienced programmers, working in pairs, will develop technical and environmental knowledge more quickly, increasing skill levels (and thereby rate of productivity) faster than is normal for new hires.
4. More accurate time estimates can be made for application software development undertaken by inexperienced programmers working in pairs, providing better overall project planning.
5. Inexperienced programmers working within the pair programming framework can be given more difficult assignments than previously thought, increasing the overall productivity of the organization.

1.4 Thesis Organization

The remaining chapters of this thesis will be organized as follows:

- Chapter 2 provides an overview of the pair programming methodology.
- Chapter 3 includes a description of the checkout system environment, an overview of software group practices, and the preparation for the experiment.
- Chapter 4 relates the details of the pair programming experiment.
- Chapter 5 discusses the experiment results.
- Chapter 6 includes the experiment conclusions and recommendations for future research into pair programming.

Chapter 2: Pair Programming Overview

2.1 Evolution of Software Engineering Practices

Coincidental with the history of computers is the history of programming, and eventually, the history of software engineering. Early programming theory consisted of the following steps:

1. Come up with a general idea for a function for the computer to perform.
2. Start “banging away” at the keyboard.
3. Try to run the program.
4. Figure out why it won’t run and fix the bugs.
5. Repeat steps 3 and 4 until the program runs.

Not all the bugs were found, of course - but at least the program (usually) ran. With luck, it might even do what it was supposed to do. Early computer programs were usually simple, and this approach was fairly successful. Today, software is much more complex and the above methodology is still common, but of limited value.

2.1.1 The Waterfall Process

In an attempt to put some discipline in the software development process, in 1970 Winton Royce [8] developed the waterfall process. Royce divided the creation process into a series of steps [20]:

- Careful definition of the intent of the program (discovery).
- The program layout (design).
- The program creation and initial checkout (development).
- Functional checkout and integration into a larger system (integration and system test).
- Efforts to keep the program useful (maintenance).

The waterfall process is serial; one step must be fully completed before the next is begun, hence the metaphoric name.

Although a significant improvement over previous methodologies, as software scale and complexity increased the inflexibility of the waterfall process had drawbacks. Trying to fully develop a large program during the code and unit test phase was difficult; the initial defects were numerous, and finding them in complicated software was problematic. Once a stage of the process was completed, it was not typically readdressed - to do so would add a considerable cost, as every previously-completed stage would need to be reworked. Change to the initial (discovery) phase could mean scrapping the entire effort and starting over.

2.1.2 The Spiral Development Model

To address these deficiencies, Boehm created the spiral development model in the late 1980’s [8]. This model has a series of stages similar to the steps of the

Waterfall process, but emphasizes the need to iterate the development stages a number of times as the project progresses. Spiral development is essentially a series of short waterfall cycles, each producing a working prototype representing a part of the overall project. This model helps demonstrate a proof of concept early in the development cycle, so time will not be wasted on dead-end designs [10]. Each iteration of Spiral development includes the following steps:

- Determine objectives, alternatives, and constraints
- Identify and resolve risks
- Develop deliverables and verify they are correct
- Plan the next iteration

Still, the spiral model has drawbacks. It is a variant of the “build and fix” methodology, where code is written and then continually modified until all the stakeholders are happy. Without proper planning, it can be open-ended and risky [10]. It also does not allow the opportunity to find flaws in the user requirements, and properly implemented code based on poor requirements performs a task no better than poorly written code.

2.1.3 Extreme Programming

Currently, the most radical implementation of software development is Extreme Programming (XP), developed primarily by Kent Beck [1]. All the development phases not only *can* be readdressed, they *must* be readdressed. A software system is dissected into minimal functional parts, then each part goes through its own small waterfall process. The most basic functionality is developed and tested, and then a little more is developed and tested, then a little more. Developers may cycle several times between design, implementation, and testing in the course of an hour’s work [28]. The twelve practices of XP are [1]:

- Start development with a simple plan, then continually refine the plan
- Release code as soon as something is developed that has business value
- Use of metaphors for the software architecture to ease communication among stakeholders
- Keep software design as simple as possible
- Automated tests designed by programmers and customers
- Continual code refactoring (modification) to make the design as simple as possible
- Pair programming
- Collective software ownership - anybody can change the code if it adds value
- Continuous code integration (individual parts into the developing whole)
- Work week not to exceed 40 hours
- Customer available on-site
- Coding standards

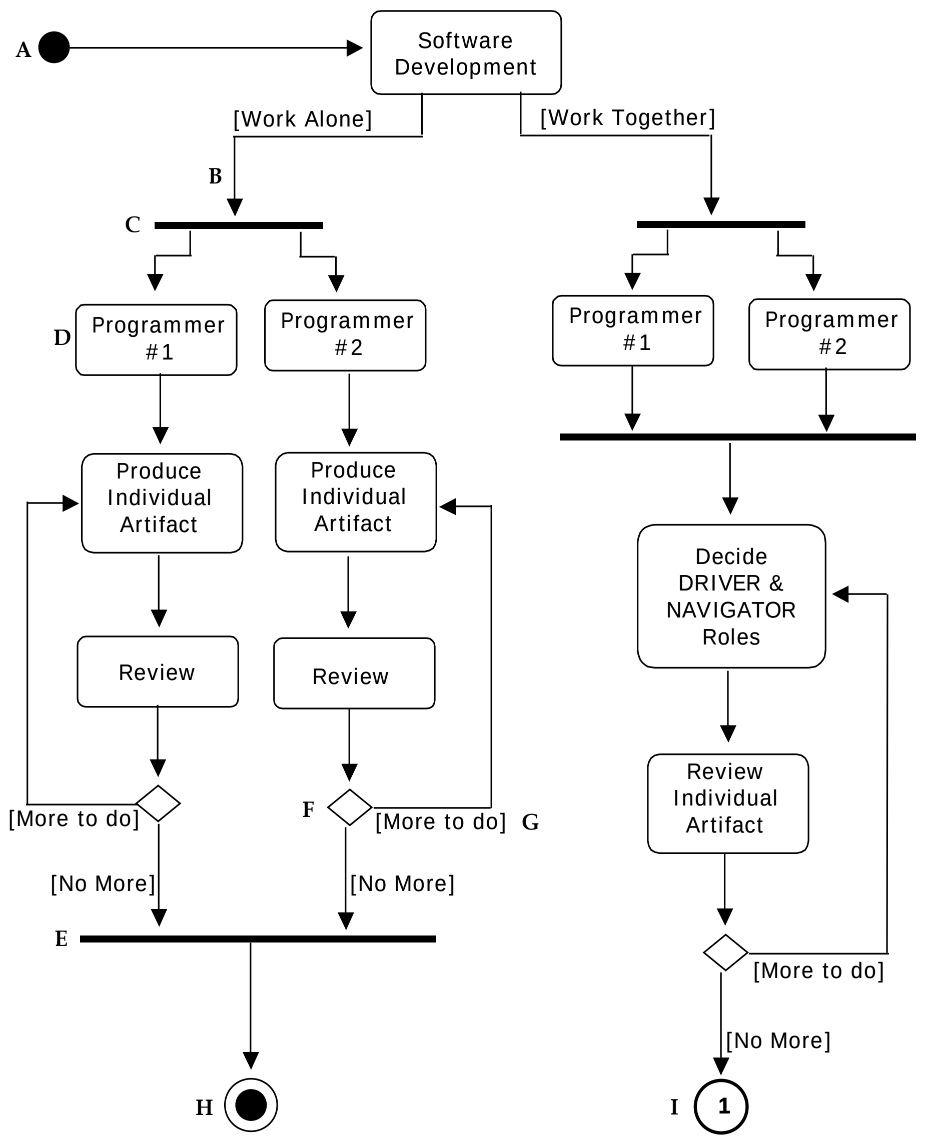
Pair, or collaborative, programming is considered such an inherent part of XP that prototyping performed by an individual is usually scrapped and re-written with a partner [23]. Extreme Programming may be considered too “extreme” by many software development professionals, but an advantage of the XP practices is that most of them can be individually implemented. Specifically, pair programming has received increased interest in the past few years as a standalone programming style.

2.2 The Pair Programming Methodology

In pair programming, two programmers work closely together on the design and implementation of a software system. This usually includes the majority of work at the keyboard. One of the pair is the *driver*, creating an artifact (e.g., documentation or code), and the other is the *navigator*, looking for errors or inconsistencies in the driver's output [26]. While both driver and navigator concentrate on the task at hand, the navigator is also able to keep in mind the overall development task, and to identify when the driver is heading down the wrong path. The pair periodically switch roles, so that the overall output is truly a joint effort. Kent Beck, the creator of XP, reports evidence that two harmonious programmers work together more than twice as fast and think of more than twice as many solutions to a problem as two working alone, while attaining higher rates of defect prevention and defect removal [23].

Pair "programming" may imply that the process deals only with coding, but the style is applicable to all phases of software development. Pairs can work together on requirements review, application design, software debug, and testing with the same potential benefits as seen for code generation.

The Unified Modeling Language (UML) *activity diagram* in **Figure 2.2-1a** and **Figure 2.2-1b** illustrates the pair programming methodology. At the start of any activity is the decision of the programmers whether to work individually or together. This decision may be because of flexible working hours, incompatible schedules, multiple responsibilities, the simplicity of the current task, or just a desire to maintain some degree of independence. While controlled experiments usually have pairs working together 100% of the time, this may be difficult to achieve in the professional environment, and is reportedly not necessary for successful collaboration to take place. Working individually is not contrary to the principles of pairing, as long as all work performed apart is jointly reviewed later.



A Initial state before any events have occurred	**F** Decision transition
B Transition after an action state is complete	**G** Decision description
C Activities which can occur in parallel	**H** Completion of an activity
D Action state or actionee	**I** Transition to another diagram
E Join transition	

Figure 2.2-1a
Pair Programming Activity Diagram

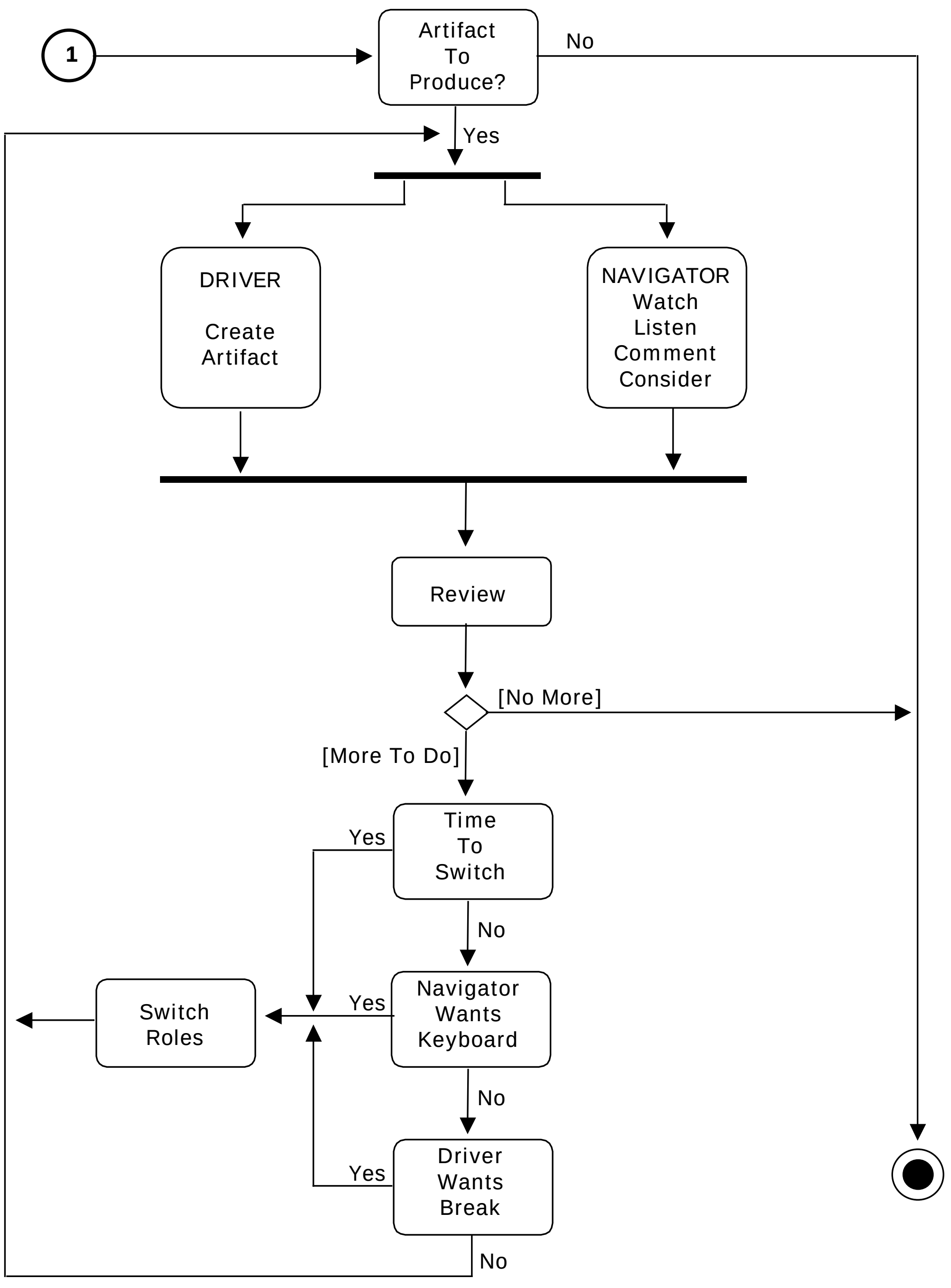


Figure 2.2-1b
Pair Programming Activity Diagram (Continued)

The left side of Figure 2.2-1a shows individual work as a parallel effort, but at a given time only one programmer may actually be working on a project. The programmer works much as he or she would in a non-paired situation, producing an artifact and reviewing it for accuracy before proceeding to the next artifact. However, the programmer also keeps in mind that the pairing partner will be reviewing the work at their next meeting.

The right side of Figure 2.2-1a illustrates the true nature of pair programming, that of collaborative software development. First, any artifacts individually created since the last pairing session are reviewed. Which programmer sits at the keyboard during artifact review is not critical but, for expediency, the artifact originator typically assumes the driver role. Alternately, the other programmer may drive, based on the concept that this will increase his or her familiarity with the artifact. Regardless of who takes what role, all independently-created artifacts are fully reviewed prior to continuing application development.

Figure 2.2-1b illustrates collaborative artifact creation, and for this effort the roles of driver and navigator become more distinct. The driver sits at the keyboard, becoming the primary artifact creator. The navigator watches for syntax or logic errors in the driver's input, keeps the technical conversation alive, provides feedback for the driver’s ideas, and thinks about the current task in relation to the “big picture” of the overall project, looking for dependencies with other artifacts.

Periodic role reversal for the programmers is important. While there may be an urge to have the fastest typist or most experienced programmer be the driver, the pair methodology requires that both participants share the work equally. Being the navigator can get boring - it is difficult to maintain concentration when one’s primary responsibility is watching someone else work. Mandatory role switching on a periodic basis - perhaps every hour - helps both programmers stay alert. This also alleviates feelings of being left out, or of one programmer developing a greater sense of project ownership. Role reversal does not have to wait for a set time limit. The navigator could have an idea and say “Let me drive,” or the driver could reach a mental impasse and need to step away from the keyboard.

2.2.1 Pair Programming Predates XP

A common misconception is that the concept of pair programming originated as part of the Extreme Programming software development style. In truth, the concept of collaborative software development has existed for decades. Here are some historical references:

- Fred Brooks, author of *The Mythical Man-Month*, reports teaming with a partner to program as a graduate student in the mid-1950s [26].

- Dick Gabriel, creator of Common Lisp, commonly used pairing in the early 1970s. His company was under a deadline to implement Lisp within nine months, and he feels that collaborative programming was the technique that allowed the project to be completed on time [26].
- Larry Constantine reported on "Dynamic Duos" working at Whitesmiths Ltd., England, in the early 1980s (as documented in his book *Constantine on Peopleware* in 1995). He found that two programmers working together were not redundant, but produced greater efficiency and better quality [22].
- James Coplien discussed project methodology at Bell Labs in 1995 when he published the "Developing in Pairs" Organizational Pattern. He found that an individual is often hesitant to tackle difficult problems, but will be much more confident if paired with someone else. "Pair [designers]...can produce more than the sum of the two individually [26]."
- Pair programming author Robert Kessler teamed with Martin Griss, coauthor of *Software Reuse*, for years without knowing there was a name for the technique they were practicing [26].

2.2.2 Who Should Pair Program?

There is no optimal personality mix for successful pairing, although egotists, extreme introverts, and just plain “jerks” will probably not be successful contributors. It is important to realize that, within a software development group, certain individuals will never work well together, and certain individuals don’t want to program collaboratively. The pair programming philosophy does not include therapy sessions with programmers to optimize their personalities. It requires people who want to work together (or at least give it a try), and optimizes their chance to produce quality software.

There is also no optimal skill mix for successful pairing. Advantages can be seen in pairing individuals with equivalent programming talent as well as pairing those of disparate abilities. Of prime importance is how the individuals interact, and what they see their role in the joint effort to be. The following descriptions of different pairings is excerpted from Laurie Williams’ book, *Pair Programming Illuminated* [26]:

2.2.2.1 Expert-Expert Pairing

Experts can push each other to exceed their perceived limits. Each feels confident of his/her abilities, and wants not to be “shown up”. Each may have expertise in particular areas, which brings an even greater chance for success - fewer problems arise in which neither has previous experience. *When egos don't get in the way*, expert-expert pairing can produce superior code in a very short time.

2.2.2.2 Expert-Average Pairing

The success of this pairing depends on why the average programmer is "average". Some are average because they're still learning; others are average because, for whatever reason, that's all they ever will (or ever want to) be. It has been found that expert-"unmotivated average" pairing doesn't work very well - the expert ends up doing most of the work. However, expert-"average but eager" pairing can work very well. Most of the challenges are on the expert in this environment - the probable need to slow down from his/her normal pace somewhat, and to perform a certain amount of mentoring. Also, the expert must realize that someone with less experience may still have innovative ideas on design and implementation details.

2.2.2.3 Expert-Novice Pairing

This combination also has benefits, although speed is probably not one of them. Pairing an expert with a novice will undoubtedly reduce the expert's productivity - almost everything must be explained to the novice. For this reason, the expert must be a patient mentor, and not under the pressure of a deadline. The benefits are obvious for the inexperienced programmer, who will undoubtedly gain an accelerated education by working with “the master”. However, the expert often gains as well; the need to teach often makes the expert learn the topic better. And just as with the expert-average pairing, even a less-experienced programmer may have good ideas.

2.2.2.4 Novice-Novice Pairing

The prime benefit of this collaboration is that both members of the pair will learn faster than they would alone. Just like students collaborating on homework, the novice pair receives training from a mentor, then work together to fully understand the instruction. Having a dedicated mentor is critical for this pairing, to keep the programmers on track, or to prevent them from getting stuck on some aspect of the project, or to keep from reinforcing in each novice incorrect learning.

Many pair programming practitioners state that they would *not* let two novices pair together. They see too many problems with keeping the novices on track, keeping them from getting stuck on (perhaps insignificant) problems, and keeping them from bothering other programmers with endless questions. Even if the novices complete an assignment, could it be trusted to be of acceptable quality?

2.3 Quantitative Experiments in Pair Programming

A number of software professionals are eager to advance pair programming as a major advance in software engineering, but the fact is that few quantitative experiments have been performed - and none have been performed in an industrial environment. The three experiments found in the literature are summarized below.

2.3.1 Temple University: 1998

The first documented quantitative experiment on pair programming was performed in 1998, at Temple University. Professor John Nosek assigned five individuals and five pairs a difficult programming assignment. All conditions were the same for the two groups. The results showed that, on average, a pair cumulatively spent 60% more time on the programming task than an individual. That is, for every hour the individual worked, *each* member of the pair worked 48 minutes. As for the elapsed (clock) time required, the pairs completed the assignment 40% faster than the individuals. Significantly, the pairs also produced algorithms and logic constructs of a superior nature. From a qualitative standpoint, the pairs indicated a greater enjoyment of the problem-solving process, and had greater confidence in their results [15].

2.3.2 University of Utah: 1999

A more detailed study was performed at the University of Utah in 1999 using senior software engineering students, hence all had significant programming experience. Professor Laurie Williams gave the students the option of working individually or working in pairs. The class was then divided into two groups - thirteen students working alone and fourteen pairs working collaboratively. Four assignments were given over a period of six weeks. It was found that pair programmers were more consistent in completing assignments on time. The programmers felt they worked harder because they didn't want to let their partner down. This is known as "pair pressure", and appears to be an important aspect of successful pairing. Not all the pairings produced superior results when compared to individuals, but the lower-performing pairs were also below the norm when given individual coding assignments.

Initially, the pairs required about the same percentage increase in man-hours to complete an assignment as reported in the Nosek experiment - around sixty percent. However, in later assignments, this number dropped to as low as 15% additional cumulative hours. When the code was exercised using automated test cases, that produced by pairs passed 15% more tests. The pairs also produced a lower KLOC count, indicating superior software design [3].

2.3.3 Poznan University of Technology: 2000

Not all pair programming experiments have produced positive results. Jerzy Nawrocki of the Poznan University of Technology (Poland) set up an experiment using three groups of senior computer science majors. The purpose was to compare pair programming, along with certain other XP practices, with the Personal Software Process. (The Personal Software Process, or PSP, relies on an individual programmer to keep extensive time keeping and software defect data during the software development process. This data is seen only by the individual, and is used to identify deficiencies in work habits and software development practices.) Six students worked to the PSP methodology; five students worked individually using selected XP practices; and 10 students practiced the same XP methods, but worked in pairs. Four assignments were given over a period of several weeks. As expected, PSP tended to take the most development time, due to the extensive logging of metric data. The experiment showed essentially no difference in development time between the XP individuals and the XP pairs (indicating that the pairs required 100% more cumulative hours). Contrary to earlier experiments, the pair-produced code was only marginally more efficient - in early assignments, code size was virtually identical between individuals and pairs, and in later assignments the pairs produced slightly smaller programs. Further, the pairs did not produce code with significantly fewer defects. The only positive aspect about pair programming demonstrated by the experiment was that results tended to be more predictable; there was less variation in the data points between pairs as compared to individual variations [13].

Obviously, three experiments - two in support of pair programming, and one seemingly indifferent - are insufficient to draw any firm conclusions about the advantages (or disadvantages) of the practice. The majority of support for the practice is anecdotal.

2.4 Qualitative Opinions on Pair Programming

In December 2001, Kevin Dangoor of the *Joel on Software Forum* [5] requested readers to submit anecdotes on their pair programming experiences. Following are excerpts from the responses.

> *"I've had lots of luck with Pair Programming as I believe it's a better work pattern for humans to discuss, create and manage abstraction...Two people working on a task can share the zone quite well with a little practice and I'd say even help you stay there. There have been many times when my momentum has been robbed by some silly little thing that I was doing wrong. When pairing, often that silly little thing is caught in mid sentence and corrected, keeping velocity high."*

"We spent the better part of 6 months...using pair programming to refactor a large chunk of the code base and to introduce some new functionality. One of the things that was apparent from the outset was that you have to choose the pairs very carefully...People need to be the type to point things out and ask questions for pair programming to be effective. When you get two people working together that do point things out, the end results are generally much more solid and innovative than a single programmer."

The ability of people to work together is true for any organization, but especially so when pair programming is being performed. In some ways, it is as much a social interaction as it is a professional collaboration.

"I have had mixed results so far doing pair programming. Note I don't do it full time but the times where I did, it all depended on who I was paired up with."

"My first programming was done in pair programming...though we didn't know that term then...Plusses: Far less bugs were made. Since we were new, we made a lot of boneheaded little errors. The person sitting back next to the typist would pick them off efficiently...There's the peer pressure

to not slack off. [Y]our partner becomes your customer. Minuses: If people are of deeply varying levels, this may be a deadly experience."

The above comment reflects two considerations of the thesis experiment: the potential advantage of novice pairings, and the benefit of involving programmers of similar skill levels.

"Certainly if people can't let go of ego it will be hard to pair...Even the least capable programmer helps me, if they'll just engage."

"In my experience (almost 20 years) I have met very few experienced, very talented and productive programmers without an ego."

"In our team I was unable to implement [pair programming]...Developers can't get that it is faster than coding alone. [They don't understand] you are moving slower, but later you have less problems that eat your time...Developers are usually individualistic...Almost half of [those in my team] hate to work in a pair...To make pair programming work you have to hire the right people."

The author of the above comment seems to have the opinion that pair programming only worked with 100% participation within an organization. This is contrary to most pairing implementations, which recognize that some programmers make effective collaborators, and others do not.

"I have pair programmed numerous times over the thirty years that span my career. I have always thought highly of my skills, but PP always surprises me with how dumb I can be. My partner will constantly question my decisions and design."

"Pair programming is a bad concept - based on a few sound principles badly grouped...Try pairing up two programmers of any skill level for a few weeks and watch. We have tested parts of XP in our office, and noticed some minor amount of good followed by a copious amount of marginal productivity."

After reading this comment, one wonders as to the expectations the author had when "parts of XP" were tested in the office. Pair programming will not make unproductive workers productive. Its strength lies in the possibility of making productive workers *more* productive.

As might be expected, not all software professionals are pair programming disciples. For those who seem to favor pairing, a recurring theme is that pair programming can be beneficial, but only if it is implemented correctly and with cooperative, compatible participants.

In an attempt to solicit more opinions on the subject, this author posted a question on an Extreme Programming message board [4], asking for any personal experience or anecdotes relating to pair programming. The sole response received was cautiously positive:

> *"I think pair programming for new hires is worthwhile to pursue...Early on in my career we had to pair up to do programming [due to a shortage of computers]. One thing to keep in mind is that...the two programmers are more or less on the same skill level. The benefits:*
>
> - *It will help the programmers to bond and teach them to work together.*
> - *It will help instill some collective responsibility for the new hires.*
> - *Such an exercise will also identify future prima donnas.*
>
> *We were able to work reasonably well for about a month. After that, individual aspirations started taking priority. Most of us wanted to work separately to establish our own identity as programmers."*

Of specific interest to this author was information which discussed pair programming in relation to the training of inexperienced personnel and their integration into a particular (technical) environment. Dr. Laurie Williams was contacted regarding any research she had done subsequent to the 1999 University of Utah experiment. She responded with an article awaiting publication, dealing with “Pair Programming and the Factors Affecting Brooks’ Law.” In 2000, she, Anuja Shukla, and Annie Anton conducted an experiment to test the law, which states “adding manpower to a late software project makes it later.” Two surveys were sent to 78 software professionals. The surveys contained questions regarding the effects of pairing on training costs, assimilation time (the time for a new team member to become productive), and intercommunication time (lost productivity as a result of time spent on team communication). Survey responses indicated the following [28]:

- The mean percentage of total time spent mentoring with pairing was 26% and the mean percentage of total time spent mentoring without pairing was 37%.
- Mean assimilation time with pairing was 12 workdays, and mean assimilation time without pairing was 27 workdays.

- Seventy-five percent of the respondents agreed that pair programming reduced intercommunication time through informal, on-the-spot discussions.

2.5 Pair Programming Best Practices

The procedural steps for pair programming (e.g., the driver/navigator concept) are straightforward, but effective collaboration is also about style, and perhaps psychology. Various techniques to make the experience more beneficial are repeated throughout the literature. They are summarized below as the best practices of pair programming adherents.

1. **Pairs must want to pair - or at least shouldn't mind being paired.** Some people work best when they can brainstorm with someone else, and others are much happier working independently. Many programmers may have initial reservations about collaboration; some may never get used to it. Some people just don't like each other. Pair programming isn't about turning introverts into extroverts, or about forcing everybody to get along. It takes willing participants to make a good team [5].

2. **Big egos produce little benefit.**

   As expertise grows, many programmers develop a proportionally greater sense of their own infallibility. Their way is surely the best way, and no junior coder is going to question their abilities. Attitudes like this are sure to make pair programming fail. “Egoless programming,” described in 1998 by Gerald Weinberg in *The Psychology of Computer Programming* [26], discusses the detriment that inflated ego can cause in a collaboration. In pair programming, the individuals should be prepared to spend time as both mentor and student.

3. **Speak your mind, but watch your tongue.**

   Two pair programming scenarios: in the first, the navigator sees the driver do something that doesn’t seem to make sense. The navigator says nothing, because the driver is much more experienced. In the second scenario, the navigator sees the driver perform the questionable action, and asks, “Why did we ever hire a moron like you?” In neither case does the navigator practice appropriate collaboration. In such a situation, an appropriate navigator response might be, “I’m not sure I understand what you’re doing here. Would you mind explaining it?” In this way, the navigator draws the driver’s attention to the questionable action, without putting him on the defensive [23].

4. **Nobody gets along all the time.**

   Any two people working together will occasionally disagree, and pairs with individual coding styles will have different ideas of how software should be designed or written. This can be a positive thing, if it encourages the individuals to reconsider preconceived, and perhaps ill-conceived, ideas. On the other hand, disagreements can be detrimental if they disintegrate into hurt feelings and non-cooperation. Attempt to resolve differences when they occur. If this initially fails, perhaps the best thing to do is to take a break, giving both programmers a chance to carefully think through the issue [26]. If resolution still cannot be reached, a third party (ideally an experienced programmer) can be brought in to resolve the situation.

   Disagreements between pairs are not always of a technical nature - they may be personal. Resolving personal issues is beyond the scope of this paper (or of pair programming, for that matter). Suffice it to say that collaboration cannot be successful where personal conflict exists. If resolution cannot be found, the best action may be to discontinue the pairing of the participants.

5. **It doesn't need to be a full-time job.**

   Pair programming isn't "all or nothing" - that is, the participants don't have to work together full time. Successful collaborative implementations have been reported with as little as 25% of the development time spent together [26], and it is only in controlled experiments that 100% teaming is typically reported. Many factors can influence the time spent collaboratively. The pair may have different working hours, or may have different meetings to attend during the day, or may just feel the need to work alone at times. Perhaps it is important for one member to have full ownership of a small part of a larger application. Working alone at times is normal, as long as when the pair get back together, they fully discuss the work done independently.

6. **Talk.**

   Effective communication is a must. The literature recommends that when working together, the pair should communicate at least once each minute [26]. At first, programmers new to paring may be uncomfortable with keeping up the chatter; they are used to working alone, and may feel that excessive talking disrupts the thought process. In fact, keeping up a running commentary tends to make the programmers stay on topic, continually ensuring they're on the right path. Communication is especially beneficial to the navigator, as watching someone typing at the keyboard can get tedious. Conversation helps the navigator stay focused on the task of reviewing the developing code.

7. **Listen.**

   This is important for all pairings, and perhaps even more critical for the expert programmer paired with someone less experienced. Realize your partner may know things that you do not. Let the other programmer freely speak his or her mind. Remember that an opinion different from yours is not a personal criticism or a put-down. Carefully listening to the questions of others may point out a flawed design [5]. Collaboration where one person does all the talking, but never listens, isn't collaboration at all. It's domination, and doesn't benefit the pair experience.

8. **Do your share.**

   Some programmers state that when they work with a partner, they feel like they have another customer. No longer is the work being done only for the requestor, it's being done for the collaborator as well. This is the ideal mindset for the paired experience. There should be a sense of obligation to actively contribute to get the job done. However, doing one's share doesn't mean doing the work of one's partner as well. Occasionally picking up the slack is fine, but covering for a slacker is not. On the other hand, a partner not doing the work is not the same thing as a partner not doing the work *to your satisfaction*. It is easy for a dominant, opinionated programmer to start taking over when the other member isn't doing things "the way they should be done." Do your part of the job, and allow (and expect) your partner to do theirs [23].

Chapter 3: Experiment Description

The purpose of the thesis experiment will be to test the theory that pair programming can produce beneficial results for novice-novice pairings. Members of the application software group at Kennedy Space Center will perform this experiment. Two novice programmers will be selected to produce portions of a software application to be used for the verification of payload hardware prior to integration into the space shuttle. This chapter provides background of the experiment environment, and details the experiment preparations.

3.1 The Environment

Before an International Space Station (ISS) payload element is launched, its functionality is verified in the Space Station Processing Facility (SSPF) at Kennedy Space Center. This ground checkout verifies not only the proper operation of the electrical systems with the payload element, but also the ability of the element to interface with other ISS elements. Testing is performed primarily through the Test Control and Monitor System (TCMS), a Unix-based platform that can be tied into the payload flight element's data buses and used to issue commands to, and receive telemetry from, that flight element.

ISS system engineers perform checkout of the flight elements by way of a written test procedure. The test procedure is based on the verification requirements levied by program management. The system engineers are told what must be tested, and must develop a means to perform the required tests. Often, the result is a request for a new TCMS software application. Over the past few years, TCMS applications have been developed to facilitate checkout of all the major subsystems of ISS hardware - command and telemetry systems, power systems, fluids systems, and environmental control.

3.2 TCMS Application Software Development

The payload checkout application software group ("App SW") is responsible for developing the TCMS applications used to verify the flight element (end-item) functionality. These applications are written in the common ANSI C, with the uncommon SL-GMS GUIs as the user interface. While all applications are different as to what they do, all are usually similar in that they send commands to the end-item, receive telemetry (measurements or data) from the end-item, and provide a user interface for command issuance and data display. The App SW software development lifecycle most closely resembles the partial iterative model [20]. After requirements are well-defined and understood, design, development, and (unit) test are performed incrementally. At any stage of the development process, revisions may be generated which cause rework of previous stages.

For a new TCMS application, the ISS system engineer prepares a Software Requirements Document (SRD) which describes commands to be issued, measurements to be monitored, automated steps to take when error conditions result, and any other required functionality. Rough layouts of the desired GUIs may also be included. When the SRD is submitted to App SW, a senior software engineer reviews the SRD for feasibility. After review, the system engineer is contacted to clarify any vague requirements. Once the SRD is fully understood by the software engineer, an effort is undertaken to estimate the manpower requirements for developing the software. Then the group embarks on the development using the aforementioned process.

3.2.1 Level-of-Effort Estimation

App SW has developed a model, unique to the TCMS environment, for estimating software development effort. For a new TCMS application, this model generates estimates based on the following information (not a complete list):

- Number of GUIs
- Number of GUI objects
- Number of GUI data fields

- Number of application function points
- Number of TCMS commands
- Number of TCMS measurements
- Assigned programmer's experience level
- Relative application complexity

While the first six items in the list are calculated directly from the requirements, calculation of the last two - programmer experience level and relative application complexity - are somewhat esoteric. Both are multipliers against the "base" man-hours calculated from the first six items. Programmer experience level ranges from .5 (expert) to 2 (novice), with 1 (experienced) as the nominal value. Relative application complexity ranges from .5 (easy and/or much code reuse) to 2 (complex and little code reuse), with 1 (average complexity, some code reuse) as the nominal value. The level-of-effort formula would look like the following:

$$(\text{Experience Level}) * (\text{Complexity}) * (M_1 + M_2 + M_3 + M_4 + M_5 + M_6)$$

3.2.2 Application Software Metrics

The App SW group gathers metrics on two items: the estimated vs. actual software development effort in man-hours, and the software defects found during (and subsequent to) functional test.

3.2.2.1 Estimated vs. Actual Software Development Effort

The App SW group was formed in 1994, and has to date developed approximately 60 TCMS applications. During the first few years, software development estimates were little more than guesses, and it was only by chance that the actual hours worked were close to the projected value. As experience grew, the estimates generally became more accurate. However, variations still occur - sometimes large ones. Presently, an estimate is considered to have been reasonably accurate if it is within 15% of the actual hours. For example, if an application development effort is projected to require 500 man-hours, that value is advertised as the nominal value only, with an expected range of 425 to 575 man-hours. The App SW estimation tool has undergone revisions in an attempt to reduce the estimated vs. actual variance, but the group still only manages to fall within the 15% variance about 80% of the of the time.

3.2.2.2 Software Defect Data

The App SW group tracks application defects found both during development and after release. Code and GUIs are created on a development testbed, a Unix platform that runs the TCMS system software but has no interfaces to ISS flight elements. The testbed does provide a very rudimentary simulation capability, allowing the basic functionality of the code and GUIs to be verified. After testbed development is complete, the application is moved to a TCMS test set - still not connected to ISS flight elements, but to a fairly realistic flight element simulation capability. Functional test, integration test, and acceptance test are all performed in this environment.

Defect data for metrics are not taken while in the testbed facility. During the application development effort, App SW programmers are free to code in any style they choose. The group philosophy is that the benefits of tracking defects at this point would be outweighed by a decrease of creativity and morale. Defects are recorded after the application is moved to the TCMS test set simulation environment, specifically during application functional test. No distinction is made as to the type of defects found, other than where the defect was located (code vs. GUI). A defect may be the result of deviation from requirements or design, a logic error, or a syntax error not detected by the compiler. Occasionally, there is contention as to whether an anomalous condition is actually a “defect”, or is the result of a poorly written requirement; in such cases, the resolution is usually to count it as a defect. Regardless of the root cause, the developer corrects all detected software anomalies, and the functionality is retested.

The programmer is often working alone while in the simulation environment, and is expected to truthfully report defects, but no independent verification of accuracy of defects noted is performed. Currently, historical defect data is not used to project the anticipated defects that will be found in new applications. Only the projected level of effort is estimated.

3.2.2.3 Historical Software Defect Analysis

For inclusion into this paper, an attempt was made to perform an analysis of the group's historical GUI defect data. For each application, the GUI effort was identified as small, medium, or large. These categories were based not only on the number of visible objects in the GUIs, but also on the imbedded complexity. For example, a TCMS GUI may have a large object count, but the objects themselves may be very basic (easily constructed, no imbedded dynamics). Other GUIs may contain relatively few objects, but each object performs complex data manipulation. GUIs of these two extremes could have a similar level of effort. The analysis of historical TCMS GUI defects is summarized in **Figure 3.2-1**.

Effort	**Defects/GUI**
Small	< 1
Medium	1.2
Large	2.6

Figure 3.2-1
Historical Group Defects vs. GUI Effort

These figures will be used to compare the effectiveness of pair programming.

3.3 Coordinating the Experiment with Management

Before work could begin on the proposed experiment, it had to be approved by App SW management. In previous development efforts, the group used teaming for particularly large applications, and it could be said that collaborative programming had informally taken place during those projects. However, this experiment required placing two novice developers on a project ordinarily assigned to a single programmer. A portion of the pair's time would be spent learning pair programming techniques, and in discussing their experiences. Management had to be made knowledgeable of the atypical required manpower expenditure.

Of even greater importance was to assure management that the experiment would not compromise the software delivery date, or the quality of the application produced. The software would be used to control and monitor critical space flight hardware, and serious damage could occur if commands were incorrectly issued or measurement telemetry incorrectly processed.

A meeting was held with the App SW manager to brief him on the fundamentals of pair programming. Two published articles were discussed, one in favor of pair programming and one which questioned its benefits. This was, after all, an experiment, and the outcome was uncertain. The experiment might be an unqualified success, or it might turn into a disaster, or it might prove nothing. The manager was assured that experiment progress would be closely monitored, and that it would be halted if progress was not going well.

The App SW manager approved the experiment, providing that he be given regular status updates, this author assumed overall responsibility for the application, and this author would step in to complete GUI development if required.

3.4 The Pair Programming Participants

3.4.1 Selecting the Participants

It was desired to perform this experiment using two specific programmers. This was done for several reasons:

1. The pair are relative novices within the group, and have approximately the same skill and productivity levels.
2. This author had previously mentored these programmers, and a good working relationship had been established.
3. The programmers seemed to get along well, which is essential for successful pair programming. It was not desired to add complexity to the experiment by trying to get incompatible programmers to work together.
4. While the selected programmers were relatively inexperienced, they were highly motivated and eager to demonstrate their abilities.

5. Inexperienced programmers are often more willing to try new concepts - they rarely say, “That’s not the way we do things around here.”

3.4.2 Introducing Pair Programming to the Participants

An informal meeting was held with the programmers to discuss the experiment and to solicit their participation. It was stressed that the experiment was of secondary priority; successful and timely development of the application was essential. The pair would be given the assignment whether or not they agreed to work together. If they started to work collaboratively but subsequently decided that it wasn’t going well, the experiment would stop; they would continue the development working independently. A valid conclusion to the experiment could be that it just didn’t work out. After the pair voiced some concern about being associated with a potentially failed experiment, it was decided that they would simply be referred to as *ProgA* and *ProgB* in this paper. Both programmers agreed to participate.

In three additional meetings prior to the start of the experiment, the participants were instructed on the pair programming methodology and best practices. They were made aware that the practice is somewhat unconventional, and reminded that there was not a preconceived need for the experiment to succeed. The two were told that certain techniques, such as equal time at the keyboard and pair review of any work done independently, should be followed closely. Other aspects were left up to *ProgA* and *ProgB* - for instance, how often they actually worked together. The two were requested to work at least 33% of the time together, 75% if possible, and 100% if desired.

3.4.3 Historical Software Metrics for the Participants

ProgA and *ProgB* have both been in the TCMS Application Software group for approximately 18 months. During that time, each has worked only on GUI development. The GUIs developed by *ProgA* and *ProgB* have had a small-to-medium level of effort. While the two have developed GUIs for a number of applications, their early efforts were as part of a team. As a result, defect data is not available for their specific products, only for the team as a whole. On the most recent assignment for each, *ProgA* and *ProgB* were responsible for all GUI development of their respective applications, and defect data for those efforts is shown in **Figure 3.4-1** (again, level of effort is small-to-medium):

	Number of GUIs	**Defects**	**Average Defects/GUI**
ProgA	18	34	1.9
ProgB	11	45	4.1

Figure 3.4-1
Historical Defect Data for the Pair Programmers

3.4.4 Software Metrics for the Experiment

Level-of-effort projections will be made for the GUIs to be developed. Two sets of projections will be made: one for a solo programmer performing the work, and another for pair programmers performing the work. The collaborative programmers will keep time sheets on a daily basis, tracking all GUI design and implementation efforts. The projected and actual man-hours spent on the application will be compared against both the individual programmer and pair programmer projections.

Estimates for the expected number of defects will also be made, again for both a solo programmer and for pair programmers. Beginning with functional testing, the collaborative programmers will keep track of all defects found. Again, this actual data will be compared with the solo/pair projections.

3.5 Criteria for Application Selection

The software to be produced by the pair programmers needed to meet certain criteria. First, it had to be chosen from the list of applications that had been requested by the users but not yet implemented. Second, it should include tasks in which the programmers had prior experience. Third, it required tasks in which the programmers had no prior experience. Fourth, for purposes of this experiment, the pair assignment needed to be small enough to allow it to be completed in a reasonable amount of time.

The application chosen met all criteria. Its overall purpose was to allow system engineers to initiate an automated sequence that would activate the various avionics ("black boxes") of a payload element. The application would require three GUIs, in some ways similar to what the programmers had previously developed, but in other ways considerably more complex. For example, user selection of one object ("clicking a button") would alter the way information was displayed in various data fields. In other display fields, data shown would be the result of manipulation of multiple telemetry items. The GUIs would have been "interesting" for an experienced developer, but quite challenging for novices.

The team for the experiment would be comprised of this author and the pair programmers. This author would be the technical leader of the project, responsible for the high-level application design and code implementation. The pair programmers would be responsible for the detailed design and development of the GUIs. Application functional testing would be performed jointly by all participants.

3.6 Estimations

The final step before beginning the experiment was to generate level-of-effort and defect estimates for the work performed by the pair. One set of estimates would be for the pair; the second set of estimates would be for an individual of the same approximate skill level as the pair.

3.6.1 Level-of-Effort Estimation

The level-of-effort estimate was developed using the group's standard software estimation tool (described previously). The GUIs to be developed were of moderately high complexity, and the developer GUI experience level was considered higher than "novice", but lower than "experienced". Based on these initial assessments, an estimate was made for the effort that would be required by one programmer to perform the work.

To arrive at the estimate for the pair, the individual estimate needed to be multiplied by some value. The University of Utah experiment [3] described previously indicated that initial pair programming efforts required 60% more man-hours than did an equivalent solo effort but, as experience grew, required as little as 15% additional hours. It was decided to adopt a multiplier of 60% for the pair programming estimate.

To begin the estimation process, the three required GUIs were roughly mapped out. This provided a count of the various GUI items, summarized in **Figure 3.6-1**.

	Active Objects	Command Buttons	Telemetry Fields
GUI 1	15	11	30
GUI 2	17	29	11
GUI 3	18	11	10

Figure 3.6-1
Summary of GUI Items

Based on the GUI complexity, the level of effort could be estimated. This estimate is shown in **Figure 3.6.2**.

	Individual Hours	Pair Hours
GUI 1	59	94
GUI 2	38	61
GUI 3	27	43
Total	124	198

Figure 3.6-2
Estimated Development Hours - Individual vs. Pair

3.6.2 Defect Estimation

Defect estimates were developed based on historical group defect data, while factoring in the experience level (somewhat above novice). The University of Utah experiment indicated that 15% fewer defects might be expected in a pair programming effort when compared to individual effort, and it was decided to use that value as a goal for this experiment. The defect estimates developed are shown in **Figure 3.6-3**.

	Individual Defects/GUI	**Pair Defects/GUI**
GUI 1	3.7	3.1
GUI 2	2.2	1.9
GUI 3	1.6	1.4
Average	2.5	2.1

Figure 3.6-3
Estimated Defects - Individual vs. Pair

Chapter 4: The Pair Programming Experiment

Within the App SW group, it is standard practice for a senior programmer to present the requirements for new development to any junior programmers being assigned to the task. This is to ensure that inexperienced developers fully understand the details of a new application, to point out functionalities previously developed (code reuse opportunities), and to alert them to requirements that may be especially challenging.

4.1 Introducing the Task

A meeting was held with the pair programmers to present the user requirements document, and to specifically discuss the application GUI design. Initial requirements from the system engineer had included a rough layout of one GUI. Boxes were used to identify the position of data fields and commanding buttons. Each was textually labeled and, where appropriate, contained cross-reference indices to other requirements which gave more detailed information on the GUI element functionality. The requirements for many of the GUI objects were considerably more complex than anything previously experienced by the programmers. In previous assignments, the most challenging GUI requirement they had encountered might be:

> *For data field 10, display the output voltage of the primary power supply. If voltage is between 100 and 125 volts, display it in black text with a white background; otherwise, display it in white text with a red background.*

For this new GUI, a typical requirement was more like the following:

> *For data field 10, display the output voltage of the desired power supply (primary or secondary) as specified by data field 7. The baseline voltage of the power supply specified in data field 7 is determined by data field 5. If the voltage is within the range of -10 volts to +15 volts of the baseline voltage, display it in black text with a white background; otherwise, display it in white text with a red background. If data field 7 is blank, display no voltage.*

While the requirements document only contained a stated need for one GUI, the requirements review led to the conclusion that two other GUIs must be developed. One would be for the user to set up parameters required for real-time configuration of the application. The other would be a message display, either providing the system engineer with test execution tutorials at appropriate times, or showing informational alerts if test parameters exceeded acceptable limits.

At this first design meeting, it was obvious that the pair were fairly intimidated by the complexity of the task. They were told that help was available within the group if needed, but that they should see what they could accomplish together before seeking additional assistance.

4.2 Tackling the Assignment

By the next day, the pair had developed an approach for GUI development. The first step would be to create each display with no imbedded functionality. These prototypes would be presented to the system engineer for review and comments. After the user approved the GUI layouts, functionality would be incrementally added. The pair would begin with the easier implementation tasks (things they knew how to do), and then move to the more challenging ones. Of the three GUIs, one would be primarily "owned" by *ProgA*, another by *ProgB*, and the third (the most challenging) would be co-owned.

At first, this author had some reservation about the idea of GUIs being owned by the programmers. Didn't this defeat the concept of collaboration? Upon further thought, and conversations with the participants, a degree of ownership was seen as an acceptable approach to the task. It gave each programmer a clearly defined assignment when working independently, ensuring they wouldn't inadvertently be addressing the same issues. Also, work performed independently would be reviewed by both the next time they got together to collaborate. With the development approach in place, work on the GUI prototypes began.

4.2.1 Initial Attempts at Pairing

At first, the pair programmers didn't seem to be "pairing" all that often. After three days of development, a brief status meeting was held, and the pair were asked about their initial impressions of how the effort was proceeding. They indicated that they were still getting used to the idea of working so closely together, and were trying to find a comfortable blend of working alone versus working collaboratively. However, the two had been working as a pair more than was perceived; they had just been doing it differently than expected. Instead of spending long stretches together, they had been getting together for five or ten minutes at a time, one usually going to the other only when a technical roadblock had occurred or there was a question regarding GUI layout. By this time, the two "independently owned" GUI prototypes had been completed, and were reviewed. While both were satisfactory individually, each had a different look and feel; it was apparent that two individuals had developed them. The pair were guided to see the advantage of providing a standardized user interface, and that they had a better chance to accomplish this if they worked more closely together. Pairing to resolve problems was a good start, but more collaboration was recommended for the design details.

4.2.2 The GUI Prototypes

Within another two days all three GUI prototypes were complete, and an informal meeting was held with the user for feedback. The pair were in charge of the meeting, with this author in attendance as application lead developer. The system engineer expressed overall satisfaction with the GUIs, but requested some changes. Most of the changes were minor, but one was fairly significant, caused by a misinterpretation of the requirements by this author. The change required a GUI to determine, from a set of communications boxes, the "operational" communication box, based on multiple measurements from each box. As the GUIs were in the prototype stage, no imbedded code had yet been written, so the change would not impact the development effort - but that portion of the design would need to be redone.

Following the meeting with the user, this author and the pair programmers discussed the required changes, and specifically the significant design change. The pair had minimal background experience for this particular design detail, so a solution was suggested to them. The proposed solution was complex, but logically sound. The pair were left to decide how to incorporate the other requested changes.

At the tag-up meeting the following Monday, the pair presented the modified GUI prototypes, incorporating the layout changes requested by the user. Additionally, the pair somewhat tentatively presented an alternative implementation for the design change suggested by this author. The new implementation was more elegant than the original suggestion, and was endorsed. Had this unexpected contribution resulted directly from the benefits of collaboration? The two were noncommittal on the contribution pair programming might have played in their coming up with the better design. However, the idea *had* been conceived while they were working together.

4.2.3 Detailed Implementation and Observations

With the GUI prototypes approved, the pair began developing a plan for making the GUIs functional. During this process, requirements were identified that were still unclear, having not been addressed in the meeting with the user. The pair developed a list of concerns, and resolved them through direct meetings with the user and via e-mail. During past assignments, neither of these individuals had much dealing with the user community, and had seemed uncomfortable in initiating meetings. Working as a team, they appeared to be less hesitant about admitting their lack of knowledge (or experience) to the requestor.

4.2.3.1 Getting Comfortable with Pairing

With most questions resolved and an implementation plan developed, the pair began to develop a routine. On most days, one or two collaborative sessions were held. Sessions held in the morning were rarely as productive as those held in the afternoon; morning collaborations involved a good deal of non-business chatter. Rather than being a waste of time, this was seen as beneficial to the rapport aspect of the pair programming effort. A good rapport between collaborators is crucial to successful pairings.

The programmers took fairly equal amounts of time as driver and navigator, but it was noted that work usually took place at the driver's desk. Also, when work was being performed on one of the individually owned GUIs, the owner was almost always the driver. Each programmer apparently wanted to maintain a certain amount of control over the software they saw as "theirs". This had not been anticipated, but was not seen as being contrary to the pair programming style. The navigator was still fully involved in the process.

4.2.3.2 Pair Mentoring

For the technical issues of GUI development, mentoring took place on a daily basis, and was offered in the same way as it would have been to an individual, inexperienced programmer. Mentor-initiated discussions were always held with both pair programmers. These discussions were included in the author's experiment journal, which is included in the appendix to this thesis.

At least once each day, the pair were asked for a brief status ("How's it going?") and whether they had any issues with which they needed help. This author monitored progress closely enough to offer suggestions for upcoming work, but also tried to avoid micromanaging the development.

An extended meeting was held weekly with the pair to specifically discuss the collaborative effort, to answer any procedural questions the programmers may have, and to provide additional pair programming tips. For the most part, these sessions were used to get the pair's impression of how the experiment was progressing. They had few questions as to how to proceed, and this author had discussed the salient features of pair programming during the initial meetings. The basic concept of collaboration is very simple.

During the first week, the programmers approached the mentor many times with questions relating to the experiment and the development effort. Most questions dealt with technical aspects of the project, and a smaller number of questions dealt with the collaborative process. It was noted that the majority of the discussions were initiated by a single programmer while the collaborators were (apparently) not collaborating.

By the second week, the number of programmer-initiated discussions with the mentor had dropped, and this trend continued for the rest of the development process. More often, it was the pair (as opposed to one of them individually) which asked for technical help or clarification. An accurate count of the number of conversations was not kept, and no historical data exists for comparison, but this author believes that the pair asked fewer overall technical questions than might be expected of a similarly skilled individual.

4.2.3.3 Observations of Note

Some technical issues were still beyond the ability of the pair to resolve, and they went to the group's GUI expert (not this author) for help. This resulted in "triplet" programming for a brief period, involving the pair programmers and the expert. The navigator was less verbal during these sessions, but still involved. While the expert was telling the driver what to do, the navigator was taking notes.

The experiment had not been widely advertised within the App SW group, so most of the other programmers were unaware of the pair programming effort. In the past, seeing two people sitting together meant that work was *not* going on, so occasionally an idle programmer would stop by to socialize. Not wanting to be impolite, the pair would accommodate the interruption, and work effectively stopped.

An immediate benefit seen by the pair was the early detection of syntax errors. In the TCMS environment, individual telemetry measurements are identified by 13-character alphanumeric strings, and are easy to enter incorrectly - especially on GUIs displaying a large amount of telemetry. Additionally, neither of the pair are particularly accomplished typists. The difficulty in finding this type of syntax error is that a badly entered identifier could be a valid name, just not the one desired. When the navigator was observing the driver, several of these errors were immediately detected.

Even though this was a collaborative effort, the personalities of the individuals were evident in how the experiment progressed, and in the design of the GUIs. *ProgA* was clearly the dominant programmer. During the prototyping phase, the collaboratively produced GUI was observed to be more like *ProgA's* individual GUI than *ProgB's*. After the GUIs were made to all look similar, they retained most of the features of *ProgA's* original design. *ProgA* tended to ask more questions during the mentoring sessions, and tended to be the one in charge as the two resolved technical issues. Additionally, the time logs indicated that *ProgA* was spending more time working alone than *ProgB*. Where *ProgB* unfailingly worked an eight-hour day and no more, *ProgA* occasionally worked past normal hours to complete an implementation detail. This disparity of effort was not large, but was evident. It was not noted to cause any friction between the pair.

Related to the above, one of the theoretical side effects of pair programming is “pair pressure”. As reported by Laurie Williams, this is the tendency for collaborators to feel greater responsibility for a project than they might feel when working alone [24]. Observations indicated that *ProgA* might be exhibiting the pair pressure effect, while *ProgB* was not. This is not to say that *ProgB* was unproductive, but that *ProgA* seemed especially productive.

Chapter 5: Experiment Results

This chapter compares pre-experiment manpower estimates with actual hours worked, details the post-implementation test results and provides feedback from the programmers regarding the pair programming experience.

5.1 Estimated vs. Actual Level-of-Effort

At the conclusion of the experiment, the time logs maintained by *ProgA* and *ProgB* were used to compare the actual hours worked to the initial estimates. Prior to the experiment, the existing estimation tool gave an estimate of 124 hours for an individual programmer to develop the GUIs. Using the University of Utah experiment [3] as a baseline, which indicated that the pair might require 60% more cumulative development time than would an individual, it was estimated that the collaborative effort would require 198 development hours. As indicated in **Figure 5.1-1** below, the pair actually required 170 hours of development time.

During the experiment, the pair programmers required approximately 37% more hours than the estimated time for an individual to complete the task. Also, the pair actual effort of 170 hours was 14% less than the pre-experiment estimate of 198 hours. It is important to remember that, as opposed to the University of Utah experiment, the pair did not collaborate all the time. A significant amount of the work was performed individually.

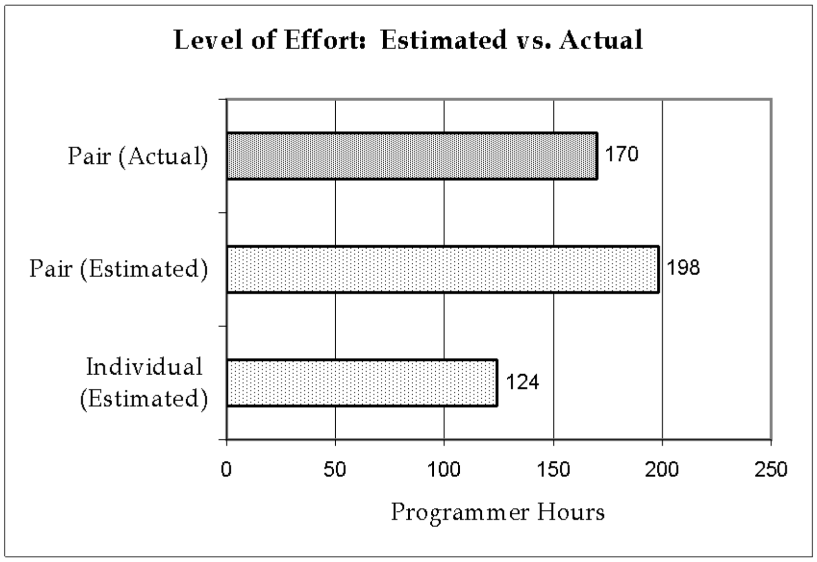


Figure 5.1-1
Level of Effort: Estimated vs. Actual

The time logs for the pair programmers had been maintained such that individual effort could be tracked and compared. This comparison is shown in **Figure 5.1-2**. In total, *ProgA* spent 93.5 hours on the project compared to *ProgB*'s 76.5 hours, a difference of about 22%.

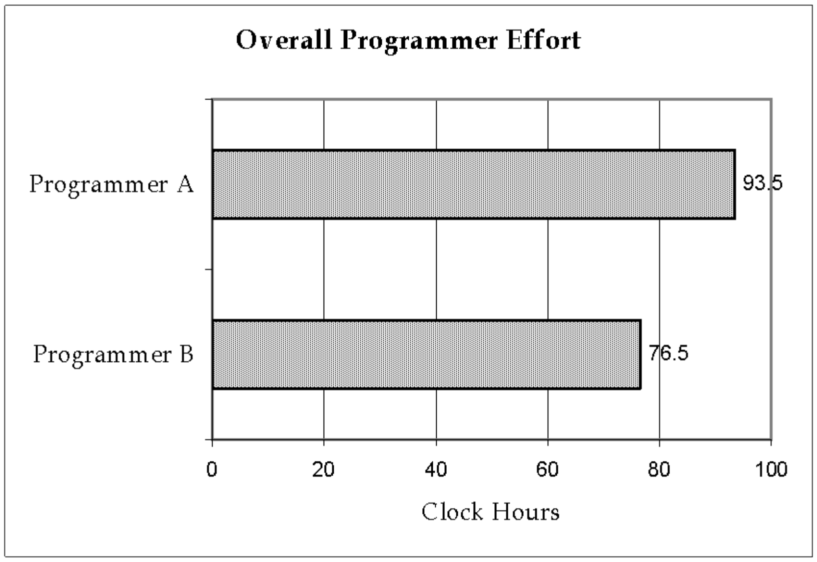


Figure 5.1-2
Overall Programmer Effort

Additionally, the time spent working alone could be compared to the time spent working together. The data obtained are shown in **Figure 5.1-3**.

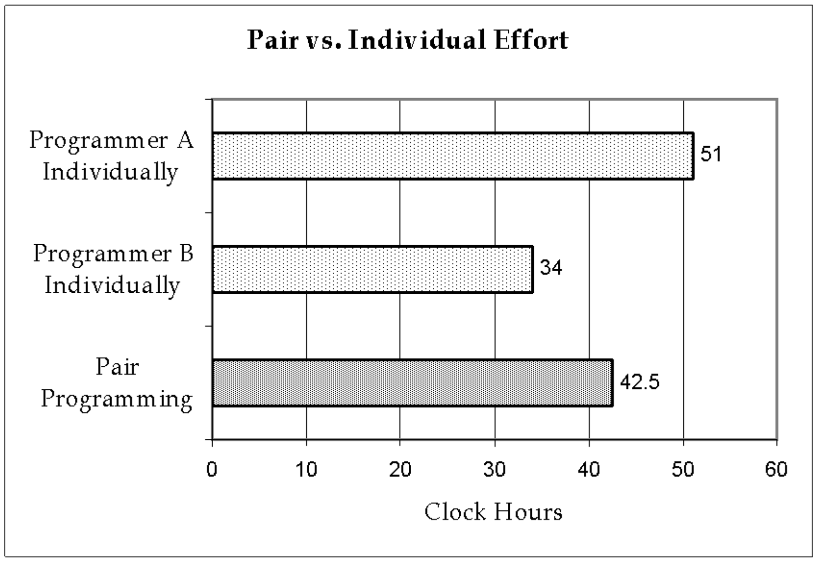


Figure 5.1-3
Pair vs. Individual Effort

ProgA spent 55% of the time working alone and 45% of the time in collaboration. *ProgB* spent 44% of the time working alone and 56% of the time in the pair environment. The percentage of time working together relative to overall time was 50%, between the 33% to 100% desired.

5.2 GUI Functional Test

Functional testing is performed in a realistic TCMS simulation environment, but is not formalized. It is a procedure internal to the group, and is the final debug opportunity prior to acceptance testing. An informal test plan, essentially a checklist of functions to be verified, was developed. The test would demonstrate GUI operability in normal conditions (success oriented), the ability to react to anomalous conditions (abnormal telemetry values), and the ability to handle unexpected user actions (rapid button clicks, abnormal values entered into response boxes, etc.). Testing would not be limited to the test plan; it existed primarily as a road map of the functionalities to verify.

The pair programmers continued their collaboration after GUI implementation was complete, becoming pair testers. They had full responsibility for the GUI functional test: developing the test plan, arranging for the TCMS testbed availability, running the test, and correcting all anomalies found. The only request made by this author regarded the driver/navigator responsibilities. For each GUI primarily "owned" by one of the pair, the other programmer would be the driver during the GUI's checkout. It was felt that this arrangement would provide the best probability of detecting problems.

As for any functional test, the pair kept a tally of anomalies discovered, but kept no records that would tie an anomaly to a specific GUI. Due to the individual ownership issue, this would avoid the possibility of tying defect data to an individual. For this experiment, each defect was additionally categorized as one of the following:

- Syntax error
- Logic error
- Requirement Nonconformance

The experiment hypotheses had not considered the types of defects that might be found in the GUIs, but it was theorized that syntax errors uncovered during testing should be unlikely due to the pair programming methodology used during coding.

The GUI functional test was performed over a three-day period. The anomaly information recorded is summarized in **Figure 5.2-1**.

Defect Type	Number Found
Syntax Error	1
Logic Error	2
Requirement Nonconformance	1

Figure 5.2-1
Summary of Defect Data

GUI defects found, as shown in **Figure 5.2-2**, were somewhat lower than expected.

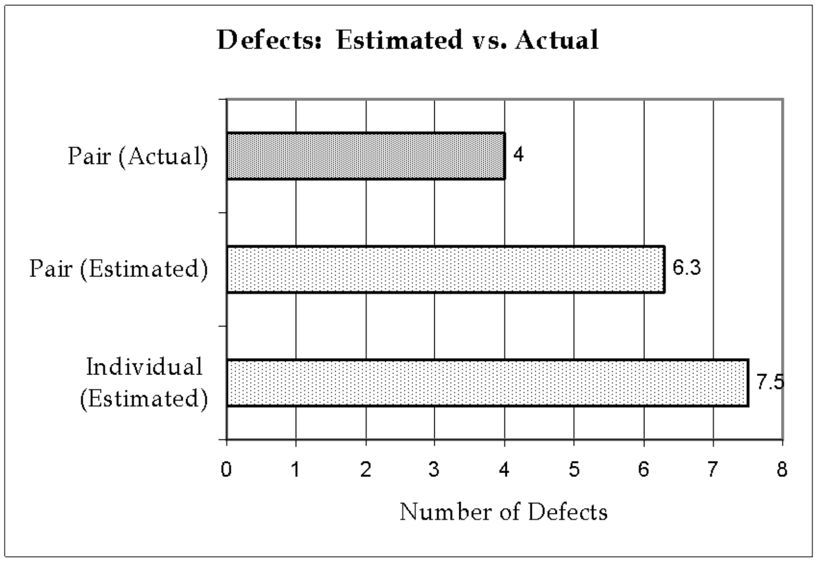


Figure 5.2-2
Defects: Estimated vs. Actual

These numbers were better than the pre-development estimates, which indicated that six or seven defects might be found in the code during functional test. The single syntax error was for an incorrect telemetry measurement identifier, and the pair was uncertain if it was a true syntax error (typed incorrectly) or was a requirement misinterpretation (read incorrectly from the user document). The requirement nonconformance was injected when the wrong telemetry measurement identifier was used to determine the color of a data field. One of the logic errors was actually injected by the GUI expert when he was consulting the pair. The programmers admitted that they had not checked this code as carefully as the work they originated. The second logic error had the "default" state for a data field being set to an undesired value. The syntax error and one of the logic errors were injected while the pair were working individually, and the other two errors were injected during collaboration.

5.3 Participant Opinions

To obtain a qualitative judgment of the pair programming experience, a list of prepared questions were posed to *ProgA* and *ProgB*.

1. *If you had to choose between "good" and "bad", was the overall pair programming experience a good one or a bad one?* Both programmers agreed that the experience was good. They were initially concerned about being the main participants in an experiment, but eventually felt comfortable with the arrangement.

2. *Would you be willing to pair with someone else for another experiment in the future?* Both programmers were cautious about this, agreeable to future collaborations, but concerned with whom they might be teamed. Each had opinions about working with particular members of the App SW group, with *ProgB*'s list being more restricted than *ProgA*'s.

3. *If you paired again with someone else, would you prefer to work with someone at your skill level, or above, or below?* Both indicated that working with a similarly skilled individual was the best arrangement. Neither indicated interest in working with a lesser-skilled programmer, but both were more positive about working with an experienced individual. In this case, both felt that personal compatibility would determine whether the arrangement succeeded.

4. *Did the fact that you were working with someone else make you work harder than you might have otherwise?* Neither programmer felt that collaboration was much different than working independently on a group project. As with most TCMS software development, the schedule was the primary driver. *ProgB* stated that participation in "an experiment" might have added additional pressure to perform. This was an unintentional, and perhaps unfortunate effect.

5. *Do you think you learned more about GUIs working together than you would have individually?* The pair felt this was true, particularly in the way they researched implementation details or overcame problems. Instead of immediately asking a more experienced programmer for help, they were more likely to first consult technical manuals or review existing applications for solutions. They sometimes worked individually to resolve issues, each searching in a different area to find answers.

6. *Do you think you learned more about TCMS and the flight hardware (i.e., the environment) than you would have individually?* The programmers did not feel that this was true, and were unable to cite any examples where any significant environmental knowledge had taken place. As *ProgB* said, "This (GUI development) was pretty much the same thing we have done before, only harder." On the other hand, *ProgA* noted that the pair worked more closely with the user than either had during past (individual) assignments. This may have indirectly caused greater environmental learning to take place.

Chapter 6: Conclusions and Recommendations

6.1 Experiment Conclusions

Prior to beginning the pair programming experiment, a series of hypotheses were made as to the outcome. Following is a review of the hypotheses, and how well each stood up to experimentation.

1. *The pair will require more "man-hours" to produce the assignment than would a sole programmer of a similar experience level.*

   This was found to be true. The level-of-effort for an individual was estimated to be 124 man-hours; the pair programmers required 170 man-hours to complete the assignment. Taking into account that 85 of the pair hours were spent working together (42.5 man-hours each), 127.5 "clock" hours were required to produce the three GUIs, which is greater than the estimate for an individual. This is contrary to initial expectations based on the research of others, which indicated that the clock hours for a pair would be somewhat less than for an individual. There are several possible reasons for the results:

- The level of effort for an individual to perform the task was underestimated. As stated in Chapter 3, the App SW group estimates are only accurate within 15% of actual man-hours worked about 80% of the time. For the pair, the actual man-hours recorded (170) were 14% lower than the estimate of 198 man-hours. Since there was no parallel effort for an individual to produce the GUIs, it is not known how many man-hours one novice programmer would have actually required to perform the task.
- The programmers did not pair full time. In previous documented experiments, comparison was made between individuals and programmers who paired continuously. In this experiment, the programmers were given the choice as to the degree of collaboration, and about half the time worked alone. Separate schedules made some individual work unavoidable, but personal preference was also a factor.
- Timekeeping was not precise. The pair usually updated the time logs only once per day, so the recorded hours were based on recollection. It is known that on a few occasions, development time was not recorded until the following day. In addition to the GUI assignment, both programmers had other daily activities to perform, which may have contributed to imprecision. At best, the “actual” time recorded for this experiment should be considered a fairly accurate estimate.

2. *The artifacts produced by the pair will contain fewer defects than anticipated for an individual.*

   This was also found to be true. The defect reduction was not only lower than that estimated for an individual, but was lower than estimated for a pair based on the results of other experiments. This could have been the direct result of collaboration, but also could have also been because the expected defects were overestimated. In past assignments, each of the pair had primarily worked on GUIs which had many data fields but little underlying logic; the GUIs of this experiment had relatively fewer data fields with a considerable amount of underlying logic. The defect estimates were based on the belief that the increased GUI logic requirements would lead to an increased number of errors. The pair may have been more adept at GUI logic than anticipated.

3. *Inexperienced programmers, working in pairs, will develop technical and environmental knowledge more quickly, increasing skill levels (and thereby rate of productivity) faster than is normal for new hires.*

   Observation, as well as comments from the programmers, indicates this hypothesis to be at least partially true. The pair required less "skill oriented" mentoring than expected, demonstrating willingness to research problems together instead of involving an experienced third party programmer. The low number of defects found during testing is another possible indication of technical knowledge gains. In retrospect, the development of GUIs may not have been an adequate assignment to test for increased environmental knowledge - there was simply no need for it. However, the fact that the programmers, as a team, were more willing to initiate discussions with the system engineer (user) indicates that greater learning in this area could have taken place, were it required.

4. *More accurate time estimates can be made for application software development undertaken by inexperienced programmers working pairs, providing better overall project planning.*

The data obtained in this experiment provide a starting point for making future estimates more accurate. A difficulty appears to be that when the programmers are left to decide the degree of collaboration, the results may be difficult to predict. Total development hours may have been quite different if the pair had collaborated a greater percentage of the time. Again, this is a difference between a controlled experiment and the particular environment of TCMS software development, where flexible work hours, multiple responsibilities, and a tight software delivery schedule may impact the collaborative effort.

5. *Inexperienced programmers working within the pair programming framework can be given more difficult assignments than previously thought, increasing the overall productivity of the organization.*

   The experiment provided encouraging results for this hypothesis. The willingness of the pair to work without close mentoring was an unexpected benefit. Most technical conversations with them were not to explain how something should be done, but to give approval to what they had already done. This was not a trivial assignment, and they performed it well. Their greater independence allowed the mentor to concentrate on other tasks, with the result being that all three participants were more productive.

In summary, the novice-novice pairing experiment should be considered a qualified success. It indicates that collaboration will reduce the defects found in work products, and increases novice skill level while reducing mentor responsibilities. However, without a large percentage of development time spent collaboratively, the man-hours required for pair programming may unavoidably be greater than that required by an individual. In such a case, it would be a matter of debate as to whether the benefits justify the additional expenditure.

6.2 Review of Experiment Implementation

Following analysis of the pair programming results, it became apparent that certain changes to the experiment implementation could have lead to more meaningful data being gathered. Given another opportunity to perform the experiment, the following concerns with the original implementation would be addressed:

1. This study was modest in scale and complexity, attempting to draw conclusions on the benefit of novice-novice pair programming by developing three GUIs over a period of approximately one month. A more challenging assignment, over a longer period of time, could have yielding additional data for analysis.

2. A significant amount of GUI "ownership" occurred during the experiment, perhaps too much to accurately study the benefits of pair programming. A certain amount of ownership is not considered detrimental. Dick Gabriel, creator of Common Lisp, discusses pairing not in terms of equal ownership, but of software modules being written by "an owner and a buddy [26]." However, observations during this experiment indicated that the amount of time spent working collaboratively might have been insufficient. Possible warning signs included:
   - Only one of the GUIs developed was a joint effort from the outset. The other two displays were conceptually designed by individuals.

- One of the programmers, *ProgA*, was a more dominant personality and perhaps took too much ownership in the overall development effort. The completed GUIs looked much more like *ProgA*'s initial design than *ProgB*'s. This may be common in most any pair programming effort, or it may be an indicator of unequal contribution.
- *ProgA* logged more hours on the project than *ProgB*. Was this unavoidable, or a sign that additional mentoring on pair responsibilities was needed?

3. More accurate time records could have been kept. Historically, the App SW group has avoided continuous task-time recording, preferring to log time charges at the end of the work day. Precise accounting of man-hour expenditures, for both individual and pair programming efforts, could remove any doubt regarding the validity of this data.

6.3 Recommendations for Future Research

Observations during the thesis experiment indicated several potential benefits of novice-novice pairing. The programmers seemed to enjoy the experience, felt that their technical skills had been increased, and appeared to require less mentoring. However, observation alone is insufficient to validate the reported benefits of this programming style. Future studies of pair programming with novices could expand the scope of this research effort, with emphasis on increased time spent in collaboration, more challenging assignments, and additional quantitative data gathered. Some specific suggestions for future research:

1. In a professional environment, budget constraints would likely not allow an individual and a pair to produce the exact same software application. However, large development efforts do occur which require multiple applications of a similar nature. Assigning some of these applications to novice pairs, and others to novice individual programmers, could provide a basis for direct comparison between the two programming styles.

2. This thesis did not focus on environmental knowledge advancement of the novice programmers. For TCMS applications, code development requires more knowledge of the environment (checkout system, payload, etc.) than does GUI development. Future comparative assignments could involve both GUI and code development, for applications pre-selected for the significant environmental knowledge required to produce them.

3. Additional metrics could be taken to determine environmental skill level advancement by the new-hires. Simple tests could include:
    - The acronym test. One of the greatest intimidators of novices is “NASA-ese”, an acronym-laden language impossible to understand by

outsiders. Pairs and individuals could be tested, before and after an assignment, on the meaning of a selected list of acronyms.

- The customer test. Novice programmers are intimidated by the thought of initiating meetings with system engineers. Even after considerable time with an organization, some novices do not know who their users are. It could be interesting to see if pair programmers learned more user names, and user responsibilities, than did individual programmers. Additional metrics for customer knowledge could include an accurate count of the number of times individual and collaborative programmers initiated technical conversations with the users.

4. This thesis did not focus on technical skill level advancement, and additional metric data could be gathered to measure gains. The idea of taking “pre-tests” and “post-tests” for technical knowledge may be threatening to novice programmers or, for that matter, to any professional. However, indirect measurements could be taken, such as an accurate count of novice programmer requests for assistance, for both individuals and pairs. Fewer questions over time could indicate that more technical knowledge is being gained.

5. In the thesis experiment, the participants were allowed to choose their own degree of collaboration - anywhere from 33% to 100%. Records indicated that

collaboration only occurred about 50% of the time, and this may have reduced the benefits of pairing. Future research with novice programmers could raise the lower boundary percentage of required pair programming. This could determine the minimum degree of collaboration necessary for pair programming to demonstrate development time advantages when compared to individual effort.

Bibliography

1 K. Beck, *Extreme Programming Explained*, Reading, MA: Addison-Wesley: NY, 2000

2 W. Bevan and C. McDowell, "Guidelines for the Use of Pair Programming in a Freshman Programming Class," *Proceedings of the 15th Conference on Software Engineering Education and Training*, 2002
Online at:
http://ieeexplore.ieee.org, Search: (pair programming) + (guidelines)

3 A. Cockburn and L. Williams, "The Costs and Benefits of Pair Programming," *Humans and Technology Technical Report, 2000.01*, January 2000
Presented at eXtreme Programming and Flexible Processes in Software Engineering - XP2000, Cagliari, Sardinia, Italy, 2000
Online at:
http://www.cs.utah.edu/~lwilliam/Papers/XPSardinia.pdf

4 Forum, "Extreme Programming Message Forum," *Software Reality*, Oct. 2002.
Online at:
http://www.softwarereality.com/lifecycle/xp/forum.jsp#id56

5 Forum, "Pair Programming Successes and Failures," *The Joel on Software Forum*, December 2001
Online at:
http://discuss.fogcreek.com/joelonsoftware/default.asp?cmd=show&ixPost=2058

6 J. Grenning, "Launching Extreme Programming at a Process-Intensive Company," IEEE Software, Vol. 18, No. 6, pp. 27-33, Nov./Dec. 2001
Online at:
http://www.objectmentor.com/resources/articles/XP-In-Process-Intensive-Company-IEEE.pdf

7 J. Haungs, "Pair Programming on the C3 Project," *Computer*, vol. 34, pp. 118-119, Feb. 2001
Online at:
http://ieeexplore.ieee.org, Search: (pair programming) + (project)

8 D. Howe, *The Free Online Dictionary of Computing*.
Online at:
http://foldoc.doc.ic.ac.uk

9 C. Jones, "Gaps in Programming Education," *Computer*, Vol. 28, No. 4, pp. 70-71, April 1995
Online at:
http://ieeexplore.ieee.org, Search: (pair programming) + (education)

10 R. Kay, "QuickStudy: System Development Life Cycle," Computerworld, May 14, 2002
Online at:
http://www.computerworld.com/developmenttopics/development/story/0,10801,71151,00.html

11 J. Kivi, D. Haydon, J. Hayes, R. Schneider, and G. Succi, "Extreme Programming: A University Team Design Experience," *2000 Canadian Conference on Electrical and Computer Engineering*, vol. 2, pp. 816-820, Mar. 2000
Online at:
http://ieeexplore.ieee.org, Search: (extreme programming) + (design)

12 R. Moore, "Evolving to a "Lighter" Software Process: A Case Study," *Proceedings: 26th Annual NASA Goddard Software Engineering Workshop*, pp. 14-21, 2002
Online at:
http://ieeexplore.ieee.org, Search: (software process)

13 J. Nawrocki and A. Wojciechowski, "Experimental Evaluation of Pair Programming," *Proceedings: European Software Control and Metrics (ESCOM)*, 2001
Online at:
http://www.escom.co.uk/conference2001/papers/nawrocki.pdf

14 J. Newkirk, “Introduction to Agile Processes and Extreme Programming,” *Proceedings: 24th International Conference on Software Engineering*, pp. 695-696, 2000
Online at:
http://ieeexplore.ieee.org, Search: (extreme programming) + (agile)

15 J. T. Nosek, “The Case for Collaborative Programming,” in *Communications of the ACM*, vol. 41 no. 3, 1998

16 D. J. Reifer, “How Good Are Agile Methods?,” *IEEE Software*, Vol. 19, No. 4, pp. 16-18, July/Aug. 2002
Online at:
http://ieeexplore.ieee.org, Search: (agile methods)

17 M. Stephens, “The Case Against Extreme Programming: Key Rules and Tenets of XP,” *Software Reality Website*, Feb. 2002.
Online at:
http://www.softwarereality.com/lifecycle/xp/key_rules.jsp

18 G. Succi, M. Stefanovic, and W. Pedrycz, “Quantitative Assessment of Extreme Programming Practices,” Canadian Conference on Electrical and Computer Engineering, Vol. 1, pp. 81-86, 2001
Online at:
http://ieeexplore.ieee.org, Search: (extreme programming) + (quantitative)

19 A. Van Deursen, “Program Comprehension Risks and Opportunities in Extreme Programming,” Proceedings: Eighth Working Conference on Reverse Engineering, pp. 176-185, 2001
Online at:
http://ieeexplore.ieee.org, Search: (extreme programming) + (risks)

20 J. A. Whittaker, *Introduction to Software Engineering*, SES Press: Melbourne, FL, 1998

21 L. A. Williams, “But, Isn’t That Cheating?,” *FIE’99 Frontiers In Education. 29th Annual Frontiers in Education Conference*, vol. 2, pp. 12B9/26-27, Nov. 1999
Online at:
http://ieeexplore.ieee.org, Search: (pair programming)

22 L. A. Williams, “Integrating Pair Programming into a Software Development Process,” *Proceedings: 14th Conference on Software Engineering Education and Training*, pp. 27-36, Feb. 2001
Online at:
http://ieeexplore.ieee.org, Search: (pair programming) + (integrating)

23 L.A. Williams and R.R. Kessler, “All I Really Need to Know about Pair Programming I Learned in Kindergarten,” *Communications of the ACM*, Vol. 43, No. 5, 2000, May 2000, pp. 108-114
Online at:
http://collaboration.csc.ncsu.edu/laurie/Papers/Kindergarten.pdf

24 L. A. Williams and R. R. Kessler, “The Effects of “Pair-Pressure” and “Pair-Learning” on Software Engineering Education,” *13th Conference on Software Engineering Education and Training*, pp. 59-65, March 2000
Online at:
http://www.cs.utah.edu/~lwimmiam/Papers/CSEET.pdf

25 L. A. Williams and R. R. Kessler, “Experimenting with Industry’s “Pair-Programming” Model in the Computer Science Classroom,” *Journal of Computer Science Education*, March 2001
Online at:
http://collaboration.csc.ncsu.edu/laurie/Papers/CSED.pdf

26 L. A. Williams and R. R. Kessler, “Pair Programming Illuminated,” Addison-Wesley, 2002

27 L.A. Williams and R.R. Kessler, “Strengthening the Case for Pair Programming,” IEEE Software, Vol. 17, July/August 2000
Online at:
http://www.computer.org/software/so2000/pdf/s4019.pdf

28 L. A. Williams, A. Shukla, and A. I. Anton, “Pair Programming and the Factors Affecting Brooks’ Law,” North Carolina State University, Awaiting Publication, 2002

29 L. A. Williams and R. Upchurch, “Extreme Programming for Software Engineering Education?,” *31st ASEE/IEEE Frontiers in Education Conference*, pp. T2D12-17, Oct. 2001
Online at:
http://ieeexplore.ieee.org, Search: (extreme programming) + (education)

Appendix: Experiment Journal

09/27/02
D.E. (App SW manager) and I discussed the possibility of having two novice programmers within the group work on an application using the pair programming methodology. I gave D.E. two pair programming articles: *The Costs and Benefits of Pair Programming* (which supports the concept), and *Experimental Evaluation of Pair Programming* (which questions the benefits of pair programming). D.E. agreed to the pair programming assignment, providing the application is released by the user need date. He also wants to see the level-of-effort estimates when I complete them.

09/30/02
Discussed Pair Programming concepts with *ProgA* and *ProgB*, and asked if they would like to participate in an experiment. They were given a couple of days to think it over, but have already agreed. I gave them one article on the topic, *Strengthening the Case for Pair Programming*. We discussed the overall theme of the article, and I encouraged them to read it (on the other hand, it isn't "homework"). They'll do all displays (GUIs), as well as some of the code (how much code hasn't yet been determined). The programmers wanted to know if I "need" this to succeed (!) The answer is no - whatever happens, happens. But, on the other hand, we would like to try to make it work (as we try to make any development effort work). I told them that we would abandon the pairing style if it just wasn't working out. We also discussed the time logs, with the main idea being to keep them simple - I don't want to make this study painful for them.

10/01/02
Met with *ProgA* and *ProgB* again. I told them there will be no overtime available for this project (they had asked this question earlier). They are to essentially decide on the time spent in actual pairing, but we need at least 1/3 of their time spent in collaboration, 50% would be even better, and 100% would be great. They asked if this is the pilot project for permanent changes within the group. No, I told them, this is currently only a small experiment. The pair seems comfortable with participating in this project, but on the other hand, they certainly aren't overly enthusiastic. My concern is that they think this assignment is mandatory. We discussed that a little, and I tried to emphasize that this was a completely voluntary effort.

10/02/02
I have reviewed the application requirements (again), and they look complete. The user has provided one GUI drawing, but we know that at least two other displays will be required. A plan of attack for this development effort is beginning to come together.

I met with the two programmers again, and they're eager to begin work (although I'm still not sure how eager they are to begin working *together*). *ProgA* and *ProgB* don't really have any other questions, so we mostly just shot the breeze. They did ask about the time logs - how often to update them (Hourly? Daily? Weekly?). The idea is to update the time logs at least once a day. I emphasized "at least" - if they update the logs more than once a day, great - the numbers will probably be more accurate. I will look the time logs over on a weekly basis, or maybe daily at first.

10/03/02
An application kickoff meeting was held with the pair programmers. To start, they'll work on the GUI prototypes, while I get the code framework functional. We spent considerable time discussing GUI dynamics (pseudocode which drives active objects). The fields, buttons, and objects relating to the GSE power supply will have a different functionality depending on the operational environment and whether a Primary or Secondary power supply is selected. Related GUI items must "know" the environment/power source settings to work correctly. We also discussed the Application State (AppState) indicator, which essentially drives the MPLM (MultiPurpose Pressurized Logistics Module) automated activation and deactivation. Ultimately, everything is based on the AppState indicator state.

ProgA asked if they should develop display prototypes one at a time or concurrently. Basically, I don't care. Is there a benefit to concurrent GUI prototype development? I'll let them come up with a plan and go with it - they just need to keep me informed. *ProgA* asked if they should keep defect data during all development effort. No, we'll just track that data as has been done in the past, during functional testing. I told the pair that we must have the prototypes to M.D. (the user) by next week for comments. Not only will this allow us to firm up the GUIs, it will also help decide which specific code modules we need to develop.

10/04/02
Of the three application GUIs, the pair programmers want to work on one GUI each (primarily) and the third one together. I wasn't expecting this, although it should

be okay - I guess - as long as they keep frequent peer reviews going. I reminded them of the guideline that they must spend at least 1/3 of their time together (and I was hoping for more). *ProgB* says "We work different hours, we'll be able to keep going without stepping on each other's toes." Okay, it probably makes sense to do this.

Their plan is to get the GUI prototypes approved, then incrementally add functionality which will more or less map to the concurrent code development I do. They will attack the most complex GUI constructs last. This is fine with me, but I reminded them that time is of the essence. At this point, I don't know how to implement some of the complex GUI functionality (I'm not a GUI expert). I can only guess at the time they would require to implement some of the more difficult functionality. When they get around to tackling the complex GUI items, I encouraged the pair to try to do it themselves, but to feel free to get with G.M. (our GUI expert) if they get stuck.

10/08/02
Met with the pair programmers to get a status on the GUI prototypes. They are looking pretty good, although still rough around the edges. *ProgA* and *ProgB* have primarily been working on the individually owned displays, with less work done (so far) on the joint display. Are they working together or not? *ProgA* says that on the individual GUIs they have, "a few minutes at a time," primarily when they have problems. This solo work shows in the individual displays, which have different fonts, and different button sizes. We talked about the need for similar "look and feel" for all the GUIs. The programmers need to come to an agreement on this. I suggested a couple of particular applications, already released, that they may want to look to as guides. On the other hand, I'd like to see what they come up with on their own (after all, they've done GUIs in the past)...

The joint display looks good. It is interesting that this one looks more like *ProgA*'s individual display than *ProgB*'s display They have paired for as long as two hours at a stretch on the joint display - I was hoping for more collaboration. If they don't start pairing more, we'll discuss the need for increased teaming after the prototyping is finished. Each of the individual displays need about one more day each, while the joint display needs about two days. We should be able to meet with user by the end of the week for a prototype review.

10/11/02
The pair and I met with the user to discuss GUI prototypes. Overall, he likes them. He requested a few fields to be moved around, some minor button renaming, and

would like certain “critical” data fields to be on each display. The user also wants the Front-End Processor (FEP - TCMS box which interfaces with the flight element) fields to not only show all FEPs, but to identify the “operational” FEP, based on two different FEP state values. We will need to evaluate two data points for each FEP, and only one FEP will have the right data values to be “operational”. The user also wants a manual override to allow user selection of a different operational FEP. As it turns out, he had tried to specify this in the requirements, but I didn’t fully realize what he had been asking for. After the meeting, the pair programmers and I discussed the FEP data fields and we came up with a modified implementation plan. Fortunately, we won’t have to undo any work, as the GUIs are still in prototype stage.

<u>Note</u>: The pair programmers have been working more closely together over the last couple of days. I hope the trend continues. These two individuals were definitely the right ones for this experiment. I <u>think</u> they’re concentrating on this being simply a software development assignment instead of an “experiment”. Good - that’s probably the way it should be. They both seem to work more productively when the pressure’s on a little. I will see if I can identify evidence of the “pair pressure” mentioned in the pair programming articles.

10/14/02
Per Thesis Advisor suggestion, I have posted a question on an online XP forum, asking for anecdotal experiences of others regarding novice-novice pair programming.

http://www.softwarereality.com/lifecycle/xp/forum.jsp

This was the closest forum I could find to maybe get a response, but it may just be a gripe group. There are lots of negative opinions about XP in the forum, and just a little information about pair programming. I’m not sure if anyone will be interested in the pair programming topic, but maybe someone will answer.

I met with the programmers to discuss the user’s comments on the prototypes, and to see how the experiment was going. They have incorporated all the GUI layout changes requested by the user. Additionally, they came up with a different solution to the “operational FEP” implementation - looks good, <u>better</u> than mine. Was this a situation where pairing led them to a solution that neither would find alone? Direct quote: “Maybe.” Not exactly a ringing endorsement, but not a denial, either.

10/16/02
No response on the XP forum to posted question regarding: others pair programming experiences. Either this is a dead forum or nobody who reads it has anything to say. (Maybe since I didn't post to gripe about something, I'm off-topic?) Per advisor suggestion, e-mailed Laurie Williams. She's the author of many of the pair programming articles found, and also wrote the *Pair Programming Illuminated* textbook.

M.D. (user) told me that they need the application by the first of December, not the first of January as originally planned. Surprise! Things around here usually slip to the right, not to the left. This seriously impacts the plans for this experiment. I had wanted the pair to do some of the coding, but that is no longer a good idea. E-mailed advisor to okay PP experiment doing just the GUIs, with me handling all the software. The advisor approved the change. There should be no harm in changing the scope of the assignment - the pair have only been working the GUIs up to now, so no work (or experiment observations) will have to be discarded.

10/17/02
Laurie Williams (pair programming guru) e-mailed me back. She sent an article on pair programming she's preparing for publication. The article is titled *Pair Programming and the Factors Affecting Brooks' Law.* Brooks' Law states "adding manpower to a late software project makes it later." Not surprisingly, Williams refutes this proposal. Lots and lots of math and statistics in the article. The subject matter isn't precisely related to novice-novice pairings, but it does discuss training, mentoring, and other topics that at least indirectly relate to our experiment.

10/21/02
Met with the pair programmers for status and miscellaneous chat. Status: things very on-track. We went over the schedule, and I emphasized that we need to get the GUIs finished by the end of the month.

I asked the pair to try to think about any positive or negative aspects of pair programming. *ProgA* stated, "We probably should have been working together more on the [independently-owned] displays. If we had, we wouldn't have had to change as many things." Good, good, good, good. *ProgB* said that "[*ProgA*] caught a few errors" that *ProgB* had made while in the "driver" mode, and added, "Sometimes we get a lot done." I guess that means that sometimes they don't get very much done? As for negative aspects, *ProgB* said, "[When we sit at the computer together] people think we're just goofing off. We get a lot of interruptions." This is partially my fault; I haven't widely advertised the pair programming experiment. Maybe people think they're goofing off because they're having a good time?

The pair also say that a benefit of having a driver/navigator setup is in catching PUI input errors. (PUIs are Program Unique Identifiers, 13-character alphanumeric identifiers of individual telemetry data items.) PUIs are on a spreadsheet, many named similarly, and it's easy to either be on the wrong Excel row, or to misread the PUI, or to type it in wrong. Some of these errors probably would have been caught when using Edit MPS (a debug tool) in local mode, but that would only work if (1) the PUIs did not exist, or (2) the programmers read the PUI correctly from the spreadsheet when testing. If they had tested "valid but wrong" PUIs by looking at the display dynamics, they wouldn't have seen it....they would nave continued to test the wrong, but valid, PUI.

10/28/02
Met with the pair programmers for weekly conversation. They are essentially finished with the GUIs, and should be able to start functional testing late this week. In previous meetings, the pair had been asked to (more or less) say whatever came to mind regarding the experiment. However, for this meeting I wanted to ask more direct questions, and had a list prepared:

1. Was this overall a good experience for you, or a bad experience? Both *ProgA* and *ProgB* indicated the experience was positive. *ProgB* said he was worried at first about how badly I wanted this to work out.

2. Would you be willing to pair with someone else for another experiment in the future? *ProgA* was hesitant about this - he named certain individuals in the group he would pair with, and others he'd prefer to avoid. He named more people he wanted to avoid than people he wanted to work with. *ProgB* echoed *ProgA's* sentiments on two individuals, but generally was more open about working with someone else.

3. If you paired again with somebody else, would you prefer to work with someone at your skill level, or above, or below? Both indicated a preference for working with someone at their same skill level. They definitely didn't want to work with a lesser-skilled programmer, and indicated tentative interest in working with a higher-skilled person, depending on who it was.

4. Did the fact that you were working with someone else make you work harder than you might have otherwise? This was a loaded question - who wants to admit they don't always work hard? We discussed the idea of "pair pressure".

ProgA said, "We need to get done by the end of the month." *ProgB* said, "You need to get your thesis done." (Ugh. Pressure was there, but because of schedule and the fact I was writing a thesis?)

5. Do you think you learned more about displays (GUIs) working together than you would have individually? Both programmers said this was true. When they came up against a problem, sometimes they would research it together, sometimes they would split up, with one researching the documentation, and the other looking in previously released applications for GUIs which contained similar features. *ProgB* said that they used the documentation more for this project, while in the past they would immediately go to someone else for help. *ProgA* said "When we were working with [the GUI expert], I didn't really get everything he was saying, but after he left, [*ProgB*] and I talked about it, and I understood."

6. Do you think you learned more about TCMS and the flight hardware than you would have individually? Both said they learned more, but were unsure if it was because they were working together or because of the nature of the application. However, when asked to list specific things they had learned, they really couldn't - so perhaps not much environmental learning happened after all. One of the pair pointed out that since they didn't do the code, they didn't have to get as involved (learning the environment) as they might have otherwise. On the other hand, *ProgA* said, "We did talk more with M.D. (the user) more than I had with users in the past to understand what he wanted....we usually got with him together."

10/29/02
GUIs complete.

10/31/02:
GUI functional test.

GUI errors:

Requirements Nonconformance:
LOCK box on monitor display green all the time? At least, once it's green, it doesn't go off. Keying on the wrong value - looking at T0 power supply voltage, should be looking at MDM lock status.

Logic Error:

Submodel logic for T0 Voltage and Current fields incorrect - if value not within -4 or +6 of setpoint, nothing displays. Should be: if value not within -4 or +6 of setpoint, value displays in red.

11/01/02:
GUI functional test continues.

GUI errors:
Logic Error:
Environment variable should default to "SSPF ?", but defaults to "SSPF".

Syntax Error:
Incorrect PUI for Cabin Fan Assembly Telemetry Valid, used RPC 11 Telemetry Valid.

11/04/02:
GUI functional test continues.

GUI errors:
No errors found.

End of functional test.

11/06/02
Per thesis advisor suggestion, I had written Laurie Williams asking if she knew of any University of Utah students who had (1) participated in her 1999 experiment and (2) had subsequently written a research paper, thesis, etc., on the topic of pair programming. Today, I received a response from Prof. (Dr?) Williams. She didn't answer the question! All she said was that she had done her "own dissertation" at Utah on the Collaborative Software Process (CSP) which, I'm pretty sure, she invented based on a combination of PSP and pair programming. Well, duh - I *know* she did her own work. Did she think I was implying her students wrote it? (Hmmmm...) I will write the University of Utah library to see if they can help.

11/08/02
I wrote the University of Utah library, posing basically same question as I did to Laurie Williams, asking for help in finding research papers written on pair

programming (or CSP). The web page indicates they should respond within a couple of days.

Someone wrote back to the question I had posted on the XP forum! A programmer (Ravi) states the XP worked pretty well in his organization, although they were forced into it by a lack of computers...it sounds like they abandoned pairing when more hardware was available. This gets included in my paper....at least I got *one* response.

The University of Utah library responded - same day. They don't keep track of papers written by faculty or students (!). I guess I don't understand these things, because I find that surprising (although the thesis advisor predicted it). The library suggested that I look elsewhere...I'm not sure if there is any big benefit in pursuing this.

01/08/03 - *Epilogue*
At the conclusion of the pair programming experiment, *ProgA* and *ProgB* moved on to other assignments. *ProgA* is still within the App SW group, currently part of a team updating an existing application, and is responsible for all GUI modifications and some of the code. *ProgB* is on assignment to another group, primarily involved in Research and Development. Approximately two months after the end of the pair programming effort, the two were asked to reflect on the experience and whether collaboration might be beneficial in their current work.

In retrospect, both programmers felt the most beneficial aspect of the experiment was not found while sitting at the keyboard, but in the design work done prior to implementation. *ProgA* acknowledged the reduction in syntax errors while building the GUIs, but added "Our setup (detailed design) time was longer - deciding how to do something. But once we got things figured out, the keyboard work was faster. And we had a much better design." *ProgA* believes pair programming could be beneficial for any project in the App SW group. On the other hand, *ProgB* feels that in the R&D organization, "It will be difficult to use pair programming to code here since there are no clear requirements to go by. When [*ProgA*] and I were working together, we knew what we had to do, and it was just a matter of getting it done correctly. That's why pair programming worked for us."

It is interesting to note *ProgB's* impression that without established requirements, pair programming would be problematic. In reality, published articles state that collaboration can be beneficial during the requirements definition phase. For the thesis experiment, the application requirements had already been established before the pair programming effort began - this is most likely what led to *ProgB's* comments.

Both programmers stated that now that they are familiar with the techniques of collaboration, they tend to notice when those practices are used by others. Even though *ProgB* believes pair *coding* might not work in the R&D assignment, "...I see it (pairing) used during the planning and troubleshooting phases of the project." And according to *ProgA*, pair programming is "Basically, just people working together. We do that all the time here."