UNIVERSITY OF CALGARY

An Exploratory Longitudinal Case Study of Agile Methods in a Small Software Company

by

Christopher Richard Mann

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

AUGUST, 2005

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "An Exploratory Longitudinal Case Study of Agile Methods in a Small Software Company" submitted by Christopher Richard Mann in partial fulfilment of the requirements of the degree of Master of Science.

Supervisor, Dr. Frank Oliver Maurer, Department of Computer Science

Dr. Ehud Sharlin, Department of Computer Science

Mr. Ron Murch, External Examiner, Haskayne School of Business

Date

Abstract

Agile methodologies, once considered new and novel are now crossing from early adopting organizations to wider mainstream use in the software industry. As agile methods become mainstream, there is a noticeable lack of empirical information on agile methodologies in industry. Much of the evidence of agile methods in industry is anecdotal, coming from experience reports. Of the few empirical studies that have been conducted in industry most focus on the short-term use of agile methods – leaving a gap in knowledge about agile methods over long-term use. One of the goals for this thesis is to start to fill this gap. This is accomplished by conducting a longitudinal case study exploring the introduction and use of agile methods such as Scrum, pair programming, and test-driven development with continuous integration in a small company.

Acknowledgements

This thesis would not have been possible without the support and effort of many people over the last two years.

First, I would like to thank my parents for their many hours of editing and supporting me throughout all these years.

I would like to thank my supervisor, Frank Maurer, for his support, and editing of this thesis and other work during the last two years

I would also like to thank my research group colleagues. First, I would like to thank Lawrence Liu, Harpreet Bajwa, and Wen Liang Xiong who helped me throughout my course work. I would also thank Carman Zannier who provided valuable feedback on this thesis.

I would like to thank Erin Morrison for her assistance with my graphs and tables.

I would like to thank Dr. Lu Xuewen for assisting me with my statistics

Much appreciation goes towards the developers and other participants at PetroSleuth Inc. who participated in the case study.

Finally, I would like to thank PetroSleuth Inc. and the Natural Sciences and Engineering Research Council of Canada for supporting this project

Dedication

To my parents and brothers

Table of Contents

List of Tables

List of Figures

List Abbreviations

Build: Software Build Process

CI: Continuous Integration

DSDM: Dynamic Systems Development Method

FDD: Feature Driven Development

GUI: Graphical User Interface

MCSE: Microsoft Certified Systems Engineer

PMP: PMP Certification

QA: Quality Assurance

TDD: Test Driven Development

XP: Extreme Programming

Publications

Materials, ideas, and figures from this thesis have in part appeared previously in the following publication:

CHAPTER ONE: INTRODUCTION

Agile methodologies, once considered new and novel are now crossing from early adopting organizations to wider mainstream use in the software industry. Agile methodologies are a set of methods that encourage interaction and communication between developers and customers, working software, and the ability to adapt and change in environments with volatile requirements or priorities (Manifesto 2005). As agile methods become mainstream, there is a noticeable lack of empirical information on agile methodologies in industry. Much of the evidence of agile methods in industry is anecdotal, coming from experience reports. There are two main problems with experience reports. First, experience reports, while usually providing anecdotal evidence do not often provide much in the way of deeper qualitative or quantitative information Second, experience reports do not provide detailed context for the anecdotes which makes it hard to evaluate if their results can be transferred to other environments. Of the few empirical studies that have been conducted in industry most focus on the short-term use of agile methods – leaving a gap in knowledge about agile methods over long-term use. One of the goals for this thesis is to start to fill in this gap by conducting a longitudinal case study at PetroSleuth Inc on the long-term impact of agile methods at a small company.

PetroSleuth Inc is a small software development company of approximately 4 to 7 software developers. They create custom windows and web applications for the oil and gas industry. I was employed at PetroSleuth Inc for four summers prior to starting this study. During this time, I was developing software and gained a number of insights about the company. First, there are many different solutions to a single problem in the oil and gas industry and the solution to the problem may change often as one solution becomes more desirable over another solution. The choice of a solution is highly subjective and is likely different between two experienced practitioners. The results are that requirements change often, depending on who is providing the requirements. At PetroSleuth, there were many differing viewpoints leading to very fluid requirements and a sense of chaos about what was to be developed, how long it was going to take, and when was the

functionality expected. This lack of control and visibility of requirements and timelines lead to releases being shifted and developers working substantial amounts of overtime. Scrum (Schwaber 2004) was suggested as a way to control both the fluid requirements and provide a more stable work environment.

A second problem observed at PetroSleuth was, at a small company the loss of any developer causes problems with continued software development as each developer was the expert on their code section and others were not familiar with this code. An example of the problems faced due to having these specialized roles happened while I was working at the company in the summer of 2003. On one occasion, the software was not working and the customer group needed the report that day as it was going out to a client. The developer responsible for the report did not come to work for most of that day. Myself and another developer attempted to figure out the problem but after half a day of work, we had no idea what was wrong. The problem we faced was the reporting code was complex and we were not familiar with the steps involved with creating the report and therefore had to debug while also figuring out what the code did. Later in the afternoon, the developer showed up and fixed the problem in 5 minutes. From this experience and others similar to it, the need to have all of the developers' familiar with the system became evident. Pair programming was introduced as part of the study as a way to reduce the risk of loosing important system knowledge when a developer is on vacation or has left the company.

A third problem encountered at PetroSleuth was a lack of testing for the applications being developed. Usually half an hour before the customers were to be given a copy of the program one of the developers would run the application and click buttons and windows for a couple of minutes until he was satisfied that the application worked or crashed. The results of this last minute testing were many bugs seemed to re-occur in releases after they were fixed and much of the program could not be executed in the testing time period. Therefore, automated unit testing and continuous integration was introduced as part of the study to provide more consistent and continual testing for the applications being developed.

1.1 Research Goals

Based on the motivations and problems discussed above, the overall goal of this thesis is to:

Explore the introduction and use of Scrum, pair programming, and test-driven development with continuous integration over the long term in the context of a small company environment.

As a result of this goal, I will provide a detailed case study of the introduction and use of Scrum, pair programming, and test-driven development with continuous integration over the long term. I will also formulate several hypotheses that are based on the empirical results and that can be empirically tested in future studies.

1.2 Structure of the Thesis

Following the introduction, the thesis takes the following form: Chapter 2 will provide some background knowledge about the specific agile practices and methods that were introduced and used at the company (PetroSleuth Inc.). The related work will focus on industrial studies and look at what results the studies have discovered. In Chapter 3, I outline detailed project and company contextual information which are important for interpreting the results and applicability of the results to other settings. Chapter 4 provides a case study overview and timeline. Chapter 5 provides the experiences and results related to the use of Scrum over a period of twelve months at the company. This is followed by the experiences and results related to the use of pair programming over a period of fifteen months at the company described in Chapter 6. Chapter 7 discusses the experiences and results related to the use of test-driven development and continuous integration over a period of eleven months at the company. Finally, Chapter 8 summarized the results from the previous chapters and discusses the limitations of the study. The final chapter also provides suggestions for future work that can be undertaken as a result of the information provided in this thesis.

CHAPTER TWO: RELATED WORK

The purpose of this section is to describe the practices and methodologies used in this thesis. I will present the current state of research for each of the areas discussed below focusing closely on ideas that are presented in rest of the thesis.

2.1 Scrum

The Scrum software methodology was created by Ken Schwaber and Jeff Sutherland (Schwaber 2004). Scrum is based on an iterative and incremental process skeleton.

In Scrum there are three roles:

1. The Team: A cross functional group of people that is responsible for maintaining itself to develop software every sprint.

2. Scrum Master: The person responsible for the Scrum process, its correct implementation, and the maximization of its benefits

3. Product Owner: The person responsible for managing the project backlog so as to maximize the value of the project. The product owner represents all of the stakeholders.

This iterative and incremental process skeleton is arranged into two distinct feed back loops as shown in Figure 2-1.

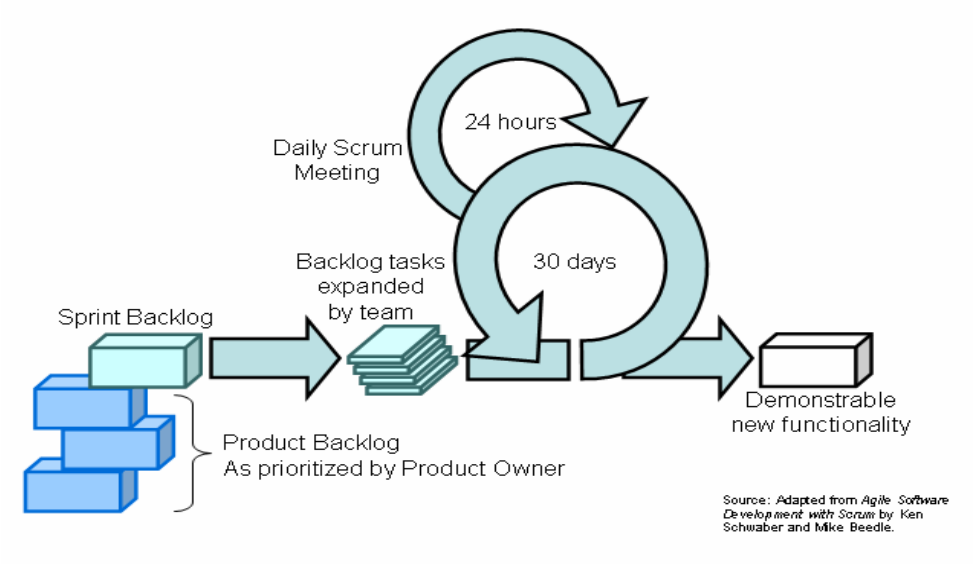


Figure 2-1: Scrum Feed Back Loops, Source: (MountainGoat 2005)

The first feedback, called a sprint, is based on a fixed length cycle (usually 30 days) (Schwaber 2004). The purpose of the sprint is to create potentially shippable functionality. The sprint is initiated by a sprint-planning meeting. During this sprint planning meeting the software developers work with the product owner to create a list of sprint backlog items to be worked on for the length of the sprint. These backlog items are a form of requirement specification that describes in customer language what the developers should create.

After the planning meeting, the Scrum process enters the second feed back loop. This second feed back loop occurs on a daily basis and is called the daily Scrum (Schwaber 2004). These daily Scrums are meetings where the team answers three questions:

- What have you accomplished on the project since the last daily Scrum meeting?
- What do you plan on doing between now and the next daily Scrum meeting?
- What impediments stand in the way of you meeting your commitments to the sprint and this project?

These three questions are used both to update the development team members of what was done and what will be done, and to update any customer representatives as to

the progress of the product and any needs the development team requires of them. As this meeting takes place on a daily basis there is less chance for a problem that would threaten the project to go undetected for long.

At the end of the sprint there are two additional meetings (sprint review and sprint retrospective) (Schwaber 2004). The sprint review is the meeting in which the development team demonstrates to the customers, and other stakeholders, the functionality that has been produced during the sprint. In the meeting, the customers and stakeholders are given the ability to provide feedback about what is being demonstrated. The second meeting is the sprint retrospective. The sprint retrospectives are meetings in which the team looks back on the sprint and asks two questions: What went well during the last sprint? What needs to be improved for the next sprint? These two questions allow the team to reflect back on what they did well and what could be improved.

2.1.1 Scrum Studies

Much of the knowledge about Scrum comes from industrial experience reports such as (Wake 2004; Rising and Janoff 2000; Sutherland 2001) and experiences gathered and published in a Scrum book such as (Schwaber 2004). These reports provide anecdotal evidence and experiences supporting the use of the Scrum process within a variety of company contexts. However, these experience reports and anecdotes are not case studies and do not provide more detailed quantitative or qualitative evidence.

One study providing this qualitative and quantitative evidence was performed by Rautiainen, Vuornos, and Lassenius (2003). For this study Rautiainen and Lassenius were the researchers, while Vuornos was the research and development team lead and process improvement champion. The second author, Vuornos over the course of eleven months introduced a set of Scrum practices into a company. After the Scrum processes were introduced and used for eleven months Rautiainen and Lassenius returned to collect data based on the process improvement. The authors reported the Scrum process initially was not working due to the developers being interrupted and sprint goals being missed. The authors attributed these initial problems to lack of management commitment and that the

new process was not fully understood. The authors noted later in the process, improvement occurred when an extra set of pre-sprint planning was instituted which allowed the process to become more stable. The study indicates that there was an increase in communication between developers and the customers which in the opinion of the study authors clarified the business objectives and the product to be developed and which meant customers had fewer surprises at the end of the sprint. The results presented in this study are very similar to what will be presented in this thesis. Both of these studies found that initially the process had difficulties because the process was new for so many people. In addition, both had problems initially with management's commitment in following the process. Once management committed to the process it went much more smoothly in both our cases. In the study Vuornos, although not a researcher was working at the company and provided insight during the period the researchers were not at the company. In the case of the current research, I was at the company as a participating observer to provide insight and context to the results of the thesis. There is one major difference between the above-referenced Scrum study and this study. The prior work focused heavily on the Scrum process, only briefly mentioning other practices such as automated unit testing. In the current study, data has been collected on Scrum, pair programming and test-driven development and continuous integration. I believed that collecting a larger range of data will give a more complete picture of how agile methods are used in the long term verses the previous study.

2.2 Pair Programming

Pair programming is the concept of having two developers work on one machine with one keyboard and mouse with the intent to develop code in a cooperative manner (Williams and Kessler 2003). Pair programming is more than just programming, it can also be used during design (pair design), for reviewing code (pair review), for testing (pair testing), or for other activities with a partner. All of these activities are considered pair programming (Williams and Kessler 2003). In pair programming there are two roles. The first role is that of the driver. The driver is the person who currently has control of the keyboard and mouse and is in the process of completing a task. The second person is called the navigator. The navigator is sitting beside the driver and is looking at the code

that is being written or has been written. The navigator is looking for errors or problems that may arise from the driver's implementation of the code. For example, an error would be a missing check for a null object or forgetting to implement some part of an algorithm. If the navigator sees a problem or has a suggestion, he can provide his comments to the driver who should take them into consideration when developing the code. An important part of pair programming is sharing the time between the two roles, driving and navigating. The transition from driver to navigator can take place when the driver completes a task, gets stuck on a problem, or an agreed upon time limit expires. At the transition point the driver and navigator switch places. This means that the driver gives the keyboard and mouse to the navigator. This now provides the driver some rest from typing and it also gives them a different perspective when the programming resumes.

2.2.1 Pair Programming Empirical Studies

There have been many studies into pair programming. In 1998, John Nosek performed an experiment to get empirical information on pair programming (Nosek 1998). The task for the programmers was to create a database script that performed a consistency check on a database and output the errors to a log file. The study points out that none of the programmers had worked on a similar problem before. The results from the study were that pairs spent 60% more combined minutes to complete the task than individuals but completed the task 40% faster in elapsed (clock) time. The study also showed that the pairs had more functionality than the individuals (Nosek 1998).

In 1999, a study performed at the University of Utah by Laurie Williams and Robert Kessler wanted to show how well pair programming would work in a classroom environment (Williams and Kessler 2000; Williams et al 2000b). The study participants were comprised of a senior software engineering class that was made up of a mix of junior and senior students. Using the GPA and the student's willingness to pair as the criteria, the class was split into pair groups and individuals. Quality in the study was measured by what percentage of pre-made test cases was passed. When pairs and

individuals were compared, the pairs on average passed 15% more test cases while working 40-50% faster than individuals (Williams at Kessler 2000).

In 2000, at the Poznan University of Technology, a study was performed by Narwocki and Wojciechowski that looked at evaluating pair programming along with other "Extreme Programming Practices" (Nawrocki and Wojciechowski 2001). The extreme practices which were used as part of the evaluation were: pair programming, test centered design, simple solution, risk minimization, and keep moving. Three groups of 4th year computer science majors were used as part of the experiment. The first group used the Personal Software Process (PSP). The second group used the XP (Extreme Programming) practices and the third group was made up of 5 pairs. The results of the study showed that there was no difference in the time it took the XP groups and the individual group to program (Nawrocki and Wojciechowski 2001). In fact, if the sum of the pair hours had been put together, the pairs took almost 100% more time to finish the assignments. This is contrary to the previous studies discussed showing pair programming increased the speed of software development. The number of defects found in the programs was nearly the same for all three groups indicating no benefit for the XP groups (Nawrocki and Wojciechowski 2001). One benefit of pair programming that was found however, was that pairs were more predictable in terms of how much time they were going to take to complete an assignment.

Research has also been done to look at how pair programming can influence the quality of problem solving. A study was done looking at the difference in performance between pair and solo groups solving a variety of problems (Liu and Chan 2003). For the first part of the study the groups were asked to complete 15 multiple choice questions which were about procedural algorithms. The results from the first portion showed that pairs spent 20.9% more time then individuals doing the questions but the pairs took 4.2% less time to get all the questions correct (Liu and Chan 2003). As part of their conclusions the authors note pair programming is good for design driven activities. The second portion of the study had people spending part of their time as a pair and part of their time as a solo programmer. The questions used in this part of the study were deductive

problems. The developers were paired for the first problem and then separated as individuals to do a second problem. The developers then reformed as pairs to correct the first problem. The study found that pairs spent significant more time to do problems versus individuals (65 min pairing vs. 72 min solo) (Liu and Chan 2003). In terms of quality, pairs were achieving an 85% correctness rate versus a 51% correctness rate for individuals (Liu and Chan 2003). The study concluded that the quality for pairs was much higher than for individuals for deduction problems.

2.2.2 Perceptions Using Pair Programming

In many pair programming studies such as (Cockburn and Williams 2001; Williams et al 2001; Williams et al 2000a; Nagappan et al 2003; McDowell et al 2003; Hanks et al 2004; Williams et al 2000b) the opinions and thoughts of pair programming participants have been captured. One of the most prevalent perceptions that developers and students have had when using pair programming is that pairing gives them increased confidence in the work they are doing (Cockburn and Williams 2001; Williams et al 2000a; Williams et al 2000b; McDowell et al 2002; Hanks et al 2004;Williams and Kessler 2000; DeClue 2003; VanDeGrift 2004). In some studies the percentage of people feeling this way was as high as 89% (McDowell et al 2002) to 95% (Williams et al 2000a; Williams et al 2000b). In my opinion one reason for the increased confidence is there are now two people looking at the same piece of code. It would be expected that there is a greater chance that two people, with different view points, should catch most of the problems in the code. In addition, having a partner brings more confidence to the solution because there is now someone to back you up and share the same opinion about the solution. There are now two people taking the "risk" for the solution. Other common perceptions of pair programming are: developers and students enjoy using pair programming over solo programming (Nagappan et al 2003;Williams et at 2000b;Melnik and Maurer 2005; Sanders 2003). In three studies the number of developers who felt this way was 59% (Nagappan et al 2003) to 90% (Williams et al 2000b). Developers also perceive pair programming produced better quality code (Cockburn and Williams 2001; Williams et al 2000a; DeClue 2003; Melnik and Maurer 2005), increased productivity (Williams et al 2000a; VanDeGrift 2004; Melnik and Maurer 2005) and provided an

environment where learning and knowledge transfer takes place (Cockburn and Williams 2001; Williams et al 2001; Williams et al 2000a; Melnik and Maurer 2005; Sanders 2003). Other studies noted pair programmers are more satisfied working in pairs (McDowell et al 2003; Hanks et al 2004) and that they enjoy their programming work or job more (McDowell et al 2003;Williams et at 2000b; Succi et al 2002). Overall developers and students who use pair programming found that it helped them in many ways; such as to increase their confidence in solutions, increase in their job satisfaction, and provided them with an environment where learning can take place.

2.3 Test-driven Development

The idea of test-driven development (TDD) is comprised of four key points (Astels 2003):

1. Maintain an exhaustive suite of Programmer Tests (unit tests).
2. No code goes into production unless it has passed associated tests.
3. Write the tests first before production code.
4. The tests determine what code you need to write.

By following a test-driven development approach, the developers receive at least three benefits. First, to be able to write a test, the developers require a good understanding of what they will be implementing. They will have to know what types and values the inputs to the function will be and they will have to know enough about the function to determine what the expected out puts are. Second, by writing the tests first the developers can allow the tests to determine how much code they will have to write. After writing the tests the developers should only write enough code to make the test pass (Astels 2003). This may mean a more compact implementation to the problem being solved. Third, by writing tests first the developers will have unit tests which can be used for regression testing to make sure that future changes do not cause errors in previously implemented code (Robillard, Kruchten, and d'Astous 2003).

The type of testing used with test-driven development is unit testing. Unit testing is where a developer creates tests that focus on verifying the smallest testable elements

(or units) of the software (Robillard, Kruchten, and d'Astous 2003). These tests are made up of a set of input parameters and an expected output answer. The unit test compares the expected answer to what is generated when the code being tested is executed. If the expected and actual answers do not match then the test will fail and alert the developer. Unit tests are usually run when the software is compiled so the developers do not have to manually run the tests to check if the program is still working (regression testing).

Some tools that assist developers in executing unit tests are: Junit – Junit is a unit-testing framework for Java environments (Junit 2005) and Nunit – Nunit is a unit-testing framework for Microsoft .NET platforms (Nunit 2005). These unit-testing frameworks provide developers with the ability to quickly create and run unit tests. One benefit is that the unit-testing framework can be integrated into the compile process which allows unit tests to become automated unit tests and run whenever the software is compiled. Another benefit is that unit test frameworks provide a graphical or text based report based on the results of running the unit tests. This report describes how many tests passed, how many tests failed, and where the failures were which can allow a developer to quickly identify problems.

2.4 Test-driven Development Studies

In 2001 a study was conducted comparing test-first development (also known as test-driven development) and non test-first development approaches (Muller and Hagner 2001). The study compared two groups of graduate students. One group used a test-driven development approach while the other group used a code and then test approach. The study was split into two phases. In the first phase the students developed source code until they thought the program was complete. In the second phase the students were given a set of unit tests to pass and were allowed to correct their program. The study found no difference in how long it took either group to complete the program. The test-first group were less reliable than the other group after the implementation phase, but were more reliable after the second phase. The authors conclude that writing programs in a test-first manner did not lead to faster development or an increase in quality. However using test-

first development may provide better program understanding and faster code reuse due to tests providing feedback on the correct usage of reused code.

Maximilien and Williams (2003) conducted a study at IBM Retail Store Solutions to assess test-driven development. The study compared a period before test-driven development where the company was using an ad hoc testing approach with a period after test-driven development was used. The study showed a 40% reduction in defect density after test-driven development was implemented without a discernable impact to productivity.

An experiment using professional developers was performed in 2003 by George and Williams (2003). For the experiment, two groups were created. One group would use test-driven development and pair programming while the other group would use only pair programming. The group using pair programming however would develop the software using a design-develop-test approach (George and Williams 2003). The results of the study indicate the group using test-driven development produced higher quality code (passed 18% more test cases) but took 16% longer to complete this work. Qualitative results indicate the developers felt test-driven development is effective in terms of code quality and improving developer productivity however some developers were concerned about the time required to write tests (George and Williams 2003).

2.5 Continuous Integration

Continuous integration is where developers "Integrate and build the system many times a day, every time a task is finished." (Fowler and Foemmel 2005). Once a system is built, regression testing is then used. "Continual regression testing means that no regression in functionality is the result of changed requirements." (Fowler and Foemmel 2005).

Continuous integration is an extension of the idea of building the software every day. Daily build is "*the team/individual controlled delivery of code to a main codeline which is built and tested daily*" (Karlsson, Andersson, and Leion 2000). The daily build is

made up of two parts. The first part is the ability to build the system on a daily basis. Karlsson, Andersson and Leion (2000) say that having the ability to build the system on a daily basis means:

1. The designers must know where to put the code.
2. The code must be consistent.
3. The system should be able to be compiled in less than 24 hours.

The second part of daily build is the testing aspects and in the case of daily build the testing is automatic testing. Karlsson, Andersson and Leion (2000) mention, it is of little use to be able to build a system on a daily basis if we cannot ensure that it has a certain level of quality. The purpose of the tests is also to make sure that existing functionality is not broken. This is the idea of regression testing which is used to verify requirements using existing test cases after software changes have been made (Robillard, Kruchten, and d'Astous 2003). There are many benefits to building and testing the system on a daily basis. Some of the benefits are (Karlsson, Andersson, and Leion 2000):

- Stability (always have a stable build, if build fails revert to old one),
- Test (teams can integrate their code and test it on a system level),
- Feedback (teams gets early and continuous feedback),
- Early fault detection (Faults are detected early),
- Isolate errors (errors come from new build but not previous), and
- Parallel system test (system testing can be run parallel to development).

Continuous integration means that when a task for the system is completed, the system will be built, integrated, and tested. The main idea of continuous integration is the build should always be in a working state. If the build breaks, work should not continue until the build is fixed. This is done so the code in the source repository can be compiled and deployed at any time. The use of continuous integration may also cause developers to be more careful, as any errors they cause will be visible to the entire team and in some cases the customers. One limitation of continuous integration is that it requires a fast build process, usually a couple of minutes in length. If the build process takes many hours then a daily or weekly build would be more appropriate.

One popular tool for continuous integration is a product called CruiseControl (CruiseControl 2005). CruiseControl comes in both Java and Microsoft .NET versions. The CruiseControl software monitors a source code repository for changes. If there are changes, cruise control with the assistance of an automated build file, will retrieve, build, and run any tests associated with the compiled software. Once the build has finished, CruiseControl will communicate the build status to the development team through:

1. Email
2. Instant Messaging
3. CcTray (program running on developers' computers to display integration status)
4. Devices such as sirens, lights, lava lamps

In combination with the above methods to inform the developers of the current build state the program also creates a website. The website shows the development team the current build, whose files were checked in as part of the build, any warnings or errors, and unit tests status (if there are unit tests). Along with showing the current build information the website also has the information from previous builds, so developers or customers can view the build history and see any past failures and their reasons.

2.6 Case studies

Case studies can be thought of as "research in the typical" (Fenton and Pfleeger 1997). Case studies have been defined by in the following way (Yin 2003):

A case study is an empirical inquiry that investigates a contemporary phenomenon within its real life context, especially when the boundaries between the phenomenon and context are not clearly evident

Yin (2003) states that one would use the case study methods because "*you deliberately wanted to cover contextual conditions-believing that they might be highly pertinent to your phenomenon of study*". I believe contextual information is very important is a workplace setting. This is due to the fact that there is less ability to control

variables in this type of setting versus a more controlled setting, such as university course or lab. This lack of control means since the context of the variables cannot be controlled it must be recorded and taken into account when analysing the results.

The type of case study developed is an *exploratory case study* (Yin 2003). In the case of this study, the research goal is addressed by exploring the introduction and use of Scrum, pair programming, and test-driven development with continuous integration over the long term. In addition, the study is looking at information from the same organization over the period of approximately two and a half years, so it can be considered a longitudinal case study as defined by (Yin 2003).

2.7 Ethnology

Ethnographic research methodology comes from the area of anthropologic studies. Its main goal is to describe the cultural context of a situation. Ethnology has been used in other information studies (Guy 2000; Sharp, Woodman, and Robinson 2000). One of the main benefits of using ethnographic methods is their depth and the context dependent insights that they provide. As Klein and Myers (1999), point out, "*Because the researcher is "there" for an extended period of time, the ethnographer sees what people are doing as well as what they say they are doing. Over time the researcher is able to gain an in-depth understanding of the people, the organization, and the broader context within which they work".*

I believe the depth of understanding and breadth of context can help explain how methods in software development interact with the developers and how they are used. In my opinion the main limitations of ethnographic research are:

1. Ethnographic research generates a large amount of information in terms of field notes and observer opinions.

2. Ethnographic studies are very time intensive as they are more suited to long term observation.

3. Researcher Interaction with participants may cause changes in participant behaviour and possible researcher bias threatening the internal validity of the study

I chose to use the ethnographic approach of field observation because of the ability to get a deeper depth and greater breadth of understanding about the people and their environment. Since software development has a large human role, I feel that it is important to provide a broader context as to the environment in which both the quantitative and qualitative results were obtained.

2.8 Summary

The material presented in Chapter 2 should give the reader a base level of understanding of the practices which will be discussed in the rest of the thesis. This chapter should also give the reader an impression as to the current state of research in the areas such as Scrum, pair programming, test-driven development, and continuous integration, along with the state of knowledge in agile methods over the long term. The background information is provided to allow the comparison of what I have done to what others have done in the area of agile methods in industry over the long term.

CHAPTER THREE: CASE STUDY CONTEXT

In this section, I will discuss some contextual information that may assist the reader in interpreting the results as they are presented.

3.1 Company Information

PetroSleuth Inc is a small software development company. They create custom windows and web applications for the oil and gas industry. The company develops software using Microsoft .Net technologies such as C# and web services. PetroSleuth is co-located with an oil and gas consulting firm that provides domain expertise for the software PetroSleuth produces. PetroSleuth has two goals, first to create and maintain software for the consulting firm so that they can increase their productivity and secondly, to provide software development services to outside industry clients.

3.2 Team Size:

Figure 3-1 outlines the team size and how it changes over time.

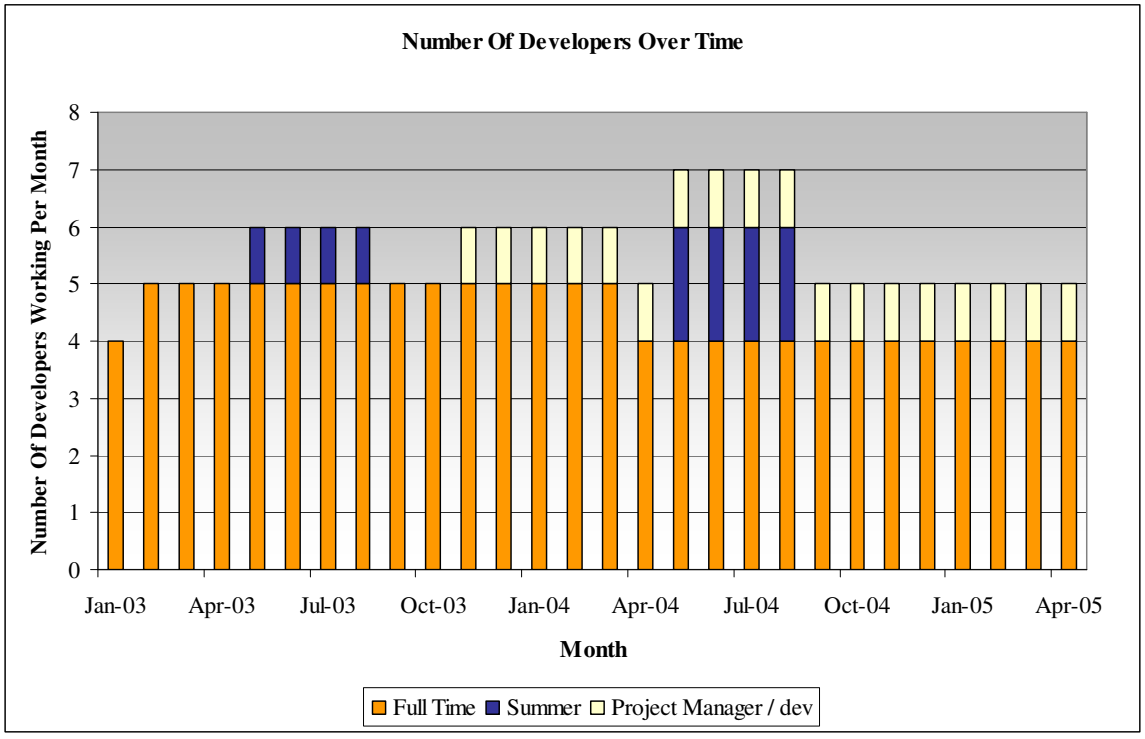


Figure 3-1: Team Size per Month

Starting in January 2003, there were four full-time developers, with the president of the company acting as the project manager. In February of 2003, a developer was hired out of university. I again worked as a summer student In May of 2003 until September 2003 when I started this research and ceased being a software developer. In October 2003, one of the senior developers was given the role of project manager. He, however, also remained as a part time developer by splitting his time 50% development and 50% project management. In November 2003, a developer from PetroSleuth's client was added to the team. In March 2004, one developer left the company. In April 2004, one developer left the company and one developer was hired. In May 2004, two computer science summer students started to work for PetroSleuth. These summer students worked until the end of August 2004. In November 2004, a developer left the team and was replaced the same month (November 2004). The replacement developer was officially expected to start at the end of November 2004 but did not end up working full time at PetroSleuth until the middle of January 2005 due to prior work from his previous job.

3.3 Highest Degree Obtained

Table 3-1 lists the degree and the number of developers holding that degree.

Table 3-1: Highest Degree Obtained

Degree	Number of Developers
B.Sc Computer Science	4
B.Sc Plant Sciences	1
B.Eng	1
MCSE	1
None	3 *

*Two of the developers with no degree were summer students entering their 4th year in Computer Science at the University of Calgary

3.4 Experience Level

The table 3-2 outlines the amount of software industry experience the development team had.

Table 3-2 Software Industry Experience

Years of Experience	Number of Developers
<5 years of software industry experience	6
15 years of software industry experience	1
>20 years of software industry experience	2

The table shows most of the software developers had less than five years of software industry experience. Most of the team is comprised of summer students and developers just out of university.

3.5 Domain Experience

Through discussions with senior oil and gas industrial personnel a high level of domain experience was 15+ years of experience in the oil and gas industry, medium as 10 to 14 years of experience and low as less then 10 years. All of the developers had less than 10 years and are classified as having a low level of domain expertise

3.6 Programming Language Expertise

PetroSleuth uses two languages for software development, C# and SQL. All of the developers had less then five years of expertise using C#. One developer had four years of SQL experience. The following table 3-3 will show the language experience of the development team as reported in questionnaire Q1 found in Appendix A.

Table 3-3: Language Experience

Language	Number of Developers	Min Experience (Years)	Median Experience (Years)	Max Experience (Years)
C or C++	4	0.5	1.3	5
Java	4	1	1	1.5
VB	4	1	1	6
Assembler	2	1	3	5
JavaScript	2	1	4	7
HTML	3	1	1	10
COBOL	2	5	5.5	6
SAS	1	4	4	4
PowerBuilder	1	2	2	2
NATURAL	1	4	4	4
PERL	1	1	1	1
Python	1	1	1	1
C#	6	0.5	1	1.5
SQL	1	4	4	4
Pascal	2	0.5	0.5	0.5

3.7 Agile Experience

Based on the results of questionnaire Q1 given at the beginning of the study found in Appendix A all developers had heard of Extreme Programming (Beck and Andres 2004) and all but one had heard of Scrum (Schwaber 2004). Two were familiar with the term (DSDM Stapleton 1997), three were familiar with the term Feature Driven Development (FDD) (Palmer and Felsing 2002) and three were familiar with the term agile modeling (Ambler 2002).

Table 3-4 Agile Experience

Agile Experience	Number of Developers
Tried unit testing at home	1
Pair Programmed assignments in school	2
Took class teaching Extreme Programming	2
No agile experience	3

Three of the developers reported they did not have any experience with agile methods before the study. Two developers (summer students) reported they had taken a course in university that had taught extreme programming (XP). Two developers

indicated they had used pair programming for assignments at university. One developer reported that they had tried unit testing at home.

3.8 Project Manager Experience

The project manager from October 2003 until the present (April 2005) had 10 years of project management experience and a PMP certification. The project manager, prior to October 2003, had 15 years of project management experience.

3.9 Ergonomic factors

The developers are arranged in private offices. Each office has a desk, two chairs, and a white board. Most of the developers have dual monitors for their workstations with wireless mice and keyboards. The customers are located on the floor below the developers.

3.10 Software Development Methodology Prior to Scrum

Before the introduction of Scrum, the software development process could be considered having an ad hoc approach. There was little actual planning involved when developing the software. There were no planning meetings per say. Every so often, when the president of the company wanted to get a status update, as to the progress everyone was making, he would call a meeting. The meetings were usually held every Monday morning. Sometimes during these meetings, as part of the update, the team would talk about what they were working on but the meeting was not seen as a place to do planning. In addition to the usual Monday meetings, the president would sometimes call impromptu status meetings. A major problem with these impromptu status meetings was they were called without much warning and though they were intended to be only a few minutes, they took much longer, sometimes 45 minutes or more. In addition to both the Monday meetings and impromptu status meetings, there were also weekly meetings on Wednesday with the stakeholders of the product.

During the weekly meetings, functionality that was completed in the past week was demonstrated. Sometimes if user functionality could not be shown, it was described. If there was nothing to describe or demonstrate, the session would turn into a question and answer session about the product. During the meetings, stakeholders would make suggestions and requests improvements and additional functionality. The suggestions and requests for the software being developed sometimes changed drastically from week to week.

There were two problems associated with the rapidly changing requirements. First, there was a very poor picture of what work was going on internally and what work was planned for future completion. Internally, there was no place that a developer could go to see an up-to-date comprehensive picture of what needed to be completed and when it was expected in relation to other pieces of functionality. One method, used to try to reduce this confusion, was to have a meeting every so often to discuss what was needed to be finished and when it was supposed to be completed. These meetings helped in the short term, but did not address the core issue of controlling the ever increasing number of requirements for the release. The second problem associated with the rapidly changing requirements was there was no control put in as to what the developers were working on. If a stakeholder suggested something in the weekly meeting, it was very likely the developer would add what was requested to what they were working on without consulting anyone.

Even though the team was communicating with the customers on a weekly basis and showing what they had done, the process was still out of control in terms of controlling what features went into the product at what point.

3.11 Project Contexts

In this section I will describe the different projects the team had worked on over the course of the study, to give the reader an indication as to the size of the projects that the team under took.

3.11.1 Windows Application 1 and 2

Windows Application 1 was created in late 2002 to provide the client company with the ability to create and manage oil and gas projects and to improve the productivity of its employees. Based on discussions with the developers the first windows application was released in January 2003. After the release, the developers started to extend the application by adding features and fixing bugs. These changes caused windows application 1 to evolve into windows application 2. The development of the second windows application continued until the end of October 2003 when the PetroCube website development started. From October 2003, forward only bugs were fixed for windows application 2. October 25th 2003 was chosen as the first date to provide project metrics for the two windows applications as that was when windows application 2 stopped new feature development. April 30th, 2005 was chosen as it was the last day of data collection for the study. The metrics provided in table 3-5 contains the code for both windows application 1 and 2 since there is no copy of the first windows application before development on windows application 2 was completed.

Table 3-5: Windows Application 1 and 2 Size Metrics

Product Type	**Windows Application 1 and 2**	
	October 25th 2003	**April 30th 2005**
C# Lines Of Code	91571	108222
C# Statements	41493	48686
C# Number of Classes	305	343
C# Number of Methods	6141	6962
C# Average Cyclomatic complexity	3	3
SQL Metrics	See Table 3-7	
Languages	C# and SQL	

3.11.2 PetroCube Website

The website was created to provide internal users and external clients with the ability to interact with charts and statistics generated, using models and procedures developed by the internal customer company. The development of the website started in late October 2003 and it was released internally in March 2004. The internal clients used

and tested the website for two weeks and then released it to external clients at the beginning of April 2004.

Table 3-6 details the metrics of the website product during two points in time. The first is on March 6 2004. This day was chosen to generate metrics on the website because it was close to the time when the website was released to the internal clients for their own testing purposes. The second metrics calculation was performed on the April 20 2005 version of the website source code, to show what the website code base looked like at the end of data collection. These metrics are being provided to give an idea of the size of the three main products developed over the course of the data collection period.

Table 3-6: Website Size Metrics

Product Type	**Web Site Application**	
	Mar 06 2004	**April 30 2005**
C# Lines Of Code	14022	17148
C# Statements	6935	8564
C# Number of Classes	60	49
C# Number of Methods	583	905
C# Average Cyclomatic complexity	4	4
Lines of HTML	357	855
SQL Metrics	See SQL Note below and Table 3-7	
Languages	C# and SQL and HTML	

SQL Note:

The SQL metrics provided in Table 3-7 are based on the combined SQL source code for both the website and the first two windows applications. The developers believe most of the SQL code is for the first two windows applications.

Table 3-7: Website, Windows Application 1 & 2 SQL metrics

	Mar 6th 2004	**April 30th 2005**
Physical Lines (no white space)	43284	45924
Commented Lines	10061	10697
Number of Procedures and Functions	487	518
Number of Select, Insert, Update, and Delete	2995	3142

3.11.3 Windows Application 3

The third windows application is a program to display statistical information generated using models and procedures developed by the client company. This application is complementary to the website as it allows users to provide customized lists on a much smaller subset of wells than the website. The client company believes that the flexibility of the third windows application will provide users with increased productivity as currently users have to export the website data to Microsoft Excel and then manipulated it. This is a time consuming process. The software, which became the third windows application started development in April 2004. Table 3-8 and 3-9 describe C# and SQL project metrics for the third windows application. March 12th 2005 was selected as a date for as a data collection point as it was a day after the internal release of the software. April 30th 2005 was chosen selected as the date for as a data collection point as it was the last day of the study and the software was about to be released to external users.

Table 3-8: Windows Application 3 Size Metrics

	Mar 12th 2005	April 30th 2005
C# Lines Of Code	77361	86139
C# Statements	37552	40460
C# Number of Classes	271	285
C# Number of Methods	4903	5271
C# Average Cyclomatic complexity	3	3
Languages	C# and SQL	

Table 3-9: Windows Application 3 SQL Size Metrics

	March 12th 2005	April 30th 2005
Physical Lines (no white space)	62482	66297
Commented Lines	10061	10697
Number of Procedures and Functions	819	827
Number of Select, Insert, Update, and Delete	20321	20585

CHAPTER FOUR: CASE STUDY DESIGN

4.1 Case Study Overview

The goal of the study was to evaluate the introduction and use of Scrum, pair programming, and test-driven development with continuous integration in a small company over the long term in the context of a small team industry environment.

To do so a longitudinal case study was conducted at a small company doing software development, PetroSleuth Inc. One of the reasons I chose PetroSleuth is that I had worked at the company for a number of summers and the company was interested in hiring me after I finished by BSc. I, however, wanted to peruse a master's degree and after some discussions between the President of the company and my supervisor, this research project was agreed upon.

Prior to the commencement of this study, I obtained ethics approval for the University of Calgary for experimentation involving human participants. The letters of approval from the ethics committee can be found in Appendix I.

The participants in the study were chosen as they were employees at PetroSleuth Inc. Prior to commencing the study the participants each completed and returned an informed consent form, which can be found in Appendix G.

The initial plan for the study was to introduce Pair Programming first and then observe its use for approximately 5 months. After that time, test-driven development with continuous integration was to be introduced. At the point when test-driven development with continuous integration was to be introduced, the developers were going to be asked if they wished to continue to use pair programming for the rest of the study. The practices were split up to attempt and gather some initial information on pair programming then after approximately 5 months when the team had stabilized using pair programming introduce test-driven development and evaluate it also. My role at the company was to

introduce the practices, to manage and provide support for any tools required for the practices and to assist in their build management and content management as the tools for the study would have to integrate with that system. I also acted as an observer, sitting in on the team's meetings, taking notes and making observations.

For the initial portion of the study, the practice of pair programming was introduced as soon as ethics approval was received at the end of January 2004. Prior to introducing pair programming the developers were given a questionnaire to gather some background information about them and their opinions on agile methods and working with a partner while programming.

At the end of January 2004, pair programming was introduced by giving the developers a presentation on pair programming. Initially pair programming was to be used until June 2004 when test-driven development and continuous integration were to be introduced. At that time, the developers were to be asked if they wanted to continue to use pair programming or not. This question was asked to provide management with an assurance that if pair programming was detrimental to their software development then it would not be continued. However, when June 2004 came, the developers wished to continue the use of pair programming and pair programming continued in use after the study was completed in April 2005.

Prior to commencing the study, the developers had been recording their office hours in an in house program called Time Tracker, which is described in Appendix C. The developers continued to use Time Tracker to record their office hours throughout the study. This means that office hours were available from January 2003 until the end of the study in April 2004. In addition to tracking office hours as usual, the developers were also asked to keep track of how much time they spent pair programming, pair reviewing, and pair designing on sheets provided to them by myself. More information about the pair programming sheets can be found in Appendix C.

In May 2004, the Scrum software methodology was introduced. Unlike pair programming, I did not do the introduction. The introduction was done by one of the developers / project manager. Even though Scrum was not planned as part of the initial study its use and possible impact on other results needed to be addressed so Scrum was added as another agile technique to explore. Scrum was used from May 2004 and continues to be used after the study was completed in April 2005.

As part of the Scrum process the developers now had to keep track of their project hours in a backlog tracking system VersionOne (VersionOne 2005). From VersionOne I was able to retrieve these backlog items on a per sprint basis continually throughout the period form May 2004 until April 2005.

At the beginning of June 2004, the first post-pair programming questionnaire was given to the participants. This questionnaire was to get feedback on what the developers thought of pair programming. At the same time, the developers were given a pre-test-driven development questionnaire to get their thoughts on developing test code before production code.

From the responses from the post-pair programming questionnaire an interview was formulated to fill in some gaps left from the questionnaire responses. This interview was conducted in July 2004.

In June 2004 my research notes and observations started to be written down on a daily basis. Prior to this point I was not very familiar with writing field notes. This left the period prior to June 2004 without observations being recorded. From June 2004 to the completion of the study however, research notes and observations were recorded on a daily basis and can be found in Appendix A.

The next questionnaire was run at the end of January 2005. This questionnaire was a post-test-driven development questionnaire. The purpose was to gather developer opinions about test-driven development after using it for 8 months.

In March 2005 two questionnaires were conducted. First a Scrum questionnaire was given to the customer group to get their opinions on the use of Scrum in the company. A similar questionnaire was also given to developers at the same time also asking about Scrum.

In April 2005 a final wrap up interview was conducted to get a final set of opinions from the developers as to their thoughts on the different techniques used.

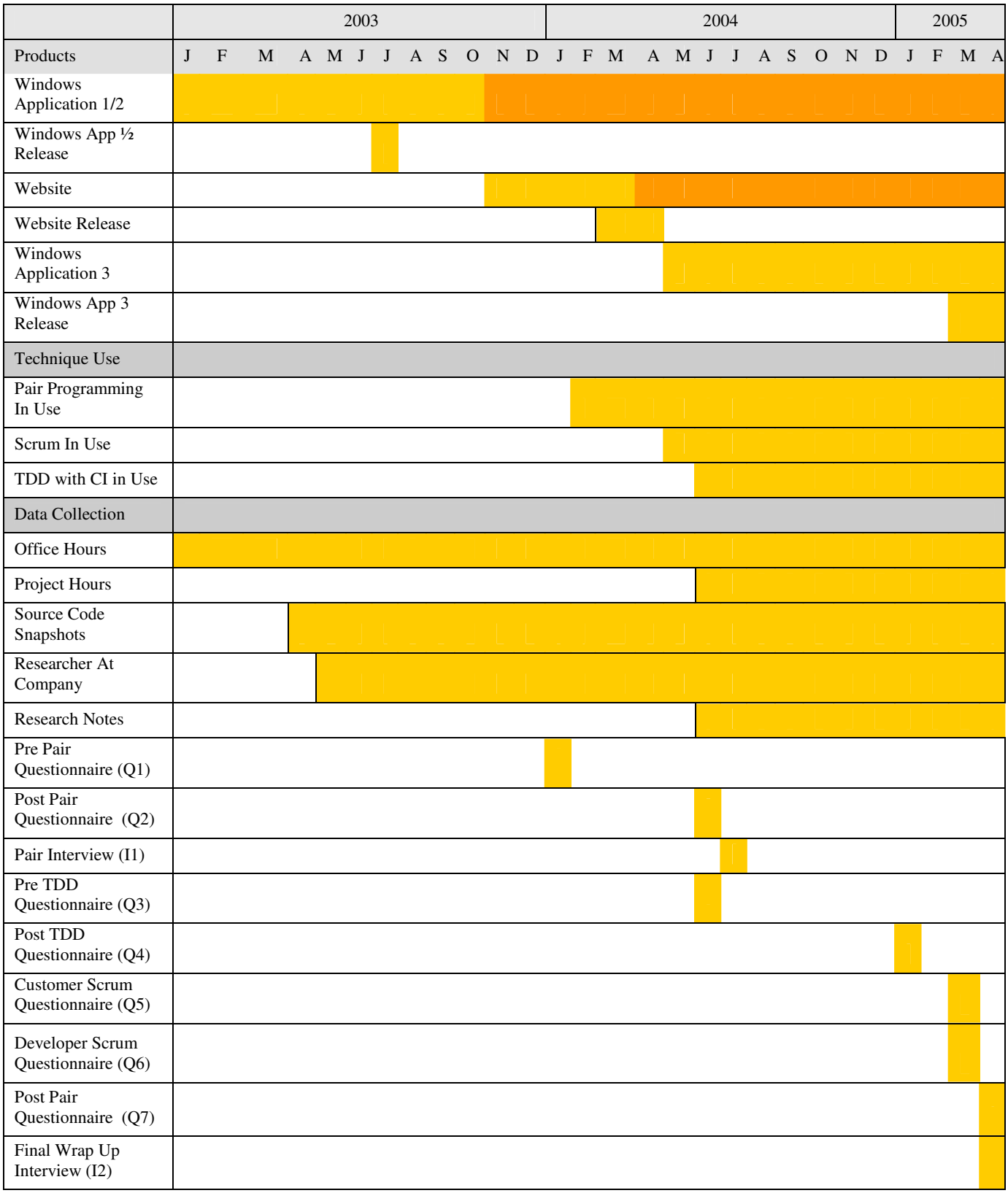
One final data source that was available and retrieved throughout most of the study were source code snapshots of the development the developers were working on. For more information on source code snapshots see Appendix C.

Table 4-1 shows a time line of the study in a graphical format. The table is split into three sections. The first section outlines the timelines for the projects (Section 3.2). The yellow (left) portion of the bar, beside the products, means the project was in active development. In the rows titled Release the coloured bar denotes the month the product was released. The orange (right) portion of the bar shows when the product was in a maintenance phase and not being actively developed.

The second section outlines when the techniques covered by the study were in use at the company. The months where the technique was being used is colored with a yellow bar.

The third section is when the data was available and or being collected. For the office hours, project hours, and source code snapshots a yellow bar means the data was available or being collected. For the interviews and questionnaires a yellow bar indicates the month the data was collected.

Table 4-1 Study Timeline



4.2 Goals Questions Metrics

4.2.1 Goal:

Evaluate the introduction and use of Scrum, pair programming, and test-driven development with continuous integration in a small company over the long term in the context of a small team industry environment.

4.2.2 Questions:

The metrics listed for each question will appear in the metrics section 4.2.3.

Q1: Does using Scrum provide a sustainable working pace for developers?

Purpose: Anecdotal evidence suggests Scrum provides a more stable and sustainable pace for developers. Does this effect occur?

Metrics:

M1: Researcher Notes and Observations

M8: Developer Scrum Questionnaire (Q6)

M10: Wrap up Developer Interview (I2)

M11: Mean Percent Overtime by week

Q2: How will Scrum be viewed by the customer group?

Purpose: The purpose of the question is to get opinions about Scrum from the customers' perspective. Since Scrum involves significant customer communication and control over the software being produced, this may lead to more satisfied customers.

Metrics:

M1: Researcher Notes and Observations

M7: Customer Scrum Questionnaire (Q5)

Q3: How accurate are the backlog estimates and does this change overtime?

Purpose: To evaluate the developers' estimates and their ranges over time.

Metrics:

M1: Researcher Notes and Observations

M10: Wrap up Developer Interview (I2)

M12: Misallocated Hours Over and Under Estimation

M13: Misallocated Hours Absolute Value

M14: Percent Estimation Error

Q4: How well do the customers and the developers adhere to the Scrum practices?

Purpose: When evaluating Scrum, it is important to understand how well the developers and customers adhered to using the practices. This understanding is needed to evaluate any results provided.

Metrics:

M1: Researcher Notes and Observations

M10: Wrap up Developer Interview (I2)

Q5: What issues are encountered when using Scrum and what are some solutions?

Purpose: To provide insight into what difficulties are encountered while using Scrum and what possible solutions were used to solve them. This information may be useful to future practitioners and researchers when implementing Scrum.

Metrics:

M1: Researcher Notes and Observations

M7: Customer Scrum Questionnaire (Q5)

M8: Developer Scrum Questionnaire (Q6)

M10: Wrap up Developer Interview (I2)

Q6: What are the opinions of the developers using Scrum?

Purpose: To receive qualitative feedback based on what the developers thought of using Scrum.

Metrics:

M1: Researcher Notes and Observations

M8: Developer Scrum Questionnaire (Q6)

M10: Wrap up Developer Interview (I2)

Q7: How often will developers use pair programming over the long term?

Purpose: To evaluate how well and how often the developers use pair programming. This information is used when interpreting the results of the study.

Metrics:

M1: Researcher Notes and Observations

M3:Post-Pair Programming Questionnaire after 4 Months (Q2)

M4: Interview after Four Months (I1)

M9: Post Pair Programming Questionnaire Second Collection (Q7)

M10: Wrap up Developer Interview (I2)

M16: Pairing Percentage per Month

M15: Pairing Percentage per Sprint

Q8: What possible factors could influence the amount of pair programming?

Purpose: To gather possible influences that may have affected the use of pair programming though out the study. These influences are used to interpret the results and provide avenues for future research.

Metrics:

M1: Researcher Notes and Observations

M3:Post-Pair Programming Questionnaire after 4 Months (Q2)

M4: Interview after Four Months (I1)

M9: Post Pair Programming Questionnaire Second Collection (Q7)

M10: Wrap up Developer Interview (I2)

M17: % of pair tasks in Sprint

M18: % of sprint estimated for pair tasks

Q9: What do developers think about using pair programming?

Purpose: To gather qualitative information about developer thoughts throughout the study as they are using pair programming. Developer opinions are important as in the end it is the developers who are using the practices. Their thoughts may also provide insight into the effect of pair programming on the software produced.

Metrics:

M1: Researcher Notes and Observations

M3:Post-Pair Programming Questionnaire after 4 Months (Q2)

M4: Interview after Four Months (I1)

M9: Post Pair Programming Questionnaire Second Collection (Q7)

M10: Wrap up Developer Interview (I2)

Q10: What do developers use pair programming for?

Purpose: To gather information about the kinds of tasks for which developers use and do not use pair programming. This information is important to interpret the results and may provide future work regarding tasks that may or may not be suitable for pair work.

Metrics:

M1: Researcher Notes and Observations

M3:Post-Pair Programming Questionnaire after 4 Months (Q2)

M4: Interview after Four Months (I1)

M9: Post Pair Programming Questionnaire Second Collection (Q7)

M10: Wrap up Developer Interview (I2)

Q11: What do the developers think could be done to make the pair programming experience more enjoyable?

Purpose: To gather developer opinions of how pair programming can be made more enjoyable. The developers may offer insight in to what could be done to improve the use of pair programming. This information is of great use to practitioners and researchers as it may allow them to have a better pair programming experience.

Metrics:

M1: Researcher Notes and Observations

M3:Post-Pair Programming Questionnaire after 4 Months (Q2)

M4: Interview after Four Months (I1)

M9: Post Pair Programming Questionnaire Second Collection (Q7)

M10: Wrap up Developer Interview (I2)

Q12: How often will developers use test-driven development and continuous integration?

Purpose: To determine how well and how often developers use test-driven development and continuous integration. This question is useful in interpreting the results.

Metrics:

M1: Researcher Notes and Observations

M6:Post-Test-driven Development Questionnaire (Q4)

M10: Wrap up Developer Interview (I2)

M19: Test code Percentage of System

M20: Number of Assertions

M21: Test code coverage

Q13: What are some possible factors that influence test-driven development and continuous integration's use?

Purpose: To gather possible influences that may have affected the use of test-driven development and continuous integration thoughout the study. These influences are used to interpret the results and provide avenues for future research.

Metrics:

M1: Researcher Notes and Observations

M6: Post-Test-driven Development Questionnaire (Q4)

M10: Wrap up Developer Interview (I2)

Q14: What do developers think of using test-driven development and continuous integration?

Purpose: To gather the opinions of developers using test-driven development and continuous integration. Developer opinions are important as the developers are the ones using the practices.

Metrics:

M1: Researcher Notes and Observations

M6: Post-Test-driven Development Questionnaire (Q4)

M10: Wrap up Developer Interview (I2)

Q15: Does the use of TDD improve the defect density of the software produced?

Purpose: Test-driven development is supposed to reduce the amount of defects in the software produced. This question asks if the amount of defects in the product improved after using test-driven development.

Metrics:

M19: Test code Percentage of System

M20: Number of Assertions

M21: Test code coverage

M22: Defects Found per Product

M23: Defect Density by Product

4.2.3 Metrics:

Questionnaire and interview responses can be found in Appendix A. Detailed metrics calculations can be found in Appendix B. Research Notes can be found in Appendix A.

M1: Researcher Notes and Observations

Some of the results obtained were through observations and conversations with developers and in some cases, opinions based on the context of the observations. The purpose of notes and observations is to provide a richer background and context to the qualitative and quantitative results of this thesis. My observations were recorded in a research note document. The research notes were started in June 2004 and prior to this date no written personal observations exist.

M2: Pre-Study Questionnaire (Q1)

Questionnaire Q1 was given before pair programming was introduced. The questionnaire focused on gathering background information about the developers, their experiences with agile methods, their thoughts on pairing, and some general opinions

about different aspects of software development. Out of 6 questionnaires sent, 4 were returned. Of the two developers that did not fill out the questionnaire, one was a senior developer who eventually left the company in March 2004 and the other was a junior developer who left the company in April 2004. The first questionnaire was also given to the two summer students when they started in May 2004. Out of the two sent out, two were returned. It should be noted that the two summer students are related to me. Additionally, the questionnaire was given to the developer who joined the company in November 2004. He returned his questionnaire.

M3: Post-Pair Programming Questionnaire after 4 Months (Q2)

The second pair programming questionnaire was given at the beginning of June 2004. This date was chosen as it was the point just before test-driven development and continuous integration were introduced. The second questionnaire focused on how pair programming was used and if there were any problems or suggestions for pair programming in the future. Out of 7 questionnaires distributed 7 were returned. The two summer students were also included.

M4: Interview after Four Months (I1)

The first interview I1 was run at the beginning of July 2004. Its purpose was to add additional information to the results obtained from the pair programming questionnaire Q2. The interview was a semi-structured interview. The seven developers were asked to provide an interview. All seven developers gave individual interviews. The interviews were recorded on digital tape and then transcribed for later analysis. As per the ethics application, the raw audio recordings were not kept after the interview had been transcribed.

M5: Pre-Test-driven Development Questionnaire (Q3)

The pre-test-driven development questionnaire was distributed along with the second pair programming questionnaire (Q2) in June 2004. The purpose of the questionnaire was to get some background information on the developers' familiarity with unit testing and test-driven development. In this questionnaire, no questions were

asked about continuous integration, because when the questionnaire was created I saw continuous integration as a tool to assist test-driven development rather than evaluating continuous integration as its own practice. Out of 7 questionnaires distributed, 7 were returned. The questionnaire was also filled out by the two summer students.

M6: Post-Test-driven Development Questionnaire (Q4)

The second questionnaire for test-driven development was distributed at the end of December 2004. The purpose of the second test-driven development questionnaire was to get some feedback about the test-driven development process after it was utilized for 6 months. Of 5 questionnaires distributed and all were returned. One of the developers was not given Q4 as he had just started working at PetroSleuth in November 2004. He was instead given the questionnaire (Q3) to complete.

M7: Customer Scrum Questionnaire (Q5)

To gather feedback on Scrum and the software developed before and after Scrum's introduction from the customers' perspective, a Scrum questionnaire was given to the three customers at the end of March 2005. Three questionnaires were sent and all were returned.

M8: Developer Scrum Questionnaire (Q6)

The Scrum questionnaire was given at the end of March 2005. The questionnaire focused on gathering some feedback about the Scrum process from developers. The questions focused on the software developed before and after the Scrum process. Out of 5 questionnaires sent all were returned.

M9: Post Pair Programming Questionnaire Second Collection (Q7)

The third pair programming questionnaire was given at the beginning April 2005. This questionnaire is the same as questionnaire (Q1) distributed in June of 2004. The questionnaire was being used again to see if there were any changes in the developers' responses after using pair programming for an additional eleven months. Out of 5 questionnaires sent, 4 were returned. The one questionnaire that was not returned was

from the Scrum master. Even though he still did some development, he spent most of his time doing project management related activities. He informed me that he had nothing else to add about pair programming as he had not used it and therefore did not fill out the questionnaire.

M10: Wrap up Developer Interview (I2)

The second interview was run at the beginning of April 2005. This interview was part of a final wrap up interview that included questions about pair programming and the other practices used during this study (Scrum, test-driven development and continuous integration). This interview was semi-structured in nature. Out of the five developers asked, all five provided individual interviews.

M11: Mean Percent Overtime by Week

Mean Percent Overtime is the percentage of hours worked over the expected amount of hours per developer per week.

M12: Misallocated Hours Over and Under Estimation

Misallocated Hours Over and Under Estimation is the number of hours over or under the estimated hours for a backlog item. The metric provides two results. First the metric shows how many hours were over allocated to a backlog item, meaning the backlog item was completed faster than the estimate (negative value). Second, the metric shows how many hours were under-allocated to a backlog item, meaning the backlog item took longer than the estimate to completed (positive value).

M13: Misallocated Hours Absolute Value

Misallocated Hours Absolute Value is the absolute value of the misallocated hours calculated in metric M12. The absolute value of the misallocated hours represents the magnitude of misallocation rather than how the hours were misallocated (under or over estimation).

M14: Percent Estimation Error

Percent Estimation Error is the error between EstimatedBacklogHours and BacklogHoursCompleted. The purpose is to show how close the estimated hours on backlog items are as a percentage of the actual completed on those backlog items. A positive percentage means the estimate was smaller than the work completed.

M15: Pairing Percentage per Sprint

The pairing percentage per sprint is the percentage of sprint hours spent programming with a partner. This metric indicates how much pair programming the developers did during the sprint.

M16: Pairing Percentage per Month

The pairing percentage per month is the percentage of office hours worked during a month spent programming with a partner. This metric indicates how much pair programming the developers did during the month.

M17: Percentage of Pair Tasks in Sprint

The percentage of pair tasks in a sprint is the percentage of tasks by task count that were assigned to a pair of developers. The purpose of this metric is to show how many tasks were assigned to pairs in relation to the total number of tasks assigned during the sprint.

M18: Percentage of Sprint Estimated for Pair Tasks

The percentage sprint estimated for pair tasks and the percentage of the total sprint hours estimated for pair tasks. The purpose of this metric is to show how much of the sprint was assigned to pairs in relation to the total number hour estimated for all tasks.

M19: Test code Percentage of System

The test code percentage of the system is the percentage of C# Lines, C# Statements and C# Members which are related to software testing compared to the application's C# numbers. Test code is code created for the purposes of testing the

system. Test code percentage is an indication of the relative amount of test code rather than describing the codes effectiveness for testing.

M20: Number of Assertions

An assertion is a comparison of an expected value with a resulting value. The number of assertions indicates how many tests of expected and actual values were performed. An assertion is counted if the word “assert.” or “Assert.” appeared on a line of code and is not part of a comment**.**

M21: Test code coverage

Test code coverage is the percentage of code executed when unit tests are run. Test code coverage shows how much of the system is executed and tested during automated unit testing. A large test code coverage value is more desirable over a small test code coverage value as a large value means more of the system is being tested by unit tests providing more chances to find bugs.

M22: Defects Found per Product

The defects per product are how many user-found defects were reported and recorded for a given product. There are three defect counts, one for each product (Windows Application 1/ 2, Website, and Windows Application 3). A user found defect is a problem reported by a user and classified as a defect in the opinion of the developers recording the problem.

M23: Defect Density by Product

Defect Density is usually considered as the number of defects in a product per line of code and is considered a de facto standard for software quality (Fenton and Pfleeger 1997).

CHAPTER FIVE: SCRUM

This chapter will discuss how a Scrum process was introduced at PetroSleuth and then provide some quantitative and qualitative results relating to the use of Scrum over twelve months. I will address the following questions:

- **Does using Scrum provide a sustainable working pace for developers?**
- **How will Scrum be viewed by the customer group?**
- **How accurate are the backlog estimates and does this change overtime?**
- **How well do the customers and the developers adhere to the Scrum practices?**
- **What issues are encountered when using Scrum and what are some solutions?**
- **What are the opinions of the developers using Scrum over the long term?**

Chapter 5 is structured in the following manner: In section 5.1, I present how Scrum was introduced into PetroSleuth. In section 5.2, I discuss the scrum practices as they are used at PetroSleuth. In section 5.3, a timeline of the sprints is presented. In section 5.4, the quantitative and qualitative results are presented. In section 5.5, I discuss the above questions.

5.1 The Introduction of Scrum

While working for a number of summers at the company I had discussions with the President of the company about instituting a more formal software process because, in my opinion, their current process was ad hoc. While discussing different software processes options with the president he mentioned did not want to institute a document

driven approach as he had previous experience with such an approach and found it too heavy on documentation. I therefore suggested maybe considering an agile methodology such as XP or Scrum. The president said they seemed like good ideas however no agile methodology took hold. One reason could be there was no champion of agile methods in the company full time.

At the end of March 2004, the project manager attended the VS Live Conference in San Francisco. At the conference, the project manager went to a presentation on the Scrum process. On his return, he said that we should transition to that process. He then became the Scrum master and introduced Scrum to both the developers and the customers. During April 2004, the project manager gave two presentations on Scrum. The first presentation at the beginning of April 2004 was a general overview of Scrum while the second presentation at the end of April 2004 dealt with more specifics of the Scrum process.

5.2 Scrum Practices as used at PetroSleuth

In this section the Scrum practices are described as they are typically used at PetroSleuth. To support my description I will include developer descriptions of the typical use of such practices.

5.2.1 The Sprint Planning Meetings

"*We have a list of backlog items that has been reviewed by the client and prioritized with the assistance of the Scrum master / project manager. We review the list, if we have any questions regarding the description we ask, then the development team estimates the time required to produce the result of the backlog item. How we divide the work is that we have the items written on a card and in an order that changes every time. We each get to choose from the table an item we want to do, and we choose in a cycle so each of us chooses a card and then we go through everyone again until either your time is all taken or all the cards are gone. We then mark our names on the card as the primary owner. Then we put the cards down again and choose what cards we want to do again as the secondary. We then go through the same process of choosing and writing our names on the cards until we are full*"

The sprint planning meeting is split into a couple of stages. The first is where the Scrum master and the customer present general goals in terms of what should be accomplished during the sprint. For example: a customer may say, "We want a secure product that can display some charts" as a general goal for what should be accomplished by the end of the sprint. The customer and Scrum master then present a list of backlog items they would like to be discussed for consideration in the sprint. After the backlog items are presented, the team goes through each backlog item and places an estimate against it. This estimate is arrived at by estimating the item as a group.

After the backlog items have been estimated, the team then calculates how many hours worth of backlog items they think they can accomplish during the sprint. To do this calculation, the team does the following:

Load Factor = Percentage of hours the team thinks they will be able to work on development.
The remainng percentage outside of the load factor is used as a buffer for supporting existing products or unexpected bug fixes.

Total Hours Availiable = # Developers * 7.5 hours per day * # of work days in sprint
Availiable Hours In Sprint = Total Hours Availiable * Load Factor

For example, lets assume 4 developers, 19 week days in the sprint, and a load factor of 60%

Load Factor = 0.6 or 60% (60% working on development tasks, 40% support and possible bug fixes)
Total Hours Availiable = 4 developers * 7.5 hours per day * 19 Days = 570 hours
Availiable Hours In Sprint = 570 hours * 0.6 = 385 hours

The available hours in the sprint are the number of hours of work the team thinks they will be able to accomplish during the sprint. The team then goes through the backlog items and discusses whether they want to do that item in the sprint based on the customer's priority and dependencies to other items. If the team decides to do a particular backlog item, it is placed in a pile in the center of the table. This choosing of items continues until the sum of the estimated hours in the pile, is approximately 385 hours.

Once the team has determined what items they would like to work on for the sprint, the customers are brought back into the room and the sprint is presented to them. The customers then approve the sprint or provide feedback for modifications such as changing their priority on items.

Once the team and the customers are satisfied with composition of the sprint, the team has to assign developers to work on the backlog items. To assign developers to backlog items the developers choose backlog items in a round robin fashion. When a developer chooses a backlog item card they write their name on the card as the card's owner (primary developer). After every card on the table has an owner associated with it, the team will go through some of the backlog items again to assign pair partners to the backlog item. Only some of the backlog items are assigned pair partners, because some tasks are determined by the team not to need a pair partner. These non-pair items are usually research-related tasks.

5.2.2 Daily Scrums

"*We come in the room and starting to the left of the Scrum master we each provide a quick synopsis, (sometimes it is not as quick), of what we did the previous day, what we were planning the day ahead, and any issues we had or any concerns we have. We bring those forward. If it needs a longer discussion than just the update or a brief statement then we wait for the end of the meeting and ask people to stay in order to discuss the issues. We all finish our description, and if that's it, the meeting breaks up*"

The daily Scrum meetings are held at the same place (conference room) and time (9:30 am) each day. During the daily Scrum meetings, the developers and the customers both have to answer the same three questions.

1. What did you do yesterday?
2. What are you going to do today?
3. Are there any issues/obstacles?

The developers answer the three questions to update the customers and the other developers as to what they have done, what they were doing, and if there were any issues.

During the meeting, the developers would also mention who they were planning to pair program with and when they would be available for pairing. In the case of the customers, they answer the three questions, but it must be in terms of what they have done for the team or the product. As part of their update, the customers regularly provide an overview of when they would be available during the day for any discussions the developers require with them. Sometimes the daily Scrum meetings would be short (5 minutes) while at other times they would be longer (30-40 minutes). The longer meetings were more common than the shorter ones. If any issues were found, there would usually be a 20 to 30 minute meeting after the daily Scrum meeting to discuss the issues and try to solve them.

5.2.3 The Sprint Review

"*We are in a room with a projector and we are looking at a list of what the sprint objectives are, what the features were that the client was looking for, and we have a demonstration of the features that were produced during the sprint…"*

The sprint review meetings are held in a large conference room with a projector in order to be able to display and demo the software and any features developed during the sprint. For the review, anyone who is interested in seeing the product is invited. The sprint review starts with the Scrum master presenting and discussing what the goals for the sprint were, what the objectives of the sprint were, and whether or not the team had met the objectives. The Scrum master would demonstrate the functionality the team had created. In the initial sprint reviews, the team members had demonstrated the functionality but over the course of the study, the Scrum master took over this role. After the functionality of the sprint was demonstrated, the Scrum master would poll the audience to obtain feedback and suggestions for the product. The review meetings usually lasted between one hour and one and a half hours.

5.2.4 Sprint Retrospectives

"*…we are asked to provide items of what we did well during the sprint and what needs improving, and then we go through the improvement items and determine what action needed to be taken to improve those thing. We also sometimes go through the past action items to decide if we had improved on those items and if actions had been taken.*"

The sprint retrospective ran in the following manner. First, the team is asked for ideas or comments on what they thought went well during the sprint. These are written down on a flip chart. Once the team has finished listing what they thought went well, the team is asked for what they think needs improvement. These are also written down but on a different page of the flip chart. Once the items identified as needing improvement are listed, the team then places a priority on each item to make a prioritized list. The team goes down the list in order, discusses the item and forms actions that can be taken to make improvements. Sometimes, if there were issues brought up during the last retrospective, the team discusses those previous issues to determine if they had been addressed and if not, what action should be taken during the next sprint to address them.

5.3 Sprint Timelines

The sprints were to be as close to 30 days in length as possible. The first sprint took 29 days. The second sprint ended up being 57 days in length. The second sprint, which started on June 4th, was originally planned to end on July 5th 2004 but it was initially extended to July 16th 2004 to accommodate additional work in the sprint. Prior to July 16th 2004, a problem with input data (provided by an outside source) was discovered and the sprint was extended to July 22nd 2004 to try to correct the problem. Eventually it was determined the problem could not be fixed quickly so the sprint was extended to July 30th.

The third sprint, which took 53 days, encountered a problem near its completion. The developers finished all the work allocated in the planning meeting a week early. They went to the customers to provide work to fill in the remaining time. The customers requested work that would take the team two weeks to complete. The Scrum master at

the time was one of the developers as the regular Scrum master was on vacation. He decided that since they could not get the two weeks of work done in the week remaining the sprint would be extend the sprint until September 15th 2004. During the sprint three retrospective, both the customers and developers agreed that there would be no more extended sprints as they had caused too many problems in terms of the developers not knowing what they needed to do.

The numerous extensions of the sprints can be seen as how to use Scrum from books rather than experienced consultants can cause problems. It was decided for all future sprints, if the work could not be completed in the sprint, it would either be trimmed down or moved to the next sprint.

For sprints four through ten, the sprints were set very close to 30 days in length. Sprint number seven was 39 days as a week was added onto the regular 30 days to make up for the week of work days lost to the Christmas break. Sprint eleven was short due to the customers and the team wanting to have a more responsive sprint during the period of testing and bug fixing immediately before the planned external release of the product at the beginning of May.

Table 5-1 reflects the start and end dates for each of the sprints completed as of April 30 2005.

Table 5-1: Sprint Timeline

Sprint	Start Date	End Date	Length calendar days
1	May 3rd 2004	May 31st 2004	29
2	June 4th 2004	July 30th 2004	57
3	July 31st 2004	September 21st 2004	53
4	September 22nd 2004	October 21st 2004	30
5	October 22nd 2004	November 20th 2004	30
6	November 21st 2004	December 23rd 2004	33
7	December 24th 2004	January 31st 2005	39
8	January 31st 2005	March 1st 2005	29
9	March 2nd 2005	March 28th 2005	27
10	March 29th 2005	April 18th 2005	20
11	April 19th 2005	April 29th 2005	11

5.4 RESULTS

In this section both quantitative and qualitative results of the Scrum process over the course of twelve months is discussed. In addition I review the amount of overtime worked by the developers, how the developers do estimation, and how much time they spend for sprint planning, sprint reviews, and sprint retrospectives. This section also provides developer and customer opinions about the use of Scrum are also provided within this section.

5.4.1 Mean Percent Overtime Worked by Team

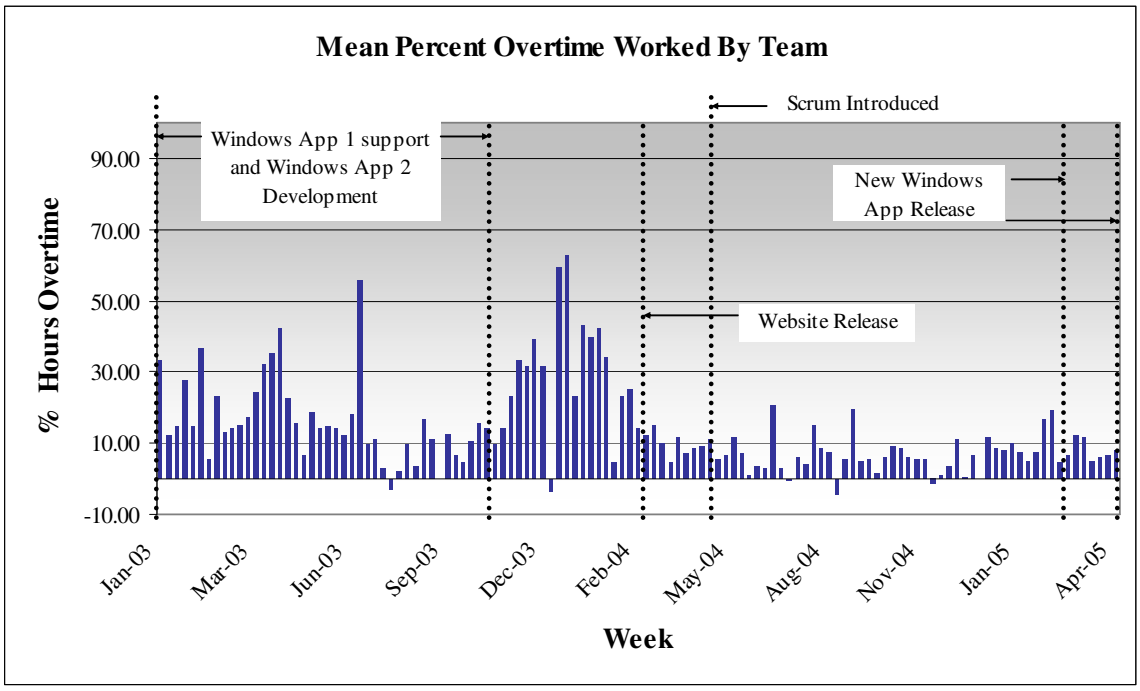


Figure 5-1. Mean Percent Overtime Worked by Team

Figure 5-1 outlines the mean percent of overtime worked by the software developers as a team on a weekly basis. This means a given percentage is the percent overtime per developer per week. From Figure 5-1, it is evident that before the introduction of Scrum there were periods of overtime spikes. In the first period before May 2003, the team was working to enhance the first windows application. The other period of significant overtime was in late 2003 when the team was developing the website application.

After Scrum was introduced, there were still overtime spikes but these were of smaller magnitude and shorter duration. In Table 5-2, I have compared the mean percentages and variance in overtime before and after Scrum was introduced.

Table 5-2. Scrum Overtime Before and After statistics

	Before Scrum	After Scrum
Mean percentage overtime worked	19	7
Standard Deviation	14	5
F Test for variance: F= 9.11 Dfn=68 DFd=51	p= < 0.001 (one tailed)	
T Test (unequal variance): DF=87	p= < 0.001 (one tailed)	

One of the study questions asked was; does Scrum provide a sustainable development pace? A sustainable pace means the developers work close to regular hours with overtime that is not highly variable. An F-Test was performed with the following hypothesis:

H_O: there is no difference between the two variances

H_A: The larger standard deviation in overtime before Scrum was introduced is significantly different from the smaller standard deviation in overtime after Scrum was introduced.

The F-Test showed at that the standard deviation before Scrum was introduced was greater than the standard deviation after Scrum was introduced indicating there was more stability with the overtime worked after Scrum was introduced. A T-test was also performed on the following hypothesis:

H_O: there is no difference between mean percentage of overtime worked before and after Scrum was introduced

H_A: The mean percentage of overtime worked before Scrum was introduced is greater than the mean percentage overtime worked after Scrum was introduced.

The T-test assumed that the variances of the period before and after the introduction of Scrum were not equal as supported by the F-Test. The T-test shows the

mean percent overtime after Scrum was introduced was less than the mean before Scrum was introduced.

5.4.2 Estimating of Backlog Items

How good are the estimates and do they change over time? The following chart, Figure 5-2 shows the difference in hours between what was estimated and what was actually worked on backlog items (that were estimated during the planning session).

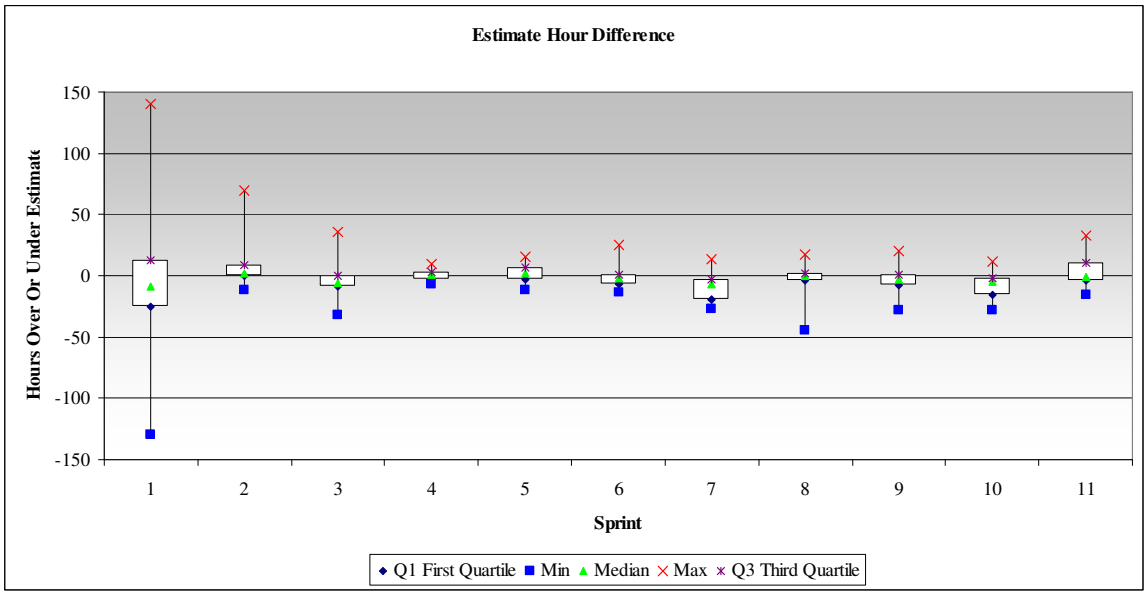


Figure 5-2: Hours Misallocated due to Over or Under Estimation

In Figure 5-2 a negative number indicates that the backlog item was overestimated meaning there were fewer hours put into the backlog item than the estimate allocated. Positive numbers indicate that the number of hours worked on the backlog item was more than the estimated allocation of hours.

In Figure 5-2 there are some large over and underestimates. For example, the minimum and maximum extremes in the first Sprint were caused by two backlog items. One backlog item was underestimated by 140 hours and another item was overestimated by 130 hours. The overestimated backlog item was originally estimated to take 150 hours

but only 20.5 hours were worked on the backlog item meaning an overestimate of 130 hours. One possible reason for this large overestimation is the backlog item was similar to another backlog item that was underestimated by 140 hours. The underestimated backlog item was estimated to take 75 hours, however 215 hours were recorded to the backlog item. I believe that the work from the overestimated backlog item was recorded on the underestimated backlog item instead.

After the first sprint, the amount of hours being misallocated decreased but then fluctuated for the remaining sprints. When the absolute value of the misallocated hours is presented, the same trend can be seen. In the first sprint, there was a large range of misallocation while later sprints had less misallocation. Figure 5-3 also shows that the amount of misallocated hours fluctuates through out the sprints.

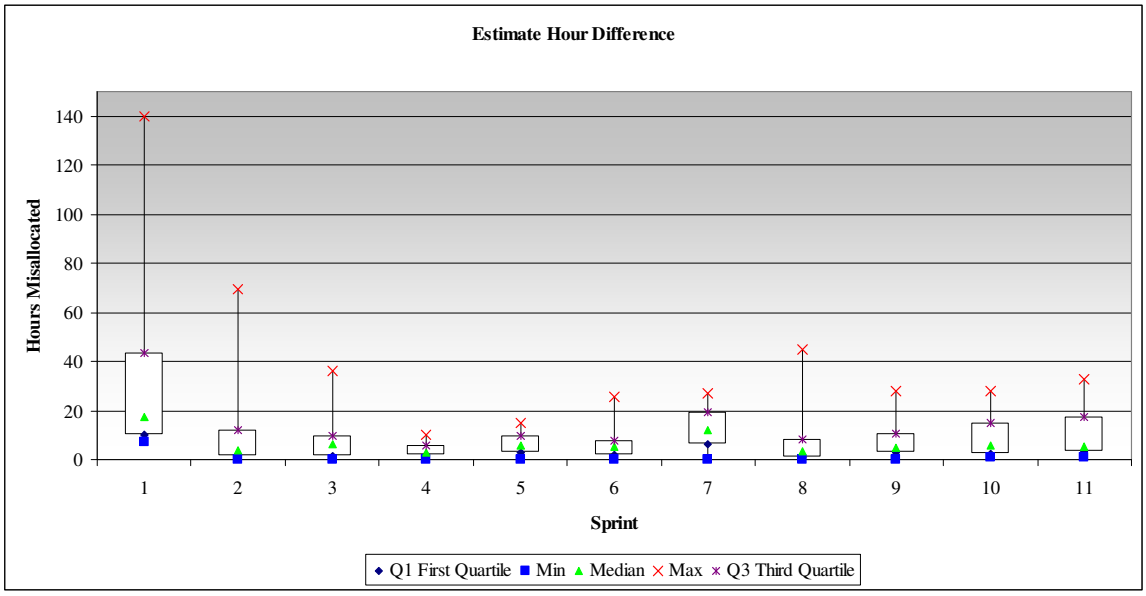


Figure 5-3 Hours Misallocated By Sprint

The results presented so far have shown the misallocation of hours has fluctuated through the eleven sprints. What could be a possible factor that may influence how many hours are misallocated due to poor estimation? Figure 5-4 is a scatter plot showing the amount of hours misallocated and the size of the estimate for all backlog items estimated during the sprint planning session.

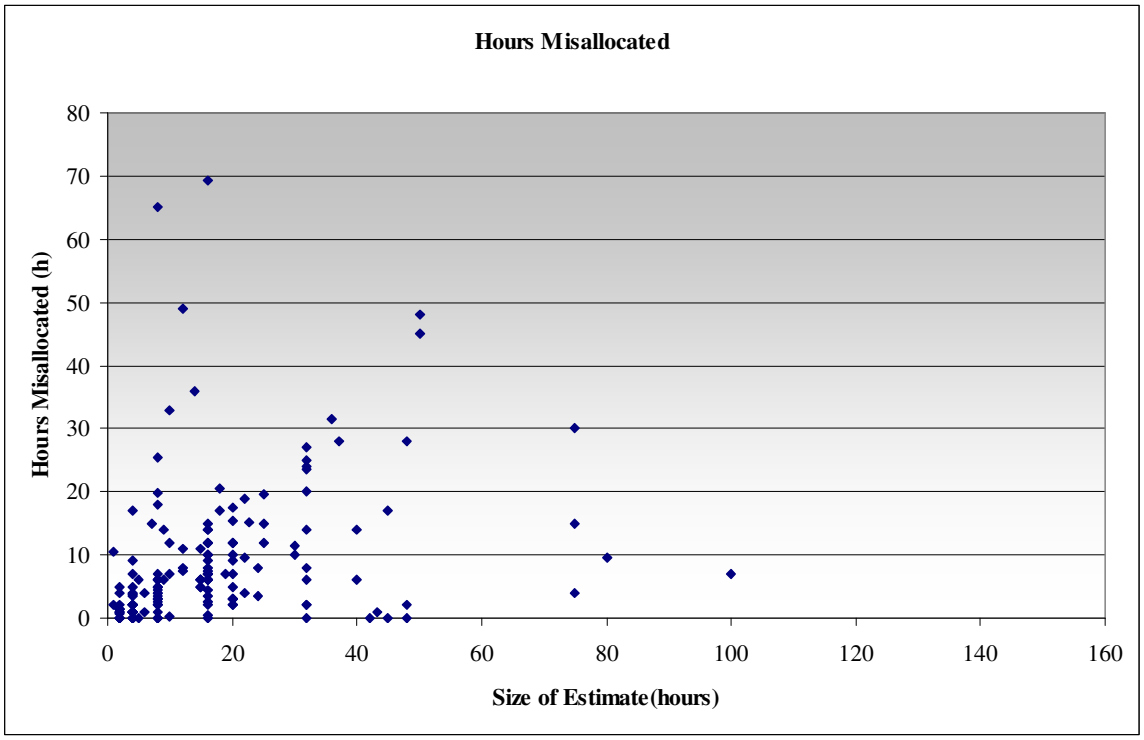


Figure 5-4 Hours Misallocated vs Estimate Size

The correlation coefficient of hours misallocated and size of estimate is shown in Table 5-3. The table shows a moderate statistically significant correlation between hours misallocated and size of estimate. This indicates that in the study, smaller estimates have fewer misallocated hours.

Table 5-3 Correlation of Hours Misallocated and Estimate Size

	r	**N**	**Statistically Significant, p**
Pearson	0.54 (moderate)	167	Yes, p: < 0.0001
Spearman	0.49 (moderate)	167	Yes, p: < 0.0001

Another relationship to consider when discussing estimation is the amount of estimation error and the size of the backlog estimate. One possible explanation for smaller tasks misallocating less time is that the estimates are more accurate on smaller tasks. Figure 5-5 shows the percentage error in estimate and the size of the backlog estimate.

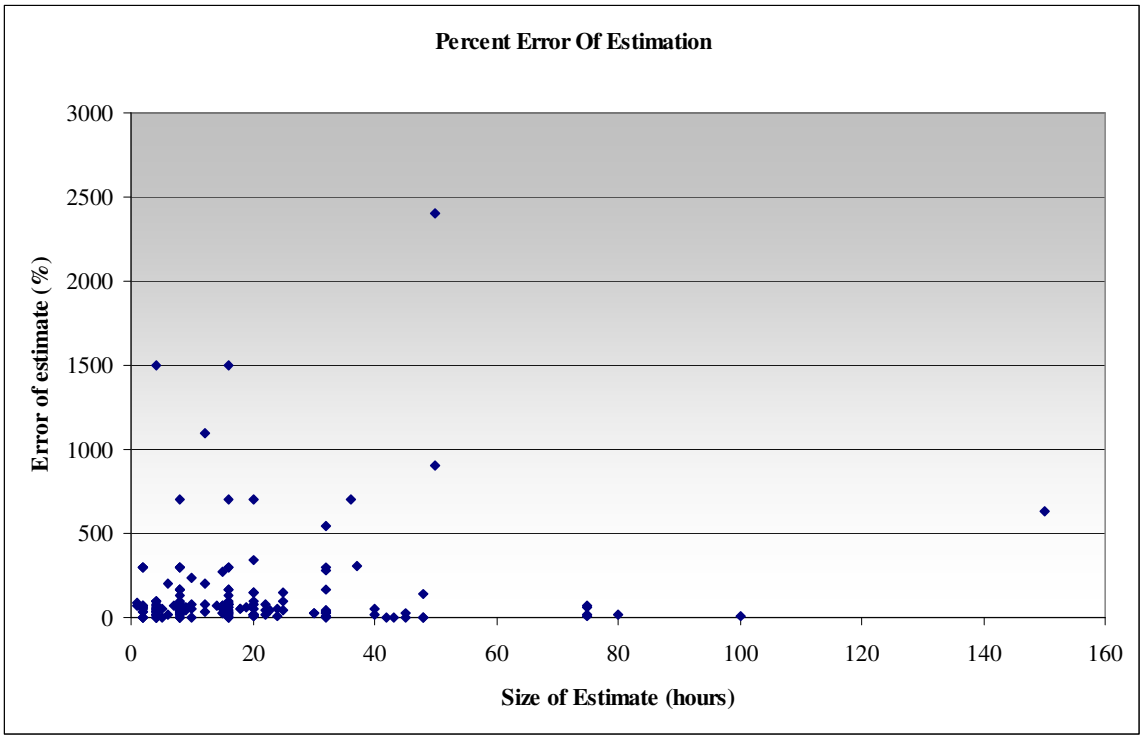


Figure 5-5 Estimate Percent Error vs Estimate Size

The correlation coefficient of percent error of estimate and size of estimate is shown in Table 5-4. The table shows a weak non-statistically significant correlation between the percent error of estimate and the size of estimate indicating that there may not be a correlation between the size of the estimate and the percent error of the estimate. This result indicates there is not a significant relationship between the size of the backlog estimate and how accurate developers are in estimating the item.

Table 5-4Correlation of Estimate Percent Error and Estimate Size

	r	**N**	**Statistically Significant, p**
Pearson	0.13 (weak)	167	No, p: 0.08
Spearman	-0.02 (weak)	167	No, p: 0.78

5.4.3 How Was the Amount of Work Distributed by Activity

In this section, I discuss how the work was recorded in terms of backlog items, sprint planning meetings, sprint review meetings, and sprint retrospective meetings. The rationale for presenting this information is twofold. First, Figure 5-6 will give an indication as to how well the team recorded their hours working on backlog items. Second, Figure 5-6 will show how much time was recorded as sprint planning, review, and retrospective time, to give an indication as to how much time was taken up with these processes.

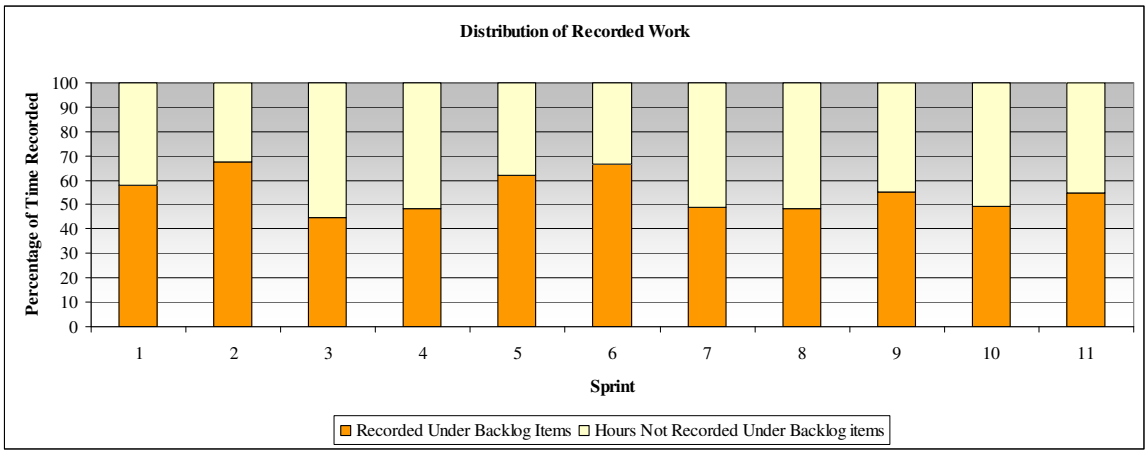


Figure 5-6: Amount of Work Recorded in Version One Vs Office hours

On Figure 5-6 the bottom yellow bar indicates that the time is recorded as work completed during a sprint as a percentage of the total number of office hours recorded in the internal time tracking software. For example, if the sum of the teams' office hours (Section 3.2.1) in a sprint was 100 hours but only 60 hours was recorded for backlog items the percentage of time recorded would be 60%.

Figure 5-6 shows, over the course of the eleven sprints, the percentage of time recorded fluctuates between 54% and 76% with the median percentage of 60%. This indicates that between 24% and 46%, with a 40% median, of the office hours were not recorded as work done on backlog items. During the final interview, the developers were asked what kinds of time do not get recorded on backlog items. The developers all said "meetings". One developer said:

"The [daily] Scrum meetings are not entered on VersionOne. I would say various ad hoc meetings. It depends on what people consider their time. Sometimes the meetings should be included as part of the task. Sometimes you make work on a task that is not in the system and sometimes your time will disappear. I know we are not supposed to do that but it happens."

Through observations, I noted that there are usually a large number of informal ad hoc meetings that take place during the day to discuss different aspects of the system. These meetings are not recorded in the system. Also not recorded in the system are the daily sprints that range from 15 to 30 minutes in length (i.e. approx. 3-7% of the daily office hours). The amount of time the project manger spends working with the customers, acting in a business analyst role is not recorded on backlog items.

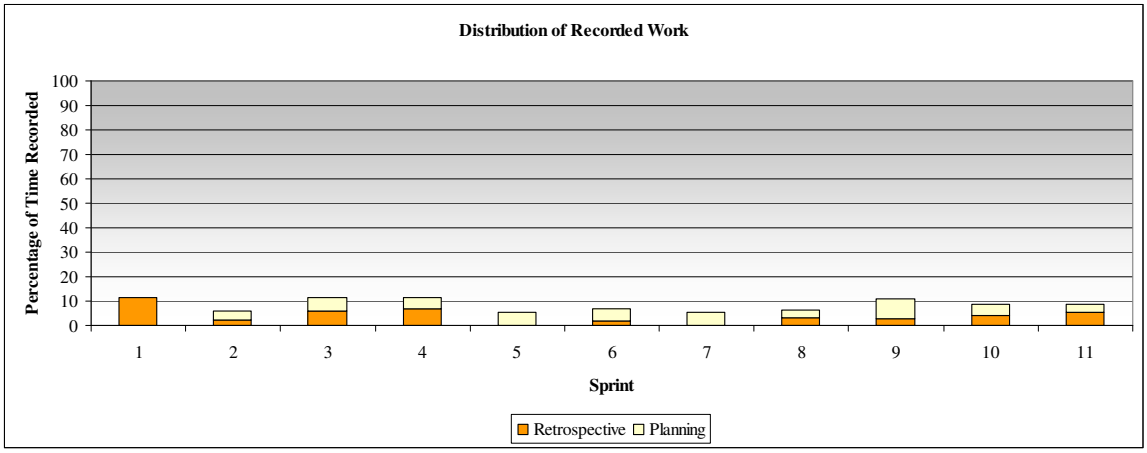


Figure 5-7 Percent time for Planning, review, retrospective meetings

Figure 5-7 shows the percentage of office hours spent in sprint planning, sprint review and sprint retrospective meetings. Figure 5-7 shows the percentage of the office hours, spent in sprint planning sessions, ranged from 3% to 8% with a median of 5% and a standard deviation of 1.4. The small range and standard deviation indicates that the amount of time spent in the planning sessions was fairly constant throughout the eleven sprints. The percentage of office hours spent in sprint reviews ranged from 1.7% to 6.7%, with a median of 4.7% and a standard deviation of 1.8. Figure 5-7 only shows June 2004 forward for planning session percentage because before June 2004 the meeting times were not recorded however if the weekly meetings are taken into account the developers would spend 3% of their time per week in the weekly meetings. These results

from Figure 5-7 indicate the team was spending at least 4.7% of their office hours in Scrum related processes.

5.4.4 Customer Opinions

Here I present qualitative results from customer feedback provided in Scrum questionnaire Q5. The overall feedback I received from the customer group was positive. All three of the customers said they would recommend using Scrum in the future, though they would like to see modifications made to the process. Of the current set of customers, some were involved with the previous releases of software in terms of testing and verification but were not as involved with the decisions relating to functionality or usability. This situation was common prior to Scrum. The customers were used to check the final result of development rather than direct what should be developed.

When asked how satisfied they were about the software developed before Scrum was introduced the customers said that they were not really part of the process and did not care for the software produced. One customer said he was "*Ambivalent,* [the] *product was alright, not great"*. Other customers had stronger opinions. "*The release of the website was quite honestly a nightmare. Very little time was spent previous to release in the verification of data and results…*". The overall theme from the customers was they were not very involved with the software produced before Scrum and in most cases, were not satisfied with what was produced. Since the introduction of Scrum, the customers have been more satisfied with the software developed and the development process.

When asked how satisfied they were with the software produced after Scrum was introduced one customer was "*Very satisfied"*. Other customers responded by mentioning other benefits Scrum had brought: "*I believe there has been far greater consistency, transparency, and coordination since the implementation of Scrum*". Another customer mentioned that they were much more involved in the process than before, *"The release of the current WellLoad (not quite complete) has thus far been a far more thorough process. The initiation of the Scrum process has lead to our being more involved in the daily*

review and discussion. This has lead to us being more aware, and being held accountable earlier in the process for any changes and concerns that have or had to be considered."

The customers said the Scrum process changed how they interacted with the developers. Some customers gained more respect for the software developers, "*I have a greater respect for the software developers and understand how easy it is for expectations and results to differ without clear instructions and regular communication between all parties".* On a similar question about what the customers expect from the software developers one customer had this to say: "*I have found that I expect a much more interactive relationship between us and the programmers. This is facilitated quite well with the location of the programmers being in our office, but the Scrums have made a portion of everyone's day where they are accessible*"

The customers said they like the sprint planning meetings. "*Superb forum for planning; the whole team is involved and thus everyone knows what is required from them*". One customer mentioned how he thought that the planning meetings prevented problems later on: "*Although the day as a whole can be a very tiring process, I have found that the time spent in the planning meetings has lead to less misdirected development and a more clear understanding of both the requirements and the limitations of the development process by both the customers and the programmers*". One customer noted however "*Initial meetings need to be comprehensive but with a project spanning many sprints the time taken for these meetings needs to be reduced particularly those involving the client. The software developers may need to spend more time on this without involving the client."* Some of the features that this customer was heavily involved in were left out for future work. For him, the meetings may objectively not be as relevant as for the other customers. This could be part of the fact that in the earlier sprints a lot more technical discussion took place with the customers in the room rather than the current practices of letting the customers leave while the team discusses technical issues.

The customers also liked the sprint reviews and retrospectives, "*While the Scrum process has often made much of the accomplishments in a sprint to be known before a*

sprint review, it has helped us as customers [to] *see visually the product, and we are able to see these earlier in the process again. This has lead to the ability to "tweak" and change the product in a more timely fashion. We as customers have found it difficult to visualize the product ahead of time, and some concerns have only arisen when the demonstrations have been shown*". One customer linked the review and retrospective to accountability by the software developers for what work they take on. "*This is a great opportunity for the programmers to demonstrate the accomplishments and leads to proving their own accountability on their tasks that they took on*".

Since the customers did not get involved with the software development decision process until after Scrum was being used by the development group, many of them either did not comment on how the transition went or said that they came on board after Scrum was introduced. One customer was part of the previous process and commented that the previous way of developing software was so bad that they were willing to try anything. "*I believe the complete frustration with the PetroCube website release made it clear that a more responsive system was needed and most were ready for anything that would have made the system better*".

The customers found that using the Scrum master / project manager in a business analyst role, between sprints, helped them be more prepared for the planning meetings. "*It has lead to a more timely completion of the planning stages earlier. In the first few sprints we were never prepared when it came to being ready for the next planning session. But with the project manager actively starting the planning, and long term visions ahead of time has made this planning session easier*".

The customers found the daily Scrum meetings allowed them to be kept up to date on the progress of the software development and to be informed on issues as they happen rather than at the end. "*Good forum for hearing progress updates and what issues/problems are lurking*". One customer commented, "*They have helped us as customers stay in the loop and have a better idea of when I should expect questions*". Another customer said that "*Perhaps they should be called a 'huddle' and involve only*

the developers with one representative of the client". In this case the customer saw the Scrum daily meeting as something that only those who need to attend should attend. Such a system would be optimal, except for the fact that in order to update the people who need to know about changes, they have to attend.

When asked if there were any difficulties using the Scrum process, one customer said that it was "*too rigid*", while another said, sometimes it was difficult for him to understand what tasks the developers were doing at times.

When asked for any suggestions on how to change the process, only one customer had a comment, *"…the daily meetings, which I think are too frequent. They need to be held as required depending on the type and status of the project."* The Scrum meetings are not just status meetings, they are used to figure out what resources need to be acquired during the day to facilitate the software development and therefore should be done everyday.

5.4.4.1 Developer Opinions

In this section the developer views and opinions about different aspects of the Scrum process are presented. This is done from both the questionnaires and questions answered on the final wrap-up interview.

Questionnaire Responses

The developers found the Scrum process very beneficial. From the questionnaires every developer would recommend using Scrum on projects in the future. There was a range of satisfaction about the software before Scrum was introduced. Some developers were satisfied with the software, while others were not satisfied at all: "*Some amazing work was done before the Scrum process was introduced. Not necessarily the best code*", while others said "*I was very unsatisfied with the product developed before the Scrum process and would not like to be associated with any of the products produced at that time*".

After the Scrum process was introduced, the developers as a group were more satisfied with the products being produced: "*I am very satisfied*" and another developer said "*Very satisfied with the software product(s) being developed*".

The developers saw that the Scrum process had fostered more customer involvement and communication: "*It promotes better communication with the client*" and "*It is useful to see customer representatives every day, since there are always questions that could be asked…and it makes me more confident in what we're doing because customer always have up-to-date information about the progress*".

The developers found the sprint planning, the review, and the retrospective meetings helpful. "…*These meetings are useful since we can choose the scope of work, although guessing on the time for each backlog is not always easy and time estimates do not always come out right*". Although the developers found the planning meetings useful they did notice some room for improvements: "*The only negative is that these* [meetings] *take a little too much time and it's hard to concentrate*". One developer noted that the customers were not as prepared as they should be "…*I feel that our Scrum master/project manager and customer are not prepared enough for them* [meetings]*, so the meeting drags on*".

This feedback was noticed in previous sprints, so for Sprint 8 the Scrum master worked with the customers during the sprint. The developers had this to say: "*Someone (being the project manager or a BA [business analyst]) needs to help the client formalize their thoughts on what they want. The development team should not decide for the client on what needs to be done. The client needs to make the decision. In order to have the client make a thoughtful decision all alternatives need to be explained in the client's language as well as pros and cons for each option. So I guess my answer is that it did help. There is more work to do on that aspect, but generally the last sprint planning was much better that way*"

The developers found the review and retrospectives useful. One developer liked the review because it was a place to show off what they had done and get feedback on it: "*Useful, because not only the customer can see what we have achieved so far, but all the developers can see how it all works (or doesn't work) together. Plus we can hear some useful questions/suggestions for the improvement*". One developer compared the review meeting to the previous process where there were weekly meetings: "*It replaces a weekly meeting and is more efficient*".

For the most part the developers like the retrospectives, but there was still the problem of the meetings not being as focused as they should be. "*Sometimes we don't keep focused and it goes on too long, but it's a good concept*".

The developers found the daily sprint meeting useful but also found that the meetings would drag on if the team is not focused on doing them quickly. One developer said "*I don't like them right now, In our group it seems to, truthfully I think, certain people talk too much, its too low of a level, or off topic, for what we want I really would like to see it down towards the 5 or 10 minute mark but really these days its 30 minutes or more. It just wears on and people loose focus for what we want to use it for. I don't really think the way we have implemented it is doing its job right now*". Another developer stated "*Usually very useful. Good to know what everyone is doing and how we are doing in terms of delivering on time as a team*".

The developers also commented on their confidence in the software they were producing after Scrum was introduced: "*…before Scrum I was not as confident since nothing was reviewed by the manager who gave the work and I had no exact deadline. The Scrum process makes me more confident since I know exactly what I'm doing and when it is due (and that it is probably doable in the time given)*". Another developer said: "*The Scrum process is giving me confidence that we are developing the software that the customer wants…*"

The developers were asked how they found the transition to Scrum. The developers all said they either did not find the transition difficult or were not part of the transition (developer added November 2004). One developer commented "*On the whole the transition to [the] Scrum process was not hard. The most difficult component is ensuring sufficient planning is done for the long term and for each sprint*". Another developer said "*I didn't find it difficult since I haven't worked here long before Scrum...*".

The developers were asked two questions related to what difficulties were encountered in using the Scrum process for the first time and what improvements could be made to the current Scrum process. Generally, the comments focused on the planning portion of the process and having the customers come prepared with what they would like done during the sprint. One developer commented "*Customer doesn't always give us answers that we need right away. Customers [have] no idea what they want*" while another developer said *"…The bulk of the tasks should have been defined by the client before the meeting and the team would remove or add tasks depending on the introduction the client made (introduction being what they would like to see during the sprint*", and "*Split the role of business analyst and project leader. Too much cross-contamination is occuring, influencing how the needs of the customers are translated into requirements*". The project manager had this to say about difficulties or improvements to the Scrum process "*The major difficulty I had was determining the right amount of planning to do for the sprint planning session. At the start, the sprint planning sessions were drawn out and took a lot of time. By doing more pre-work with the product owner, more was achieved in a shorter duration*".

Interview Responses

In the final set of interviews given to the developers, they were asked if they had any comments on the Scrum process. Some of the developers did not have any additional comments while others did. One developer said this about the Scrum process:

"*I have been in IT for a long time, so I am over 15 years now. And it seems to me that this method is much better at keeping everybody on track and focused. Both the customer and the developer in terms of developing what someone actually needs and, on the*

customer side, them understanding the costs of what they are asking for. It seems to be a lot tighter, more face to face with each other and overall much better communication and understanding between the two parties".

5.5 Discussion

Quantitative and qualitative results based on the use of Scrum at PetroSleuth have been presented in the above section. I will now discuss questions relating to these results:

Does using Scrum provide a sustainable working pace to developers?

I define a "sustainable pace" as an environment where there is little or no overtime and the hours worked are stable in terms of hours worked above or below regular work hours. The qualitative results from the developers indicate Scrum provides better time management through deadlines and better customer communication. By giving the developers deadlines and communicating with the customers about what is expected, the developers should have the ability to manage their time across multiple tasks for the sprint. In my opinion, this time management allows the developers to leave an unfinished task until the next day, rather than work overtime to complete it with the knowledge they will still meet the sprint objectives. The quantitative results indicate Scrum provides both a decrease in the variance as shown in the F-test and a decrease in the mean amount of overtime worked, as shown by the T-test (Table 5-4). Based on my observations, before Scrum, there was confusion as to what needed to be completed and when it needed to be finished. Communication between the developers and customers about what was feasible in a given amount of time was at best, poor. This lead to un-realistic workloads being placed on the developers by the customers After Scrum I observed a more relaxed team that knew what needed to be completed, when, and had negotiated with the customer about what could be done in the time available. In my opinion this is a more sustainable environment because the developers are not being burnt out with large amounts of overtime and their hours are stable, meaning they can make plans outside of work hours.

The results have shown that Scrum leads to a more sustainable pace for the developers. One concern a customer may have with respect to the developers working

less overtime hours, is developers may be getting less work done for the customers over the course of a sprint. This may cause the customer to be less satisfied than when the developers were working more overtime. The next question will discuss the customer's opinion to see if this is the case.

How will Scrum be viewed by the customer group?

Prior to Scrum the customers felt "out of the loop" and were ambivalent and unsatisfied with the software being produced. During the period prior to Scrum, I observed the customers were not engaged in the process and were mostly used for validation and verification of the software produced. This meant the development team was controlling the requirements rather than the customers. This lead to products that had functionality the customers could not use, because important requirements from the customers' perspective were missing. The mismatch in requirements caused frustration in both groups. The customers had a product that they could not fully utilize and the developers were wondering why their product was not being used.

After Scrum was introduced, the views of the customers changed. The customers liked the Scrum process. They found the daily Scrum meetings kept them up to date and the planning meetings helped to reduce the confusion about what was being developed. The customers therefore, felt more involved in the software being developed. They were more "in the loop" and had a more defined role in the software process. The customers went from a role of testing and verification to a role that exercised control over what features were to be produced based on their needs. Customer responses indicated the customers' satisfaction has increased after Scrum was introduced. Communications also increased, which has lead to less misdirected development. The use of Scrum has transformed the customers from being passive consumers to active participants in the process. This, in my opinion, has made them more satisfied with what is produced by the team. The fact that the developers may not be producing as much product as before Scrum does not seem to bother the customers. They seem pleased with the product being developed, as they are receiving the software they ask for on time. This result indicates that even though the developers are not working as much overtime as before, the

customers are actually more satisfied with the software created. From this result I can formulate the following hypothesis to test in future studies: Scrum will reduce the amount and variance in overtime while increasing customer satisfaction.

How accurate are the backlog estimates and does this change over time?

Initially the accuracy of the backlog item estimates was poor (wide range of misallocated hours) as seen in Figure 5-2 and 5-3. After the first sprint, the estimates seemed to get better with the range of misallocated hours getting smaller. However, even though they got smaller, the variance in hours misallocated fluctuated significantly (Figures 5-2 and 5-3). From the quantitative results, the fluctuation is moderately correlated with the size of the backlog item estimate as seen in Table 5-3. This means, larger items had more misallocated hours. If this result is found in general, the result would argue for smaller backlog items to be estimated. If smaller backlog items have less misallocated hours, then by estimating smaller backlog items, fewer hours should be misallocated. This thought process leads to the hypothesis: Smaller backlog estimates will decrease the amount of misallocated hours.

The relationship of smaller backlog items and smaller amounts of misallocated hours could also be the result of smaller backlog items being estimated more accurately. If smaller backlog items are estimated more accurately than larger items, there should be a relationship between backlog estimate size and error percentage of the estimate. To check this possibility a correlation coefficient was calculated in Table 5-4 to look for a relationship between the backlog estimate size and the percent error in the estimate. The correlation coefficient calculated showed a weak, non-statistically significant correlation between task size and percent error in the estimate. This indicates that there is no significant relationship between backlog item size and the percentage error in the backlog item estimate. This results also argues for smaller backlog item estimates, because if the percent error for backlog items are going to be variable regardless of size of the estimate, it is better to have smaller hours estimated in order to misallocate less time.

How well do the customers and the developers adhere to the Scrum practices?

Based on observations and the length of the sprints in Table 5-1, I can say the team adhered to some of the practices well. These practices were sprint planning, daily Scrum meetings, and sprint review. They followed these practices well as they used them at the appropriate times and followed them to a fairly well in comparison to what is recommended in literature. For example; the team put significant effort into the sprint planning, review, and retrospective meetings as can be seen in Figure 5-7. However, there were some problems. One practice not followed was having fixed length sprints. When discussing why some sprints were extended, a developer indicated that they were extended so the customers and the team did not have the perception of failing to achieve their sprint objectives. This response seems to go against the notion of collaboration and negotiation between the customers and the developers. If problems arise, they should be dealt with, time permitting, in the current sprint. If not, they should be moved to the next sprint. By extending the sprints the team was going back to a chaotic state where no one knew when the next release of the software would be and what would be expected for that release. I think another reason fixed length sprints were initially difficult, was that, in the past, the team had extended release dates to get functionality into the product. So in their opinion, extending deadlines was an accepted way to meet goals. After extending two sprints (sprint 2 and 3) the developers and customers realized that they did not want to extend sprints any longer and decided to stay with fixed length sprints, as close to 30 days in length as possible.

What issues were encountered when using Scrum and what are some solutions?

There were two main issues that were encountered by the team during the use of Scrum. The first was that the customers were not prepared for the sprint planning meetings. When the fourth sprint started, the customers did not know what they wanted or even what could be done for them in terms of future work. This caused significant frustration with the developers and also with the customers. The solution to the problem was found to be twofold. The first part of the solution was to hold a few days of tutorials where the developers provide some possible ways they could move forward on developing software based on customer feedback. The second part of the solution was to

have the Scrum master work with the customers during the sprint to get them prepared for the next planning meeting. This preparation took the form of the Scrum master acting as a business analyst to help the customers understand what is possible and what implications their choices had for both the development team and themselves. After the project manager started to work with the customers, the customers were much more prepared for the planning sessions.

A second issue was the team loosing focus in Scrum related meetings. For example; the developers said a few of the daily sprint meetings were too long and not focused. At other times the review and retrospective meetings seemed to drag on and not be focused. The solution to this was to raise the issues during the retrospectives and remind people to try to stay more focused on keeping the meetings on track.

What are the opinions of the developers using Scrum?

Prior to the introduction of Scrum, some of the developers were unsatisfied with the products being produced. Based on observations and experience developing software prior to Scrum, the team was working with few deadlines, a large list of functions to implement and no plan on how to accomplish the goals. After the introduction of Scrum the developers have a better idea of what they are working on and when requirements need to be completed. They are more satisfied with the products being produced and are more confident that they are producing the kind of functionality the customer requires. I believe this knowledge has increased the morale of the team and allows them to focus on providing the customers with what they need versus what the developers think the customers want. The team is now confident that they are building the right product after Scrum was introduced, as opposed to before Scrum when the developers created products the customers could not fully utilize.

The developers also believe there is better communication between themselves and the customer. I think this increase in communication has caused both the developers and the customers to know what is expected from each other. The management of expectations has benefited both the developers and the customers.

One part of the Scrum process the developers did not like was the fact that they now had to make estimates for their work and that those estimates would be used to plan the work for the Sprint. Before Scrum, estimates either were not made or the estimates were not used to plan when a release should happen. The developers also thought some of the meetings were too long.

CHAPTER SIX: PAIR PROGRAMMING

In this chapter, qualitative and quantitative results based on the experiences gained by introducing pair programming into PetroSleuth will be presented. I will first discuss how pair programming was introduced, then provide quantitative and qualitative results related to the use of pair programming. In this chapter the following questions will be discussed:

- **How often will developers use pair programming?**
- **What possible factors could influence the amount of pair programming**?
- **What do developers think about pair programming?**
- **What do developers use pair programming for?**
- **What do the developers think could be done to make the pair programming experience more enjoyable?**

Chapter 6 is structured in the following manner: In section 6.1, I present how pair programming was introduced into PetroSleuth. In Section 6.2, I discuss some initial observations while the developers started to use pair programming. In Sections 6.3 and 6.4, the quantitative results are presented. In Section 6.5, the qualitative results from developer questionnaires and interviews are discussed and in Section 6.6 I discuss the above questions.

6.1 The Introduction of pair programming

At the end of January 2004, after receiving ethics approval, a short presentation on pair programming was given by myself to PetroSleuth. For the PetroSleuth presentation on pair programming, materials from Dr. Frank Maurer's course on agile methods were used. During the presentation I discussed how pair programming should be done, I emphasized that the developer who is not at the keyboard (the navigator) was not

supposed to just watch but to be thinking ahead and trying to consider issues with the code. I mentioned that pairs should be switching occasionally so that the same person was not always controlling the keyboard and mouse (the driver). As part of the presentation, I stated that, ideally, all code that was to go into production should be pair programmed so the code is reviewed before going into the product.

6.2 Getting started with pairing

Initially the developers would form impromptu pairs throughout the day. This impromptu pairing happened when they encountered difficulty on their task or were doing similar tasks. The initial pairing would last anywhere from a few minutes to a couple of hours. In this case pair programming is when two developers work on completing a task. When pair programming initially started, the pairs would sometimes request assistance to understand the process.

To assist the developers I interacted with the pairs by offering suggestions on how they could pair more effectively. For example, initially the navigator would be watching the screen observing what the driver was doing. During these sessions the room was completely silent, which is not very collaborative. To increase collaboration I would get both programmers to continue but I asked questions to driver and offered suggestions to the code. At times I would ask him to think out loud more as I could not read his mind and needed to know what he was doing and why he was doing it

During the initial stages of pair programming, the programmers had difficulty talking to their partner while developing. One reason is that when developing alone there is no reason to vocalize your thought process. However, when developing with a partner there needs to be communication between each other as your partner cannot read your mind. Over time, the developers improved in discussing their thoughts with their partners while programming.

Another difficulty was that the dominant partner would usually take control of the keyboard and not give the other partner time to act as the driver. After some reminders to

share the amount of keyboard time, the pairs seemed to share the keyboard and reverse their roles a couple of times per day.

6.3 Amount of Time Pairing

In the following two sections, I will provide an illustration of how much time the developers spent pair programming both as a percentage their time spent doing tasks during the sprint (Figure 6-1) and as a percentage of their total work day (Figure 6-2). This information will be used partially to answer the question of how rigorously the developers used pair programming and to lead into a discussion of what could be some influencing factors.

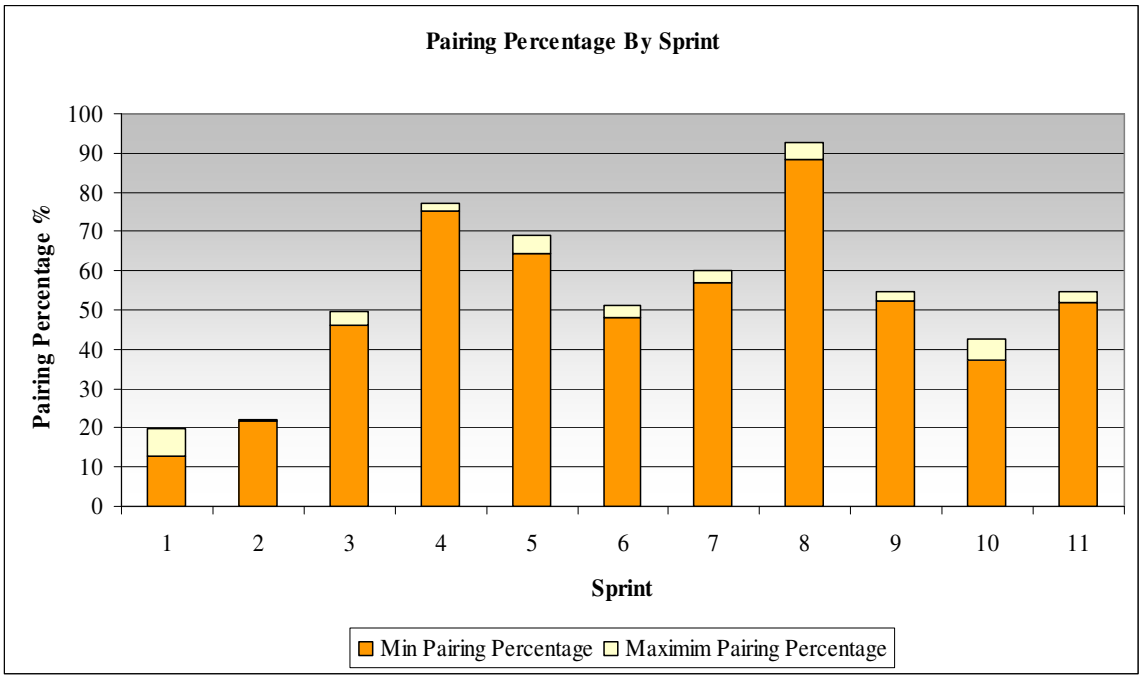


Figure 6-1 Percentage of Pairing By Sprint

Figure 6-1 shows the pairing percentage based on the amount of time recorded on backlog items during the sprint. The size of the beige bar on top shows the differences between what the pairs record on their pair programming sheets found in Appendix D.

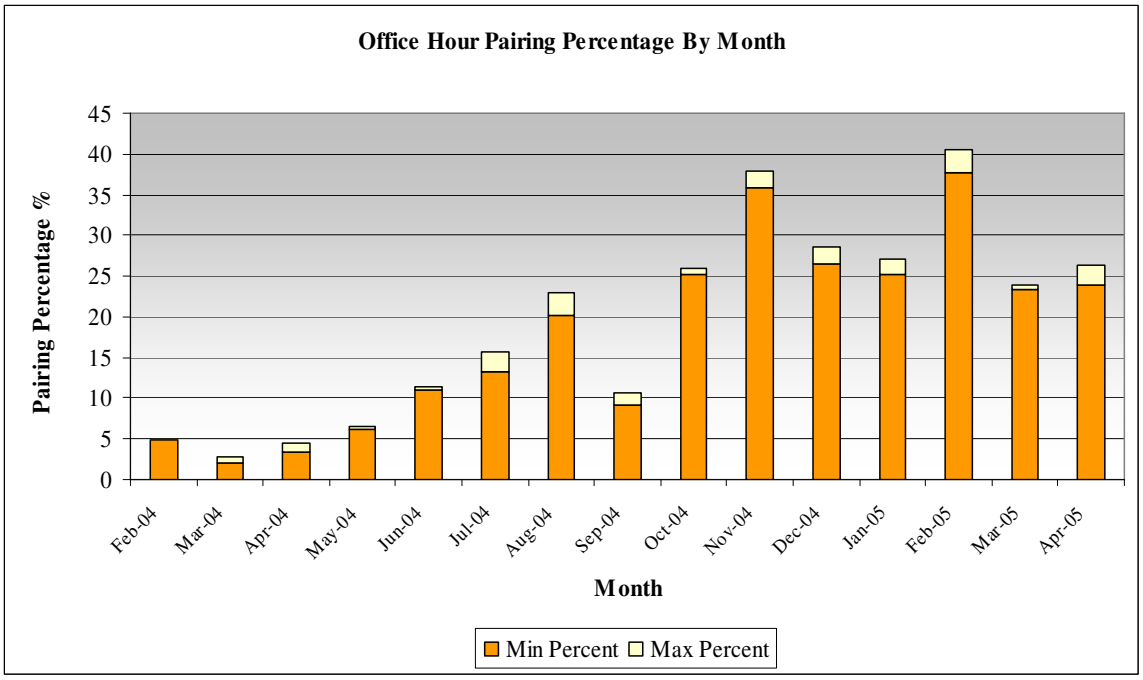


Figure 6-2: Percentage of Pair Pairing by Month

Figure 6-2 shows the pairing percentage of the team per month based on the office hours recorded for that month. The size of the beige bar on top shows the differences between what the pairs record on their pair programming sheets, discussed in Appendix D.

During the three months preceding Scrum and the 1st and 2nd sprints, the pairing time was very low (15% to 20% of sprint and 2% and 11% of office hours). During the 3rd sprint the percentage of time pairing increased to 50%. This increase is also observed in August 2004 Figure 6-2. One possible factor for this change was that Dr. Maurer came to PetroSleuth at the beginning of August 2004 and gave a presentation on agile methods, which included pair programming. After this presentation the developers said that they would recommit to using pairing as part of their daily programming methodology.

In sprint 4, the percentage of time spent pair programming went up to almost 80%. A possible factor for the increase is that most of the sprint was taken up by planning. The development team conducted a couple of tutorials with the customers to show them what kinds of applications could be feasible based on their goals. Due to the

amount of time spent on meetings during the sprint, only a small portion of the sprint was available for development work; however, most of this work was completed using pair programming. This result is shown by sprint four in Figure 6-1 being almost 80% while the monthly pairing percentage for September 2004 shown in Figure 6-2 is only 10%.

In sprints 5 and 6 the amount of pairing decreased. One factor for this is that one of the developers left the company at the end of sprint 5. He impacted the pairing amount in two ways. First, based on my observations he did a considerable amount of the pair programming with other developers. Second, after he left, there were an odd number of developers (3 developers and 1 project manager). In the same month, November 2004, another developer was hired. He did not start to work full time until the middle of January 2005 as he had a significant amount of work to finish from his pervious job. This meant that for sprints 5 and 6 and half of sprint 7 there were only 3 full time developers available to form pairs consistently.

The 8th sprint (February 2005 to March 2005) saw the amount of pairing increase. During this period, two different things were occurring. First, the customers brought the development team a clear set of items that they wanted to have completed by the end of April 2005. After analyzing the requirements, the team identified two independent development streams that could be started giving each pair the ability to work without waiting for work from the other. Second, the developer hired in November 2004 started to work full time.

After the 8th sprint (February 2005 to March 2005) the amount of pairing decreased. There are a couple of factors for this decrease. First, although the developers were assigned tasks as pairs, they appeared to pair program only at the end to review what they were doing rather than pairing the entire backlog item. Second, the new developer seemed to work on his smaller tasks alone rather than with a partner. He would only bring his partner in to review and discuss what he was implementing

The amount of pairing for the last two sprints, 10 and 11, was lower than what had been seen in the past. There are two factors for this decrease. First the developer hired in November 2004 was side-tracked by a different project. During the planning meetings, the group discussed whether or not the task should be a paired task. The task was assigned to an individual as the Windows application needed to be finished and the team did not seem to buy into the idea that pair programming could increase productivity. Second, most of the last two sprints were "clean up" tasks to polish the product before its release. This meant that there was more individual work-related tasks regarding the mechanics of deploying the application.

6.4 Distribution of Work by Task Count

Figure 6-3 shows the distribution of tasks based on how the tasks were assigned to the developers when the tasks were first created. There are two different counts. The count in beige (bar on the left) is the percentage of the sprint that is made up of tasks assigned to individuals. The count in orange (bar on right) is the percentage of the tasks that were assigned to a pair.

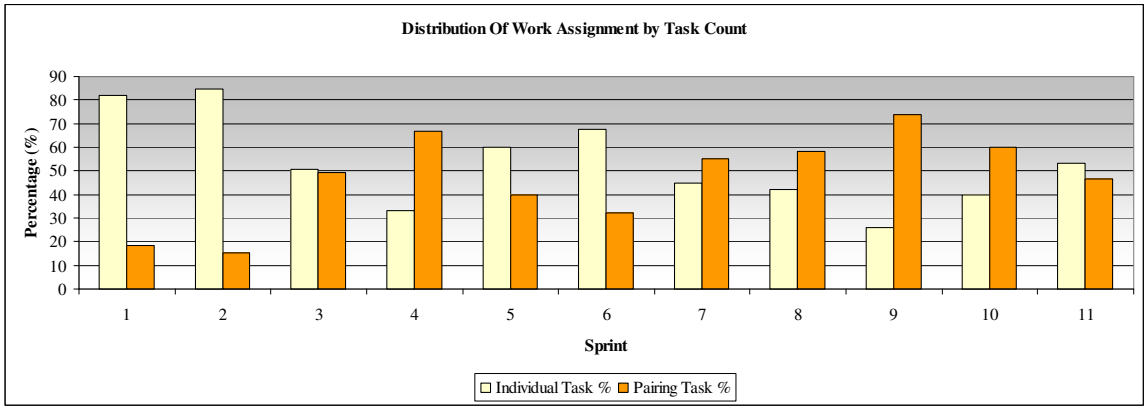


Figure 6-3: Distribution of Work Assignment by Task Count

The first two sprints, 1 and 2, had more individual assigned tasks than pair assigned tasks. A reason for having more individual assigned tasks is tasks were being assigned to individuals and then it was up to the developers to pair program those tasks or not. An exception to this is the summer students who were assigned tasks as pairs. If the

developers decided to pair on individual tasks then that would show up in the pairing percentage graphs as a high pairing percentage (Figures 6-1 and 6-2).

As the sprints progressed, it is evident that the number of paired tasks increased dramatically after Dr. Frank Maurer gave a presentation in August 2004. After the increase in sprint 3 of pair assigned tasks being assigned the percentage remained higher than the first two sprints for the rest of the study. Figure 6-3 also shows the percentage of individual tasks is still sizeable even in later sprints. The reason is that there are still a lot of tasks such as research and small changes, such as change a title, or reformatting a chart, that the developers feel are too small for pair programming.

After creating the distribution of how the tasks were assigned, a correlation coefficient was computed to check for a relationship between the percentage of tasks assigned to pairs and the percentage of pairing done by the developers. Figure 6-4 shows a scatter plot of the pairing percentage by sprint verses the percentage of tasks assigned to pairs and Table 6-1 shows the correlation calculation results.

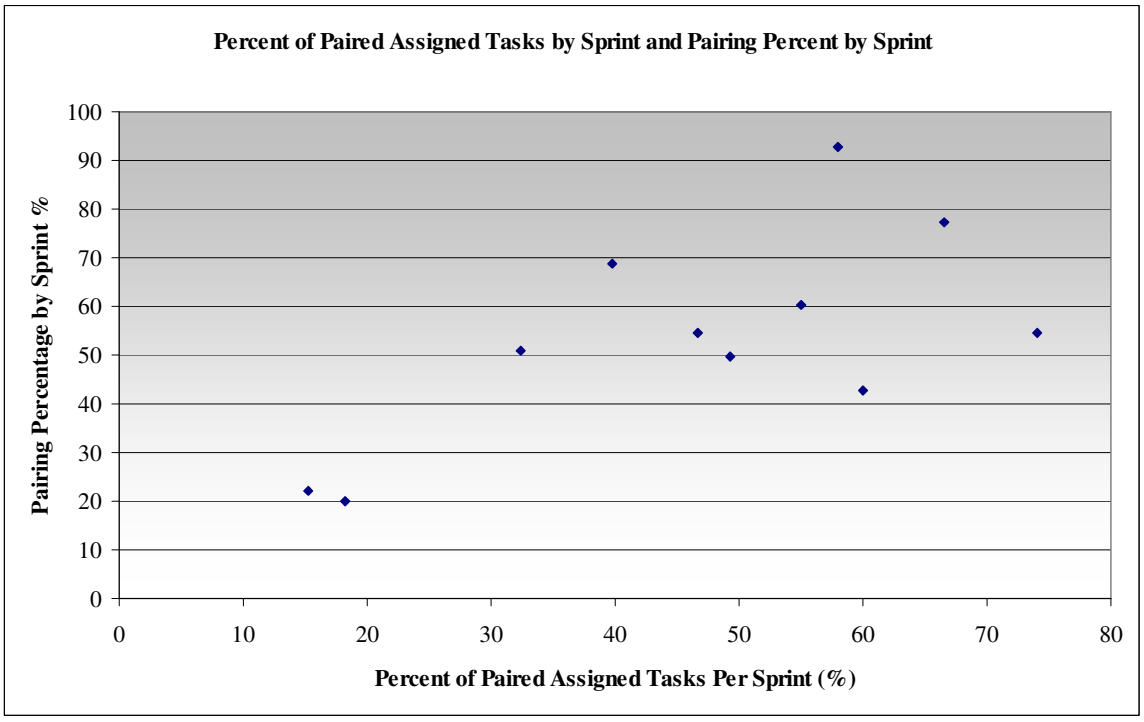


Figure 6-4: Pairing Percentage by Sprint vs % Tasks assigned to Pairs Per Sprint

Table 6-1: Correlation Coefficient: Pairing percent by Sprint and % tasks assigned to pairs.

Correlation Coefficient Without First Two sprints (min)	**Correlation Coefficient Without First Two sprints (max)**
0.196	0.183
P Without first two sprints (min)	**P Without first two sprints (max)**
0.613 (not statistically significant)	0.638 (not statistically significant)
Correlation Coefficient with all sprints (min)	**Correlation Coefficient with all sprints (max)**
0.670	0.669
P with all sprints (min)	**P with all sprints (max)**
0.024 (statistically significant)	0.024(statistically significant)

Table 6-1 has four sections. The top two sections show the correlation between pairing percentage by sprint and percentage of tasks assigned to pairs. Both a minimum and maximum correlation calculated because the pairing percentages have both a minimum and maximum value. The top two correlations are calculated over the last 9 sprints rather than the full eleven sprints. The initial choice to remove the first two sprints was made as the presentation by Dr. Maurer may be considered a separate treatment on the development team and therefore the first two sprints are a different treatment than the remaining nine. I would argue that the first two sprints should be included as there are changes through out the eleven sprints with Dr. Maurer's presentation being one of the changes. I would therefore consider all eleven sprints part of the same treatment and should be included in the statistics. For this reason I have provided results with and without the first two sprints. The results without the first two sprints included show a weak non-statistically significant correlation. However, if the first two sprints are included there is a moderate statistically significant correlation. The results indicate there may be a weak to moderate relationship between percentage of tasks assigned to pairs and pairing percentage.

6.5 Distribution of Work by Work Estimated

Figure 6-5 shows the distribution of how much time was estimated on both tasks assigned to individuals and tasks assigned to pairs. The percentage in beige (bar on the

left) is the percentage of time estimated to complete the individual tasks. The percentage in orange (bar on right) is the percentage of time estimated on tasks that were assigned to a pair.

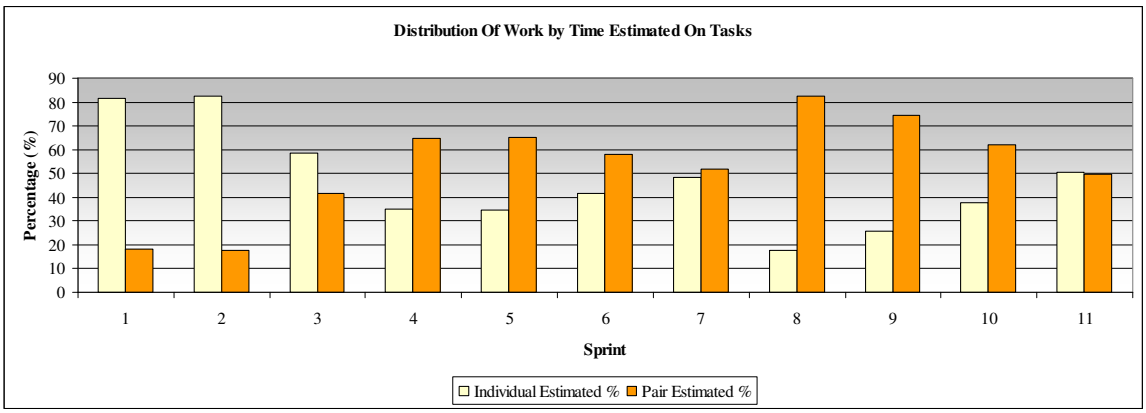


Figure 6-5: Distribution of Work by Time Estimated On Backlog Items

Figure 6-5 shows after the initial percentage increase in the amount of time estimated on paired tasks in August (due to Dr. Maurer's presentation) the percentage stayed high until December, 2004 and January 2005. During this period the percentage of work estimated on individual tasks and pair tasks is about equal. The main reason for this is that during this period, there was a lot more research being performed and this is generally an individual task from the perspective of the developers. In February and March 2005 the percentage of work estimated again strongly favoured the paired tasks, since most of the work was given to pairs in terms of time. In the last two sprints, 10 and 11, the percentage of individual and paired estimates again were similar. During these two sprints there was much more individual work related to product deployment being estimated.

After creating the distribution of how much time was spent estimating individual and pair tasks, a correlation coefficient was computed to check for a relationship between the percentage of work estimated to pair tasks and the percentage of pairing done by the developers. Figure 6-6 shows a scatter plot of pairing percentage vs the percentage of the sprint assigned to pair task and Table 6-2 shows the correlation calculation results.

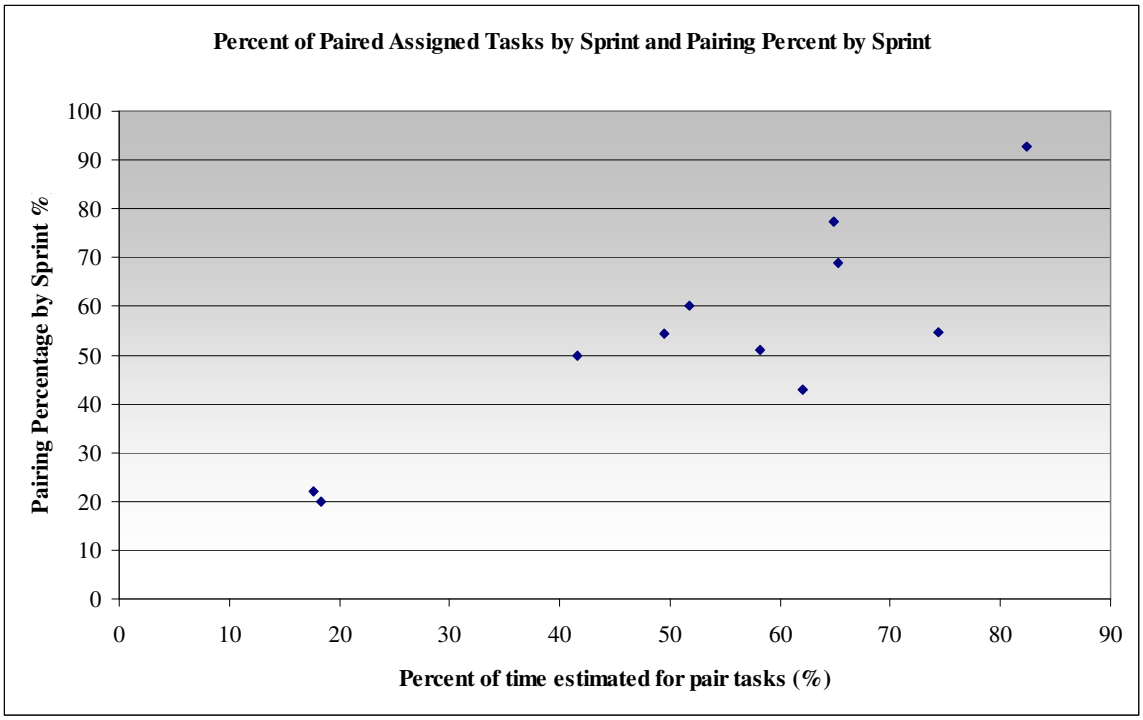


Figure 6-6 Pairing Percentage vs Percent of time estimated for pair tasks

Table 6-2 has four sections. The top two sections show the correlation between pairing percentage by sprint and percentage of tasks assigned to pairs. Both a minimum and maximum correlations were calculated because the pairing percentages have both a minimum and maximum value.

Table 6-2: Correlation Coefficient of Pairing Percentage by Sprint and Distribution of Work by Time Estimated On Backlog Items

Pair Correlation coefficient Without first two sprints (min)	**Pair Correlation coefficient Without first two sprints (max)**
0.620	0.631
P Min Without first two points	**P Max Without first two points**
0.075 (not statistically significant)	0.068 (not statistically significant)
Pair Correlation Coefficient with first two sprints (min)	**Pair Correlation Coefficient with first two sprints (max)**
0.852	0.859
P value with first two sprints (min)	**P with first two sprints (max)**
0.01 (statistically significant)	0.01 (statistically significant)

The top two correlations are calculated over the last 9 sprints rather then the full eleven sprints. The first two sprints were removed from the top two correlation calculation because they were recorded before Dr. Frank Maurer's presented to the team on August 6th and were the only two sprints where the group was not assigned pair tasks.

The results without the first two sprints show a moderate non-statistically significant correlation. However when the first two sprints are included there is a strong statistically significant correlation. The results seem to indicate a moderate to strong relationship between percentage of time estimated for pair tasks and pairing percentage.

6.6 Developer Opinions on Pair Programming

In this section the developer opinions on the use of pair programming at PetroSleuth throughout the course of the study are presented.

6.6.1 Developer Thoughts before Pairing before Pair Programming was Introduced

The pre-pair programming questionnaire (Q1) given at the end of January 2004, ***before*** pair programming was introduced, asked "What is your opinion on working with a partner while doing daily programming tasks?" and "If you had a choice to program with a partner or not during your daily programming activities would you choose to program with a partner or not, why?". The developers mentioned that they would want to work with a partner if the task they are working on was difficult and they require the assistance of another developer to finish it. They also would like to work with a partner to pass information between each other. One of the main concerns raised by a number of developers was that the developers do not want to work with someone all day long, and at times, want to work at their own pace.

"*I think for learning and being familiar with more of the code it would be good. My only concern would be working with someone for the entire day. Sometimes it's just nice to work alone and work at your own pace (faster or slower)"*

The same developer also noted:

"Currently if we have any difficulties with a module we may bring in another person to sit with someone to help figure it out. I really like the idea of working in pairs on an as needed basis. So, I would like to have the ability to choose when working in a pair will work."

When answering the question about whether or not they would want to work with a partner during their daily tasks, the developer had this to say:

"*Sometimes it is harder to work with someone else when there is a particularly intense or difficult area to program. The other person is distracting. Alternatively, it is good to have the other person there when you're not sure how to approach a particular problem. You can share and discuss ideas. Another person can also be very helpful in debugging, as they see things you do not, and have different ideas about the problem***."**

6.6.2 Developer Likes and Dislikes about using Pair Programming

The developers were asked what they liked and disliked about pair programming after using the practice.

After Four Months

The developers, in general, liked working with another developer to brainstorm and solve problems. Three out of the four said that they got a lot of knowledge from working with other developers and they also liked the ability to brainstorm with another partner. The fourth developer said that with a pair you may not be stuck for as long on a bad idea as you may have done as an individual.

"I liked the fact that I can learn much more during the paired time than just trying to figure out something by myself…"

"*I like it in a brainstorming phase, when you are working out why something does not work or how to approach it…"*

"*I think pair programming forces the pairs to think more, and you end up not being stuck on bad ideas as long as doing it individually, so you get to a better solution quicker I think, or at least you rule out the bad ones quicker*"

Developers disliked pair programming when their partners go too fast and did not slow down. Another developer did not like the idea of spending a lot of time pair programming with certain other people; they called it being "stuck" with another person. One developer said that when pairing they could get distracted and they loose track of what they were thinking of at the time. The developer stated that when most of the program is still in one's head and the pair partner is a distraction, this causes one to loose their train of thought. A fourth developer said that there are some tasks, such as things that can be done quickly, that should be done by an individual rather then a pair.

"..I don't like the idea of being 'stuck' with a person all day. Sometimes you just need to work alone. The paired programming style we attempted to adopt was very good in the fact that it gave us flexibility around when to pair."

"I guess even though we did not do this, sometimes I don't think pair programming is needed. Just when you have certain things you can do yourself relatively quickly that should be elementary to everyone else. There are times you don't want to work with someone and have someone over your shoulder for the whole 7 hours"

"Sometimes if the other person is going too fast, if they know what they are doing right away and you don't get it yet and you miss half it…"

After Fifteen Months

The developers said that they liked how pair programming found errors, that problems could be solved faster, and they had someone to discuss problems with.

"*Liked having someone to discuss with when you're stuck on a problem…Like getting to understand other developer's style and point of view on things, as well as learning the parts of the code that they know*"

"*I liked that a lot of errors could be caught by the partner and quite a few problems can be solved faster since you talk it out with someone instead of sitting by yourself and trying to solve it*"

The developers said that they did not like pairing all the time as it is too intense and they would also like some personal time.

"*Pairing all day long is sometimes not a good idea since we occupy the same office and some people have different schedules, plans, phone calls etc*".

6.6.3 Developer Viewpoint on How Much Pair Programming was Done

The developers were asked how much time they thought they had spent pair programming. One developer said 20%, another said 10% and one said 5%. The project manager said that he had not used pair programming. Another developer who had joined in April 2004 after pair programming was introduced, had this to say about the amount of time he thought he had spent pair programming: *"…during the first month it was probably 50% but then probably 10% after that and following that even less*". The summer students said that they had paired programmed 50% of the time, one of the reasons given by the summer students is that they had been forced to pair program because they had been given the same tasks as each other.

Along with asking the developers how much time they spent pair programming they were also asked how formally (closely) they followed pair programming. All of the developers said that they use a more informal approach to pair programming

The developers were also asked how they chose when to pair program and how they chose with whom to pair program. The summer students said they did not choose with whom to pair as they were assigned the same tasks as each other and therefore had to pair program with each other. The other developers said, when choosing a pair partner the other person's knowledge of the code they were going to implement was one reason to pair with them. Another criterion used to determine whom to pair with was if someone needed training on a certain part of the system, then they would pair with that person to bring them up to speed. Two developer responses mentioned that their main criteria on, when choosing a pair programming partner was that the two developers should have similar tasks, as this would make it easier to justify the pairing. Other developers

mentioned that when developers were free to pair played a factor in determining who they would pair with.

"Mostly it was the person was free or they had a similar task, there was no real pairing procedure. For example, if programmer A was working on wells and I was working on wells we would probably work on it together"

"I think for me personally you end up pairing with whoever is the natural person who is doing the work, I think pairing here is along the lines of those who are doing the work, whether it is the architectural standard and that information has to be imparted down to the rest of the team so there is natural pairings now of experienced to less experience that takes place"

The developers were asked how they chose when to pair program. The summer students said that they would pair until they figured out what needed to be done and then they would break up, work a bit, and then review the work with each other. One of the full time developers said that they would pair if the task was complicated or it was related to one of their other tasks. Another developer said that whenever they had a question they would find someone to answer it and that would become pair programming.

"I like to use it during design time even if someone is not related to the tasks just to talk the task through, kind of use them as a sounding board. Other than that it would be just when necessary"

"Whenever I had a question, that is when I asked someone for help, that becomes pair programming"

For the most part, the developers seemed to pair program when they thought that a second person would be helpful, for some, this meant that they would go and find a partner to help them design a function while for others it meant that they would not seek out a pair unless they had a question.

6.6.4 Developer Views on Obstacles to Pair Programming

The developers were asked if there were any obstacles preventing them from pair programming.

After Four Months

One developer said that there were no obstacles while another said that it was "*Initially Management*". Other responses included "*Not having enough time to do any programming",* and *"no clear guidelines on when to pair, heavy amount of work load assigned, work load is now also split up so that work is pretty independent from each other*".

The developers were also asked if there were any other factors that made it difficult or easy to pair program. The answers to this question were very similar from each of the developers. One developer and the project manager commented that pressure to deliver and the timelines that everyone was working on were one factor preventing more use of pair programming. Three of the programmers said that the pairs familiarity with the area and knowledge base of the other people were factors in how much they paired. Also, rapport with the other developers was mentioned as a reason to pair. One developer noted that they work on "a different clock" than the other developers, therefore their schedules may not synchronize well with other team members.

"Related modules, who the pair was and how busy the other pair and yourself was. rapport with the other developer"

"*Mostly our timeline, I would also say it does not seem to be a priority from the management … I would rather have the time to sit and experiment with it to see where it would be useful for us. Well one point in time the manager would come in and say we are ready for pair programming but then when they assign tasks for the next sprint and there is not enough time to try it out especially when it is due the next day*"

"…I think I have a different clock than other people who I work with so I suppose if you are doing a lot of pair programming then it would be difficult and I mean by that I like to come in early and get my work done early so by the time mid afternoon comes around I am ready to fade out and they like to work later"

As mentioned before, the summer students who seemed to do a lot of pair programming said that one of the factors that influenced the amount of pair programming

they did was the fact that they were given the same tasks and essentially forced to pair together.

After Fifteen Months

The developers said the obstacles to pair programming were: the time it took to learn how to pair program, the differing work hours, and size of the team. One developer said,

"*The only real obstacle was the adjustment period of developing in pairs. All the developers on the team have never been in this type of process before with paired programming before*"

Another developer commented:

"*Yes, Differences in core hours worked by team members. Uneven numbers of developers available for work*".

While a third developer said it can be difficult if the developers have different programming styles.

6.6.5 Developers' Views on Pair Programming and the Effect on Software Quality

The developers were asked if they had noticed any changes in software quality, testability, readability, or maintainability since pair programming had been in use. It was left up to the developer's natural interpretation of what quality, testability, readability, or maintainability mean.

After Four Months

The responses were generally positive:

"*Yes I do, quality, readability, and maintainability is affected in a positive way. Two programmers have now taken a look at the code. Therefore code does not rely on one programmer, two people have seen it. Pairing eliminates small mistakes from code, or even potentially large ones. I am unsure about testability*"

Other responses were more cautious however and they said that they try to make code with the above attributes anyway:

"*I think that it CAN. But, being snobbish here, I'd like to think I invest effort in making code this way anyway. Maybe I'm wrong. I think that it makes you more aware of the need for anyone to be able to follow your code, understand and maintain it easily*"

Two developers said that there may have been some quality increase due to the fact that little errors should be caught more often, while another two developers said they did not think there was any change. One developer said that they had not been at the company long enough to comment on any effect on quality.

"yea definitely, there was, even the simple errors, these are the errors that you shouldn't make but you do because you know you are not supposed to do it, and your mind is somewhere else or you are thinking of a different variable, or something weir., Those are the things that are sometimes the hardest thing to track down and the easiest thing to catch when someone is there with you"

"I wouldn't say that it did. And I think it is difficult to answer that because in my thinking there was not a lot of pair programming", another developer added *"I would say, verdict is still out, still waiting for results"*

After Fifteen Months

The responses indicated that they thought pair programming did increase software quality. One developer noted that they can find bugs faster and it takes less time to understand them when pair programming is used. Another developer mentioned that two people may make a better design/algorithm, which could be considered better quality. One developer however noted that pair programming did not help with the team conforming to coding standards, meaning good programming practices.

"*Yes, it probably did have some effect in terms of quality since code is reviewed/designed by two people and two people can sometimes come up with a better design/algorithm***"**

"*Yes, I find that now when bugs do come up, I can find them in less time and understand the problem faster with the other developer there*"

"*The interesting thing from having the one little code review meeting is that even with pair programming, the pairs have not been as diligent in test driven or doing good practices in their development*"

6.6.6 Developers' Views on Pair Programming Effect on Knowledge Transfer

The developers also were asked about the effects pair programming had on the transfer of information between developers.

After Four Months

The answer from all of the regular developers was "yes" pair programming had an effect on knowledge transfer though some expressed uncertainty about their answer. Of the summer students one said it helped while the other just said "no".

"I think it definitely has an effect, but same as above we needed to more fully embrace the philosophy to see better results"

Another developer had this to say about knowledge transfer and learning with pair programming:

"To a limited degree. Sometimes you think that a person understood or 'got it', but evidence after the fact proves otherwise…In some cases, it is a great help in getting someone oriented to an environment/application when they are new to it. They get going faster; the flip side being that the other person is slowed down during this process. (Inevitable, I think, no matter how you train someone.)"

"I am doing some paired programming right now to pass around some information about the new architecture that we have designed. Without it, it would be very difficult for someone to come in and try and emulate what we have done even with an architectural skeleton without sitting down and walking through why we did it the way we did it. So for knowledge transfer and growth of employees I think it is very useful"

"I think it was very very useful for me at least. I am not sure about the other person but it is definitely much easier to get information from another person during pair programming than searching the internet or looking up documentation that is not really available, so it was very useful"

After Fifteen Months

All the developers said that they thought pair programming did help with knowledge transfer. Some of the developers commented on how it helped bring people's

skills forward and fostered better communication, while others said that the knowledge transfer did not always happen due to the pair partner not following along.

"*Yes, but not always, I found that sometimes partners tend not to follow what you're doing and code looks very new to them when they get to work with it*"

"*Yes, working with others helps to share skill and improves each other coding styles. It allows coders to become better at explaining their code and expands their understanding of design and how best to communicate it*"

6.6.7 Developer Views on Pair Programming Effect on Speed of Development

After Four Months

The developers were asked to comment on the amount of time it took to develop software using pair programming and not using pair programming. Two of the developers commented that pairing might have slowed them down because the other partner could have been off doing something else or they could have got the task done faster as an individual. While one developer said it speeds up what they are doing when they use pair programming.

"*I think that we tend to use it sporadically, when we are using it, it definitely speeds up what we are doing… So I would say it would speed it up when I use it and no I don't use it because it would slow me down if I can do it on my own*"

"when I am paired with someone in certain phases in programming, it is difficult for me to complete a thought and get to the end point so what I was doing took much longer and my frustration level was much higher"

After Fifteen Months

Although the question was not asked in the wrap up interview, from observations at the company and discussions with the developers, I can say that this opinion did not change. Some developers thought it sped up development while others thought pairs were still distracting and slowed them down. The developers told me that from their

perspective pair programming did not make the coding any faster but it may have helped in designing the task better.

6.6.8 Developer Comments on Continued use of Pair Programming

The developers were asked if they would like to continue to use pair programming In the future.

After Four Months

All of the developers responded that they would like to continue to use pair programming. One developer said they would like to continue to use pair programming in a limited fashion while another developer said

"*Yes, I would like to continue with pair programming, gives me greater confidence and understanding when I have to go fix bugs or change code if the original developer is not at work for one reason or another*"

As part of their responses some of the developers discussed some reasons why they like to use it to a limited degree. One developer said "*sometimes I find it difficult to work with other developers*" while another developer said that they sometimes find pair programming frustrating especially if the programming partner points out little problems often.

"*…I find that I need to work alone sometimes when the thinking is particularly heavy and complex and I have a limited tolerance of working in pairs: I find it frustrating at times. Feels like being on a leash. However, there are times that I find it helpful: usually this is when I am fishing for ideas and I'm stuck on how to move forward with something. I'll admit too, that I find it boring to a degree when the other person is working at the PC. It's also irritating sometimes when the other person is following too close, in terms of telling you about typing mistakes as you make them. Obviously there's some learning of etiquette and personality traits that happens as you get used to working with someone*"

A secondary question to using pair programming in the future was whether they would recommend pair programming to other teams. All except one developer said that they would recommend pair programming to other software development teams.

"*Yes, if the development team is small. Small teams usually have to multitask in responsibilities therefore it would be better for developers on that team to have more than one person see particularly difficult sections of code, for maintenance and upgradeability of code*"

One developer said they would not recommend pair programming to another development team because its value had not been established yet at the company.

After Fifteen Months

All of the developers said yes they would like to continue to use pair programming. One developer mentioned though that they would like to continue using it but would like more personal tasks to do on their own.

"*Yes, but not 24/7 as we do right now…there should be individual programming tasks just to get us a little relaxed*"

When asked would they recommend pair programming to other development teams. All of the developers said yes. One developer pointed out that they would recommend that other teams have someone experienced with pair programming be part of the team full time to assist with the transition to pair programming.

"*I would because this practice seems to work well in terms of knowledge transfer and code is more error free than it could have been otherwise*"

6.6.9 Recommendations for Improving Pair Programming

The developers were asked what could be done to improve pair programming and its use at the company

After Four Months

At four months the developers were also asked; would increasing the amount of pair programming be desirable. One developer said, "*It is very important to have full managerial buy- in to make trying pair programming successful*" while another asked that there be more ground rules: *"Set some ground rules between the pair or within the*

group before you start. Just as we do for meetings". While another developer said "*Developers need to start planning and thinking on how to work in pairs and then working in pairs*"

For the second part of the question all of the developers including the summer students said that increasing the amount of pair programming would be a desirable goal. The two summer students said that to increase the amount of pair programming similar tasks needed to be assigned to the developers. Other individuals wanted more planning in how the pairing work was split up, while one developer said that the team had to recommit to using pair programming. One developer also mentioned that having units that have to be done by two people would help.

"I think it would be and I do think it will be a little but more costly in the short term. Right now I hope we move toward doing more pair programming but I don't know. I think as a team we have to recommit to it, because we are committed to it but we did commit to it in the wrong time, we were taxed heavily with time and now we are still busy but less busy so I think now would be a good time to recommit to it"

"…maybe some of the tasks would be needed to be set right away as pair programming tasks instead of trying to do it yourself first then try to get someone else to pair program with you then maybe that would be more effective"

After Fifteen Months

After fifteen months the developers were asked what could be done to improve pair programming but unlike before were not asked if this was a desirable goal. To Improve pair programming, one developer wanted to have someone in the team who was experienced with pair programming while others wanted more individual projects and more experienced team members to continue their skill growth.

"*Yes, we shouldn't pair all the time, there should be personal tasks (other than research)"*

6.6.10 Wrap Up Interview after Fifteen Months

As part of the final wrap-up interview the developers were asked what their thoughts were about pair programming. This question was to get a final set of comments

in their own words. All of the developers found that pair programming was a positive experience and they liked the knowledge transfer and the ability to bounce ideas off each other. They did mention that pair programming can be draining at times.

"*It is a very interesting process to learn how to do it, I still have mixed feeling about it. I like it better then I did in the beginning, I think it is difficult to get to know someone that well sitting beside them day after day, I think it comes down to getting to know the person you are working with and until you go through that process pair programming is hard. However, you stick with it and get to know each other and how each other works and you build a level of trust that you don't get if you don't do something like that. I do find it difficult. It can be very intense when you are in an office with somebody for that many hours a day and I can find it a little bit to much to do it all day so we try and break it up a little bit and then come back, but it is a bit much overexposure to somebody. Overall as a process, and the results we get with it, and the cross training and bringing everyone up to a similar level, it's really good and I am in favour of continuing with that kind of a technique*"

The developers were also asked how they dealt with the situation where they have a paired task but all of the other developers are busy. The developers said that they would start the task by themselves and then review it with a partner when their partner got free.

"*What we tend to do, is that if you have nothing to do then you go ahead and start the task but it is expected that you will review the all of that code with the person who is your pair before you check it in, that includes tests and the whole things, if you have to work alone you go ahead and do it because we don't want to waste time but you are expect before the code goes back In the repository that you have to review it so that they understand it and everything is good*"

6.7 Discussion of Pair Programming Questions

How often will developers use pair programming?

During the first four months the developers did not use pair programming often. The developers thought they used pair programming between 5% and 20% of the time. They characterized their use of pair programming as informal. Based on the results of the pair programming sheets, the developers used pair programming between 5 to 10% of office hours and 10 to 20% of sprint hours. Initially, I observed that the developers did not seem to actively try to pair program entire tasks, instead they used pair programming when they encountered a problem and needed assistance to solve it. This pattern changed

over the next eleven months. As the study progressed I observed developers using pair programming more often. This increase is reflected in Figures 6-1 and 6-2, with pairing now ranging from 10 to 40% of office hours and 40 to 90% of sprint hours. Along with the increased use of pair programming, I observed the developers were using pair programming for entire tasks. When the developers were asked after fifteen months of use, how often they used pair programming, they said they now use it all the time. This was a significant change from using pair programming for just debugging or for reviewing their source code with someone. Even though the frequency of pairing increased from the beginning of the study, there were still periods where the amount of pair programming fluctuated. Some of the possible factors that contributed to this are discussed next.

What could be some possible influencing factors to pair programming use?

The possible influencing factors to the use of pair programming provided by the developers included; not enough time initially to pair program due to lack of management support and periods of uneven team sizes. These factors contributed to the varying amounts of time spent pair programming. Based on what I observed, in the beginning of the study, I think the largest obstacle to pair programming was the time pressures on the developers due to management not giving them time to get acquainted with working with a partner. Throughout the first four months, the developers often indicated that they would like to increase the amount of pair programming, but in their opinion they did not feel they had time to pair more often due to the amount of work assigned. To try to get more management support for the transition into pair programming, Dr. Frank Maurer came to the company and gave a presentation on agile methods, including pair programming. The use of pair programming at the company increased after this presentation, from 15% to 22% of office hours and from 20% to 50% of sprint hours. Two changes occurred after the presentation: first management and the developers recommitted to using pair programming and second, the team started to assign tasks to pairs rather than individuals.

After Dr. Maurer's presentation at PetroSleuth, more tasks were being assigned to pairs rather than individuals. To check for a relationship between the percentage of tasks assigned to pairs and the pairing percentage, a correlation coefficient was calculated. The result was a weak non-significant correlation between the two (Table 6-1). This indicated there was not a significant relationship between the number of tasks assigned to pairs and the amount of pairing done by the team.

When the correlation coefficient between the percentage of the sprint estimated as pair hours and pairing percentage per sprint was calculated, a moderate non-significant or a strong significant relationship was found (Table 6-2) depending on the inclusion or exclusion of the first two sprints. This calculation indicated there was a relationship between the percentage of the sprint hours that were estimated to pair tasks and the percentage of pairing the developers accomplished. This result can be formulated as a hypothesis to be tested in future work: The more hours estimated and assigned to pairs, the more the developers will pair program.

What do developers think about pair programming?

Before starting pair programming, the developers thought pair programming may be useful for learning, debugging, and "figuring out" solutions. The developers also expressed concern about having someone else working with them all day. This concern was common throughout the team. They wanted to try pair programming but the prospect of working with someone for the entire day did not appeal to them.

After using pair programming for four months, the developers thought that pair programming was helpful for learning and commented they felt they learned more while paired. They discovered they spent less time figuring out solutions as a pair versus working alone. The developers liked brainstorming ideas with their partner and found they were not getting stuck on a bad idea as long.

The developers did not like it when their pair would go too fast, too slow or they had to work with someone all day long. This problem addressed the personal

programming styles of each developer. Since each person had their own ways of working (faster or slower) and approaching problems, these approaches could conflict when they were in a pair. This period of conflict and integration of programming styles is usually referred to as jelling (Williams et al 2000b). Since the developers had not used pair programming often in the first few months, they had not had time to "jell". After fifteen months the team as a whole still had not jelled, as certain pairings still had issues, such as being distracted by their partner, speed differences in writing code and problem solving. Another reason for this lack of jelling was the loss of one developer and the subsequent addition of a new member in November 2004 requiring the team and the new developer to start the jelling process over again. After four months, the developers commented they found working with another developer all day, very difficult. They held the same view after fifteen months. The developers said they felt pair programming was too intense and they wanted some more personal time instead of working with the same person all day.

The developers had many views on pair programming and quality. After using pair programming for four months the developers noticed a positive effect on quality. The developers mentioned pair programming seemed to catch the smaller errors, such as incorrect variables. After fifteen months of pair programming, most of the developers still thought pair programming increased quality, but one developer thought that even with pair programming, the developers were not following good coding practices. This comment is similar to what Hulkko and Abrahamsson (2005) stated in regards to pair programming and coding standards. They found that pair programming did not increase the conformance to coding standards over solo programmers. One possible reason could be that neither of the pair partners was checking their code for coding standard conformance. The validity of this observation and possible factors will have to be left for future work.

One developer thought that when they used pair programming it sped up the development of software. The developer however, did not want to use pair programming for simple tasks as they thought they could do those tasks faster. Another developer mentioned that when they used pair programming they found it distracting, thereby

slowing the speed of completing the task. This slow down, due to a distracting partner could be seen as part of the jelling process, but based on my observation this distraction for some pairs did not decrease over time. After fifteen months of pair programming some developers told me that they did not believe that pair programming increased the speed of software development, while others said it had sped up what they were working on. The overall result is that there is no firm consensus as to whether or not pair programming increased the speed of software development. The lack of consensus in these results, points to the need for more pair programming studies in industry.

The developers thought pair programming had assisted with transferring knowledge between each other. After the first four months of using pair programming the developers thought pair programming did have an effect on transferring knowledge between them. Based on my observations this was especially true for new developers. In April 2004, a new developer was added to the company and to get them started they paired with the other developers. They thought that pair programming was very useful to them as it got them up to speed with how the other developers worked quickly. After fifteen months, the developers still thought that pair programming was useful at transferring knowledge between the developers and that is helped developers increase their skills and communicate better.

What kinds of purposes do developers use pair programming for?

In the beginning, the developers used paired programming for pair help and debugging rather than pair programming the entire task from start to finish. When tasks were assigned to an individual, the developers only did pairing when they encountered a problem.

After tasks started to be assigned to pairs however, the developers used pair programming for the entire lifecycle of the backlog items and tasks. The developers then decided research tasks would be assigned to individuals. Research tasks are where the developer has to research a new product or technique and evaluate it for use at the company. The reason research tasks were given to individuals is that research usually

requires significant reading and experimentation and having two people do this type of work was seen as wasteful. Other than research tasks, all other tasks were assigned to pairs.

What recommendations do the developers have to make the pair programming experience more enjoyable?

The developers made four main recommendations:

1. Give developers more individual tasks that are not related to research, to break up the intensity of pair programming.
2. Paired developers should be assigned similar tasks.
3. Developers should create some base rules as to what should be pair programmed and what should not be pair programmed.
4. There should be someone who is experienced in using pair programming at the company full time to assist with the transition.

The first developer recommendation came from the developers' comments after using pair programming for fifteen months. At this point in time, the developers thought they were pairing too many tasks and wanted some individual time to break up the intensity of pair programming. I had asked if they had thought of taking breaks. The developers said they had taken breaks but they actually wanted to have meaningful tasks that could be accomplished individually rather than with a pair partner.

The second recommendation came when the developers were asked how pair programming could be increased. The developers mentioned they would pair if they were assigned similar tasks as another developer. The reasoning is, if the tasks are the same or similar, than it would "make sense" to work with the other developer because they would be accomplishing the same goal.

The third recommendation came about after four months of pair programming. Initially the only rule was to pair program production code. What the developers wanted were rules as to what should be paired and what should not be paired. This

recommendation was implemented by saying research is the only task to be done individually, everything else is paired.

For the fourth recommendation the team indicated it would be nice to have someone full time at the company who was an expert in pair programming. This individual could help in transitioning the team to using pair programming and could coach them in using the process.

CHAPTER SEVEN: TEST-DRIVEN DEVELOPMENT AND CONTINUOUS INTEGRATION

In this chapter, I present both qualitative and quantitative results based on the experiences gained by introducing test-driven development (TDD) and continuous integration (CI) into PetroSleuth is presented. The following questions will be addressed in this chapter:

- **How often will developers use TDD and CI?**
- **What are some possible factors influencing the use of TDD and CI?**
- **What do developers think of using TDD and CI?**
- **Does the use of TDD improve the defect density of the software produced?**

Chapter 7 is structured in the following manner: In section 7.1, I present how test-driven development and continuous integration were introduced into PetroSleuth. In section 7.2, I discuss some initial observations made while the developers started to use test-driven development and continuous integration. In section 7.3, the quantitative results are presented. In section 7.4, the qualitative results from developer questionnaires and interviews are discussed and in Section 7.5 I discuss the above questions.

7.1 Introducing Test-Driven Development and Continuous Integration

On June 3rd 2004, I gave a tutorial and overview on test-driven development and continuous integration. The power point slides used for the presentation are included with this thesis, in Appendix A. During the presentation, I went over what test-driven development and continuous integration were about. The presentation took the developers through the tools they would be using to do test-driven development, which in

this case were Nunit (Nunit 2005) for unit testing and Nant (Nant 2005) to run automated builds. I then implemented some example code using a test driven approach and discussed how continuous integration would assist using test-driven development. The remainder of the presentation was spent answering questions about test-driven development and continuous integration.

7.2 Getting Started

The use of test-driven development started off slowly. After the presentation, the developers started making small applications by themselves related to work they were currently doing. A few days after the test-driven development presentation, a developer asked for some assistance in getting started using test-driven development. He had partially implemented the functionality he was developing, but decided that he wanted to add unit tests and try the rest using test-driven development. After creating his first test, the developer ran the test and that test failed. He thought his program was working correctly as he had verified it manually before. As a result of the failed test, the developer went to the line in his code (as output by the unit testing program Nunit) and stated, "*Wow this unit testing is cool, I would have never found this problem out*". The problem the unit test found was due to the function not handling certain inputs. The developer made a change to the code and the test passed. The developer continued to create the rest of his functionality but this time he wrote his unit tests first. Even though the problem was discovered by a unit test written after the code was implemented, the ability of unit tests to catch problems provided the developer with an incentive to start to write unit tests before writing the code to prevent such errors from happening.

As the study progressed, I noticed the developers were not always writing tests first. I would ask the developers how they were using test-driven development they would state that sometimes they forgot to write tests before implementing the functionality. Based on what I was seeing, the developers were **not** using test-driven development instead they were just unit testing code.

7.3 Quantitative Results

In this Section, I present quantitative results relating to the amount of test code that was generated. In addition, the defect numbers such as number of user found defects and defect density is discussed.

7.3.1 Amount of Test Code

Here I present how much unit test code is contained within only the third windows application because the first two windows applications and the website do not have any unit tests associated with them. Figures 7-1 and 7-2 show the amount of test code written from two different perspectives. Figure 7-1 shows the percentage of the source code for Windows Application 3, that is unit test code in terms of C# lines of code and statements. Code and statements are defined in Appendix C. From Figure 7-1, you can see the percentage of test code started at approximately 15% and then increased to 30%. The final percentage is still less then the expected amount for test-driven development. In test-driven development, it is expected that there should be about as much test code as production code, although this is just a rule of thumb. If this were the case, the percentage of system code that is unit test code would be approximately 50%.

There are various factors to be considered when looking at these results. First, since September, much of the unit test code is auto generated, which means the amount of test code should grow automatically as new business objects are created. When the team auto generates a skeleton for the business objects, a skeleton for the tests is also generated. The test skeleton already has some basic test checks, such as, checking if the object validates itself properly. After the team auto generates the test, they manually go into the source code and set some expected values for the auto generated tests. At the same time they add their own unit tests for functionality. Secondly, since there is a GUI component, and the code is not unit tested, there will be a decrease in the percentage of the application that is test code. GUI code is not unit tested as there are currently no easy to use tools for unit testing GUI components.

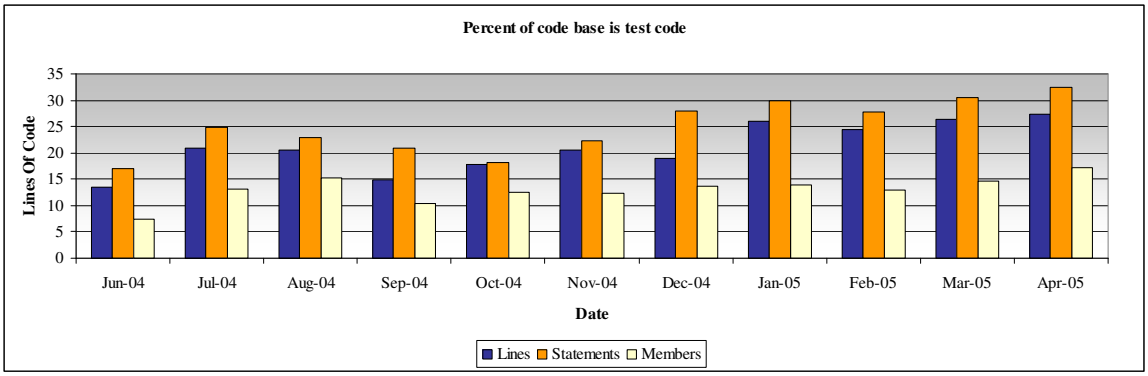


Figure 7-1: Percentage of Overall System Code that is Unit Test Code

Figure 7-2 provides another perspective as to how much unit test code was in the system. This chart shows the number of assertions in the application. An assertion was counted if the word "assert." or "Assert." appeared on a line of code and was not part of a comment. An assertion is a comparison of an expected with a resulting value.

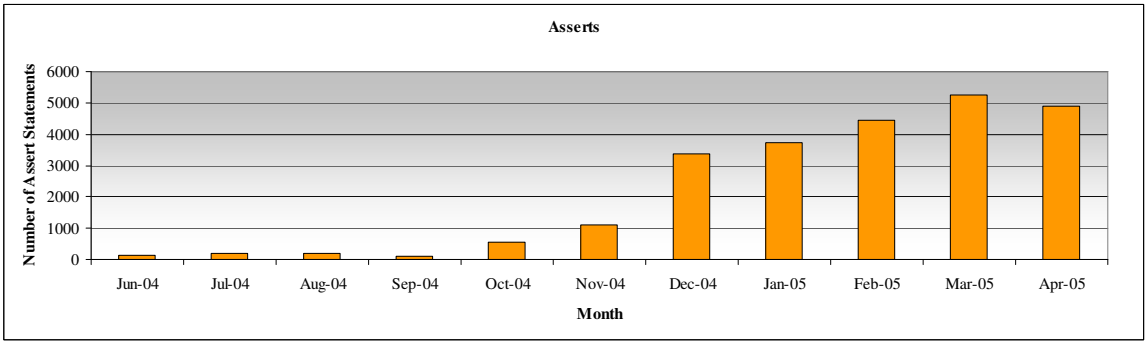


Figure 7-2: Number of Assert Statements

Figure 7-2 shows initially there were very few assertions in the system, but over time, the number of assertions increased quite quickly. This fast increase was during a period when the development team was doing a fair amount of automatic code generation. It would be interesting to determine how much code was auto generated and how much was manually created, but due to the developers making modifications and mixing auto generated and manually generated code this was not possible. An increase in assertions represents an increase in the number of comparisons (tests) between an expected and an actual value.

7.3.2 Test Code Coverage

In addition to looking at how much test code is part of the system, we need to consider how much of the code is covered by executing the test code. The following Figure, 7-3, shows how the amount of test code coverage changes over time. Test code coverage is the percentage of code executed when the unit tests are run. In Figure 7-3 the first bar on the left shows the test code coverage for assemblies that have unit tests written for them. The second bar on the right shows the test code coverage for the entire system including.

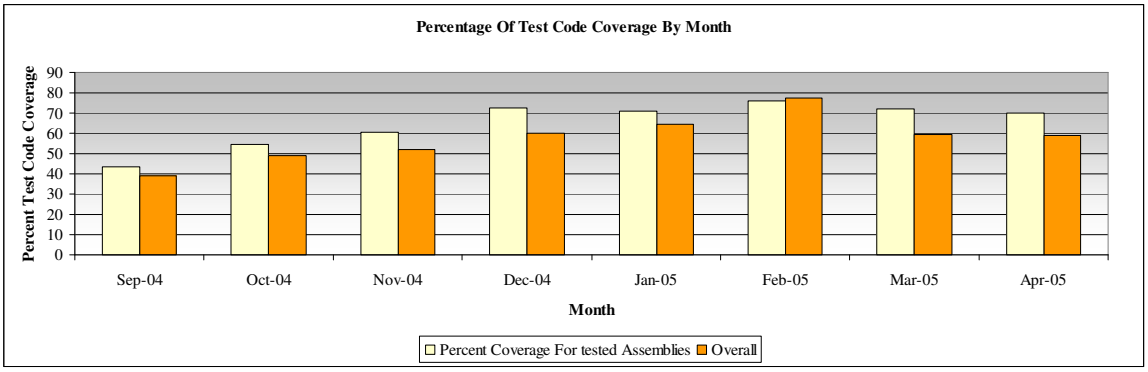


Figure 7-3: Test Code Coverage Percentage by Month

The data displayed in Figure 7-3 starts on September 2004 versus June 2004. The reason for the time difference in data collection, is that prior to September 2004 the unit tests required access to a development database on a centralized server. During September 2004, this server had its databases deleted and restructured. The deleting of the database caused unit tests written before September 2004 to become non-functional. From Figure 7-3, you can see the test code coverage starts at 40%, due to the fact that the first three months of data is not available. The test code coverage seems to stabilize between 60% and 70% over time. There are three possible reasons for this:

1. First is that most of the test code is generated when the business objects are generated. Therefore, the level of test code coverage will vary with the amount of business objects created.

2. Second, there are methods that are inherited as part of the business objects that are not tested in unit tests. Therefore, there is code in every object that will contribute to a decrease in the test code coverage amount.
3. Third, the GUI is not unit tested but it is included in the overall test code coverage percentage. GUI code makes up 5% of the system.

7.3.3 Defect Numbers

Figure 7-4 shows the number of "user found" defects reported in each of the three main software projects created. From Figure 7-4, there were a large number of errors reported for the first windows application and fewer reported for the website and the third windows application. One possible explanation for this difference in errors is Windows Application 1 and 2 had been in used by the customer for a longer period of time than the website or the third windows application

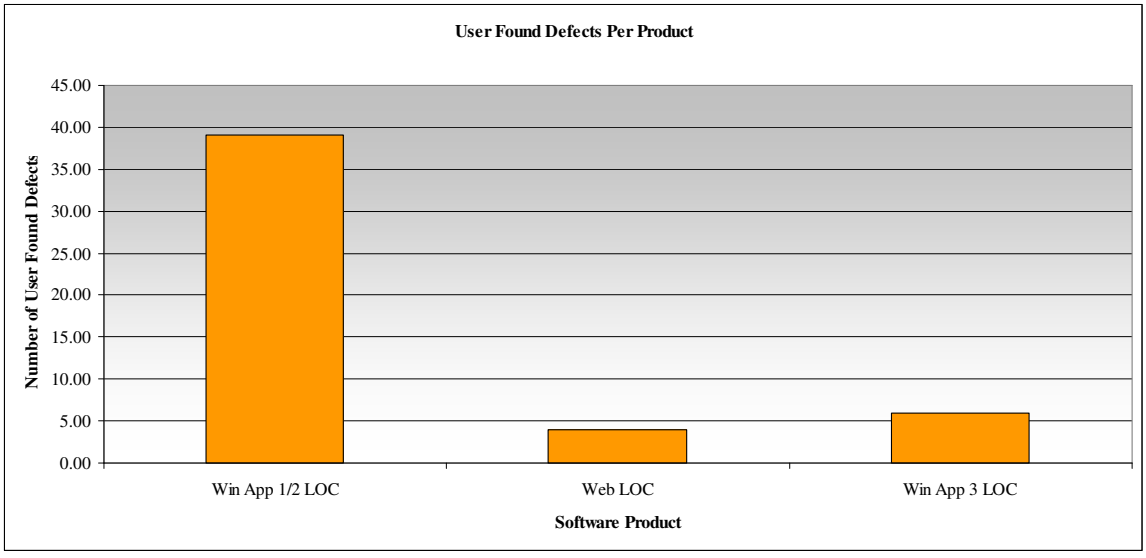


Figure 7-4: Number of User Found Defects Per Product

Figure 7-5 shows the defect counts discussed above and displays them as a defect density. The defect density is the number of defects per unit of size. In this case, the unit of size is the number of kilo statements or thousand statements in the system. The use of statements was chosen over line of code, because the line of code count generated by

DevMetrics includes comments and blank lines, which should not be considered as part of the code when calculating defect density (Fenton and Pfleeger 1997).

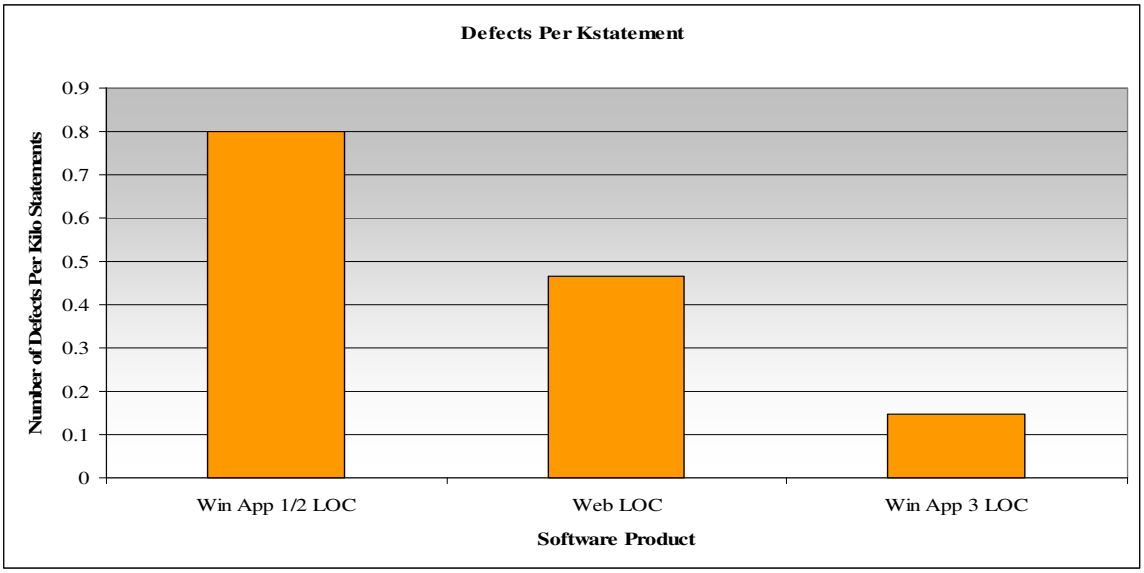


Figure 7-5: Defect Density of User Found Defects

Figure 7-5 shows that the defect density of the windows application was much higher then the other two applications. The website is about half as much as the windows application and the third windows application is half of that again. Initially this may seem like an excellent result, but based on the fact that the user defect reporting is so poor, making conclusions based on these results would not be advisable. There is a possibility that the quality of defect reporting has changed but I have no way of quantifying that other than saying based on my personal opinion I don't believe that the defect reporting quality has changed.

7.4 Developer Opinions

In this section, I will discuss the developers' opinions of test-driven development and continuous integration.

7.4.1 Pre test-driven development Questionnaire

At the beginning of June 2004 a pre test-driven questionnaire was given to the developers to get an idea of how much experience each developer had with test-driven development or unit testing in general.

The developers were first asked how much experience they had with test-driven development. All but one of the developers said they did not have any experience with test-driven development. One developer said "*Yes and No, I haven't used it for large project[s], but I've looked at the Junit quickly just for myself*". This indicates the developer has used unit testing but not Test-driven development.

The developers were asked if they had any experience with software unit testing. Three of the developers said they had experience with software testing, but they had not done any unit testing. The one developer, who had used unit testing, was the Scrum master. He said "*Yes, defined and developed testing environments and automated tests. I have two years of experience in software testing and regression testing*". The other developer who had used software unit testing in the past said "*I have used it for 15 years*". A third said that they had used it "*Just for development testing and QAing*" QAing is quality assurance.

The developers were asked for their opinion on writing test code before writing production code and when it may be good to write test code first and when it may not. Many of the developers thought writing test code first may be a good idea and it could provide some benefits in understanding the problem and tasks better. One developer wondered if writing tests first may be time consuming and may not happen if they are under time pressure.

"*Neat idea. Challenging, since I haven't tried it before. I think it could be a good thing since it forces the programmer to think about the construct of the logic before coding it. Forced design, you might say. Then, you have tests all ready to go when you're done. And a suite of tests built-in for regression testing as the system matures*"

"*I believe that writing test code before will force the developer to better understand the problem* space"

"*I think it is very time consuming so if you're pressured for time constraints then it would be hard to write all the tests*"

The final question asked of the developers was, if they had the choice to write test code before writing production code, would they choose to write test code before production code. The response was they would like to write tests before production code. Overall the responses seemed positive, but some were cautious at the same time.

"Today, I'd write the production code first, because that's how I think and I'm used to that. Which isn't to say that I think this is the only/best way"

"*I would choose to write test code, because if I need to make a change in the future I will know if I will break the program with my new changes*."

At the end of the questionnaire the developers were asked if they had any additional comments that they would like to add. One developer added that it will take time to establish the habits of test-driven development and to see any results.

"*It will take time to establish the habit of doing test-driven development and therefore it will take time for PetroSleuth to really benefit from the techniques*"

7.4.2 Post Test-driven Questionnaire

A questionnaire was given in January 2005 to get an idea of the developers' opinions after using test-driven development and continuous integration for 8 months.

The developers were first asked if there were any obstacles preventing them from using test-driven development with continuous integration. The main obstacles noted for using test-driven development with continuous integration, were learning to use the process and the build breaking due to environmental rather than code issues. As part of their comments, one developer noted that due to the sparse specifications they received, it

was sometimes difficult to figure out what the expected behaviours should be and therefore hard to write tests.

"*This required a change in thinking and ingrained habits. Initially it is easy to forget to write tests first, and it's tempting to just go ahead and code...Main issue was very sparse specifications for the items we are building. (It's difficult to come up with tests when expected behaviours are not defined.) Another major issue was the propensity of the build to break. Fixing this can use up a lot of time, and becomes an obstacle to productivity when the break is environmental and not code failure*."

"*The main obstacles were learning how to write tests and use Nunit effectively***"**

The developers were asked what they liked and disliked about test-driven development with continuous integration. All of the comments were positive. The developers liked the fact that using test-driven development with continuous integration gives the developer feedback, not only about the source code but also about the integrated system. The developers liked the immediate feedback test-driven development and continuous integration gave with the code they wrote. One developer however, dislikes waiting for the build to complete.

"*This is very valuable in determining if code works as expected at all levels, not only in the unit-level, but also within the integrated whole. It is more productive to fix unexpected errors while the code is fresh in your mind, and before others have built on top of it. To summarize, I appreciate the way in which it drives quality in our coding, and the feedback it gives the programmer in the integration of the new code into the whole*."

"*Love it!!! Great way to be sure at all time where you code is and how it is integrating with the overall code package*"

"*I really like the fact that I'd know right away if something was broken.*
*I didn't like to wait for the build to be complete*d."

The developers were asked if they think using test-driven development with continuous integration had any effect on the quality, testability, readability, or maintainability of the code. The developers responded by stating that, in their mind the

code is better, and therefore, will have the attributes of being readable, maintainable and testable. Again it was left up to the developers to determine what quality, testability, readability, or maintainability meant to them.

"*Absolutely, it forces the developer to preplan his code and his approach to the problem. It also gives you the comfort of knowing after every change if your code will still work with previous tests. When the tests fail it also helps to clarify and better discover your errors.*"

"*Definitely the quality of the code is higher than it would have been otherwise, and I would guess it will be easier to maintain this code down the road.***"**

"*Yes, I know this by the amount of times I have to go backwards and fix code… when breaks occur now, I know why and where it happens relatively quickly."*

The developers were also asked if they thought test-driven development had any effect on knowledge transfer and learning. Some of the developers said they thought test-driven development provided a way for people to learn different testing techniques and to better think out the code. It also allowed them to discuss the process with the other developers. Some developers thought there was minimal to no knowledge transfer or learning with using test-driven development

"*It has encouraged a specific kind of learning. We have learned different testing patterns from each other. Each of us contributes additional tests to the objects and as a group our testing coverage gets better. Also as we build more tests, we are learning how our architecture actually functions and how it is best used. As we add new tests, we have gone back and updated older objects, and patterned them the same way as the new ones. I think this reinforces the new techniques and breaks up old coding habits… for those of us that have bad, old, coding habits*"

"*I guess it helped to share the information in some way, because all developers have their own styles of writing code, so through tests we could all learn about how the code written by someone else works."*

The developers were asked if they would use test-driven development with continuous integration in the future. The developers all said they would like to use it in the future. Several of the developers said they wanted to use it as it increased their confidence in the code. One developer also said that the customer was pleased as they

feel that we are building a quality product by using test-driven development and continuous integration.

"*Yes, I believe that in improves the quality of code that is produced*"

"*YES! I believe this is very beneficial and is well worth pursuing. The customer is also pleased with the outcomes, and feels confident that we are building a quality product*"

The developers were asked if they would recommend test-driven development with continuous integration to other development teams. All of the developers said that they would recommend it to other development teams.

The next question was to discover if the developers thought anything could be done to improve the test-driven development and continuous integration experience. Most of the developers said they did not have any suggestions to make the experience any better. Two of the developers did offer some suggestions. One developer wanted a better test database, because of the tests dealing with the business logic that required a test database rather then mock objects. Another developer commented that they would like to have better specifications so that tests could be created to test against these specifications.

"*As usual, I think that better sharing and communication among members of our team would be great. (Ideas for new tests could be discussed during the Scrum, for example.) I also think that if we had a framework of some specifications of what is the expected behaviour of the component we are building, then we could start by building tests to check for these behaviours*"

The developers were asked if they had any comments about test-driven development with continuous integration. One developer commented that the time to rebuild and create tests is becoming time consuming and it may be nice to create a better architecture to try and mitigate it.

"*We are finding that rebuilding and running tests is becoming time consuming as the code base becomes larger. It would be beneficial if we could optimize our architecture somehow to minimize the impact of this. Either to package code into larger solutions, to compartmentalize the build somehow… or some idea no one's suggested yet*"

The final two questions dealt with asking the developers if they would use agile methods in the future and if they would recommend using agile methods to other software development teams. To both questions the developer said yes.

"*YES. Definitely! I really like the continuous feedback aspect of the agile methods, and the flexibility also. We avoid huge deliverables, major drop-dead dates, and so on, and instead deliver smaller packages of functionality which are well-tested, and are directly meaningful to our customer. We don't go too far off track in terms of designing what the customer wants. They can see what's being delivered and can feel confident that progress is being made and is measurable*"

"*I would. IT has a bad record in project/product delivery. This may not be a silver bullet that magically fixes all problems, but it goes a long way to remedy the problems existing in classical project management, such as delivering the wrong product, going over budget, missing deadlines, poor quality code, and so on*"

"*I would, it has been the only real method that has worked that I have been involved with*."

7.4.3 Final Wrap Up Interview

Questions relating to test-driven development and continuous integration were asked as part of the final wrap up interview conducted with the developers. The developers were asked what their thoughts were on test-driven development. The responses were all very positive with respect to using test-driven development as part of the programming process. Some of the comments stated that writing the unit tests gave them more confidence in the code and they think that it saves them time in finding bugs. One developer described a "game" that he and another developer play, where one pair writes the tests first, and then the other pair has to implement the function to make the test pass. This seems like a good way to practice test-driven development in a pair setting. The developer who mentioned "the game" said he only uses the technique when

he pairs with a certain other developer. He also commented that he has not seen any other pairs use the technique.

"*Since I have really started using it, I think it is great and how you should develop. I remember when we used to never do testing and it would take me forever to chase bugs, and now if the tests are written properly or well, I can find it relatively quick and find the problem quick, even if they are written not that great I can usually find the ballpark of where the error is, so finding and tracing something it helps 1000x to what we used to do, it also actually make me more confident in the code that it has been tested and not just something in the developers head. Also the way that me and dev 10 do it. I write the tests he writes the code to pass it, he writes the test and I write the code to pass, I think it is a good way of learning and driving the other developer to code the way you want them to code*"

"*I think it has a very positive effect on our development, for a couple of reasons, one is that it makes you build testable code, the second thing is that you do test your code, we do, do it on a regular basis to our code so that tells us our code solid as we know, that does not mean that you wont find any bugs or new things that you need to test for but I think it gives us a more solid base of code, it does takes a little longer but I really believe we save that time not having to track down flaws in the code*"

The developers were asked to comment on how they determined what to test and what not to test. The two developers, who said that they did test-driven development, said that if you write the tests first, then everything you code should be tested. They said the kind of code they do not test for is in unreachable regions that contain exception code, such as catch statements. Other developers commented that they try to test everything except for GUI code which is very hard to test using unit testing frameworks.

"*Well, I would say that we are supposed to test everything that you are going to write, you are supposed to write the tests first then implement the methods to pass the tests .but no I will not tests whether or not the compiler will work properly*"

"*The only things that I would not test are things that deal with user interaction with the GUI. And that also relates to website because basically all you can do is test a click, how can you test what picture is displayed on the screen, most of it is visually based but you can't test that. I guess that goes back to how you build them, so we need to build them in a way so that most of it there is testable*"

The developers were asked how they knew they had enough tests. The developers, who said they played the testing game, said with test-driven development you always have the right amount of tests because the code should be written to pass the tests. Other developers mentioned their use of Ncover to check and see the test code coverage of the assemblies (NCover 2005). One developer mentioned that if you write code before the tests, then you will never know if you have enough tests. I think that this is due to the fact that, in order to know you have enough tests, you have to know the function very well.

"*Well if you use test first approach then you write your tests as you are building so when your application passes the acceptance test and the clients says it is doing what I want , if you wrote tests for everything that you wrote then you would have enough tests for the current release I would say. And as you find bugs you will write more tests. So it is not a number per say. Like, if you work in a restaurant how many times do you have to wash your hands, well you wash them when you need to. So you write tests as you go, if you do it properly then you have enough tests when you are done, if you wrote code first and then tests then you will never know*"

"*You never know that you have enough tests, the only thing you can try to do is that we have the tool ncover to see if all the lines are tested, it does not tell you if the tests are good or if you have enough of them. You just know you ran something that tried to run that line of code*"

The developers were asked what their thoughts were about automated builds (used with continuous integration) and continuous integration. The developers really liked continuous integration. One developer commented that with continuous integration, unlike other process they had seen, you get integration feedback continuously rather than at the end.

"*I really like it, it is the first time I have used these techniques and I am very much a fan of it, it puts the onus back on teams of developers to make sure that their code fits with the whole and we are continuously testing that everything will work together as expected, and all too often in the past I have seen software development cycles where you don't find out until the end that highly critical pieces will not work together because the nature of the developer who likes to write code and is not very interested that everything hangs together.*"

"*It is very helpful so that you know that what you just did, the changes you just made got integrated properly. So you get more feedback right away. You push it in and the build starts and the tests are run and it tells you if you broke anything. The Nant part is great, so automated builds are necessary and the continuous integration of everything is necessary because you need to know what is going on*"

The final question was about what the developers do when the build breaks on the build server and they are notified that it is broken by the CcTray program(CruiseControl 2005). Many of the developers said that they would look to see if their code broke the build. If their code broke the build they would go and fix it. If the code was broken by another developer they would go and check to see if they knew it was broken and if they wanted some help to fix it.

"We all have the monitor[cctray] up that tells you when the build breaks, and typically at my desk I would open that up and go have a look to see what was last checked in, if it was my code I would try and figure out what I did to break it, if not then you make sure the person who last checked in knows, you go talk to them and see what they were doing and see if they are working on it and try to help them out if you can, sometimes it's a combination of things and its not just one of you that did something so you kind of have to work together"

"I wouldn't get the latest version of everything, I would check my local copy, I would check who broke it and would go tell them in a professional way. Usually the developers are pretty good about it and already know about it. Lately I don't do anything, I note it is broken and if it is broke 24h after I notice it then I will go tell them something"

7.5 Discussion

How often will developers use TDD and CI?

Initially the developers were not using test-driven development. They were creating unit tests after creating the production code. This pattern continued until a developer was added in November 2004. He and his pair partner started to use test-driven development frequently. Based on my observation, they were the only two developers who used test-driven development daily in their software development. The other developers tried test-driven development but frequently forgot to write the tests first. From these results, I would say most of the team did not use test-driven development. This is in conflict with the developer feedback, from the questionnaire and interviews,

where they say they used test-driven development. I think they were using the words test-driven development when they really meant unit testing.

The developers used continuous integration regularly, since continuous integration is passive, in the sense the developer only has to monitor the build status and act only when the build breaks. Through observations, I did see the developers monitor the status of the build using the build monitor program on their desktops.

A secondary question was; since the developers did not seem to be using test driven development, did they use automated unit testing? The developers did use unit testing (Figure 7-1, 7-2, and 7-3). Figure 7-1 showed the percentage of system code which was test code. This number started at 15% of statements in June 2004 and increased to 30% of statements by April 2005. Figure 7-2 showed initially there were a small number of assertions, but over time the number of assertions increased. The increase in assertions meant the team was doing more comparisons between an actual and expected value. The test code coverage for the team increased from 30% to 60% over the 8 months. These results indicate, that even though the team was not writing tests first, they were still putting a lot of effort into writing unit tests for the system.

What are some possible factors influencing the use of TDD and CI?

There were a number of factors that influenced the use of test-driven development. One factor was, in many cases, the developers were trying to get the functionality finished and therefore did not want to write the tests before creating the code. When test-driven development was introduced, this response was very common and this view continued until the end of the study. Another factor was the time needed to write the unit tests themselves. One portion of writing tests, which the developers said took a lot of time, was that the test data had to be created and then entered into a test database. This gathering and entering of test data took considerable time due to the fact large amounts of data were required to verify the system. This meant when a developer wanted to get started, they could either start the implementation or spend time gathering and entering test data. The developers would usually choose to start implementing the

functionality first. A final factor influencing the use of test-driven development was that developers had trouble writing unit tests. Through my observations and the developers' comments, it was obvious that the developers at times did not know how to test a method so, instead, implemented it and then wrote the test.

There were two main factors influencing the use of continuous integration. First, the continuous integration process is mostly passive in nature. This meant that developers do not have to change their development practices to use it. Continuous integration from their perspective is a program that runs on their desktop and tells them when the build breaks or succeeds. The second factor in the use of continuous integration was the fragility of the build process. Initially the build would frequently break. The cause was not due to coder error but due to the complexity of the build process. This caused many build failures unrelated to the developers' code. Over time, this problem was fixed by making the build simpler such that build failures did not occur unless there was a test failure or compilation error. When failures occurred because of the fragility of the process the developers would pay less attention to the broken build. Once the build process became stable, they focused more attention to the build status because before it was a false alarm and now it was because something really was wrong with the build.

What do developers think of using TDD and CI?

Prior to the introduction of test-driven development, the developers thought writing test code before production code may provide some benefits in better understanding the problem and the tasks. However, writing tests first may be time consuming and may not be done if the developers are under time pressure. After using test-driven development for 8 months, the developers mentioned some problems they encountered while using test-driven development and continuous integration. The developers discovered three obstacles: First, the build would break due to the complexity of the build rather than a problem with the source code. Second, the developers sometimes had difficulty translating the task specification into a test. This is an odd comment as a difficult requirement specification should also cause difficulty in implementing the functionality, test-driven or not. Third, the developers said they had

difficulty learning how to write tests and use the testing framework. From my observations I would say the first and third obstacles (breaks in the build and hard to write unit test) were more common than the second obstacle (difficult requirements). The build (first obstacle) initially did break often, but over time it was fixed as the build became simpler. The third obstacle, of having difficulty writing unit tests, resulted in the developer writing the code first and writing the test after. I believe the reason for this is the developer did not know what they wanted to test until they had finished writing the function.

The developers commented on their likes and dislikes about test-driven development with continuous integration. They liked how the testing along with the continuous integration gave them valuable feedback on the integrated system but did not like to have to wait for the product to build. The developers stated that with test-driven development, they had to pre-plan their functionality and it gave them more confidence in the code they were writing. Some developers felt by writing tests first, it helped transfer knowledge to other developers, especially if they also use pair programming. The developers thought that test-driven development with continuous integration improved the quality of the software produced and would recommend it to other teams. Through my observations and discussions with the developers, they also did not like the fact that the test database was difficult to use.

Does the use of TDD improve the defect density of the software produced?

Figure 7-4 and 7-5 showed the use of test-driven development improves the defect density of the software produced. This result however, may not be valid. There are a couple of factors such as poor reporting of defects by users and the poor recording of defects by the developers which make it difficult to determine whether defects actually have decreased. I also believe the results in these charts are skewed toward the defect density decrease because the third Windows application and the Website have not been used as much as Windows application 1 and 2. Therefore, defect reports would not be as likely on the new application as with the previous applications, which were heavily used. A reason for the poor user defect reporting is that the internal customers are now

accustomed to dismissing the error and continuing using the application, rather than reporting the problem. In previous applications the customers were accustomed to errors happening frequently. They generally dismissed these errors, as this was the fastest way to get back to work. Based on this, there could be a number of errors present, but they are not being reported.

CHAPTER EIGHT: SUMMARY AND FUTURE WORK

8.1 Limitations

Process Conformance

Within PetroSleuth, Scrum, pair programming, test-driven development and continuous integration had difficulties with how well the developers followed the processes. Scrum initially had difficulties with sprints being extended but, over time, this ceased and the developers and customers conformed to fixed length sprints as described by Scrum. Pair programming had initial process conformance issues. Even though the developers were asked to pair program all production code, they did not use pair programming rigorously. However, after some intervention by my supervisor, the developers conformed to using pair programming for entire tasks rather then only for debugging and reviewing. One practice that had poor process conformance throughout the entire study was test-driven development. For the most part the developers did not use test-driven development but instead wrote unit tests after writing the production code. One group however started to use test-driven development frequently; but, this was in the last few months of the study. Continuous integration had good conformance with the developers taking action to fix builds when they broke.

Internal Validity

Since ethnological methods are being used, I, as the researcher, am part of the research environment I am studying. I have a direct impact on the environment as I am trying to change the environment itself. This influence has an impact on the results that are produced through possible expectancy effects due to my enthusiasm with agile methods in general (Lefton, Boyes, and Ogden 2000).

Other possible problems being embedded in the environment are the “placebo effect” and the “Hawthorn Effect” (Lefton, Boyes, and Ogden 2000). The placebo effect is where a change occurs because of a person’s expectation. While the Hawthorn effect

occurs where people behave differently when they know they are being observed. The placebo effect in the field of psychology is described as “temporary in nature” and decreases with time (Lefton, Boyes, and Ogden 2000). The Hawthorn is also “temporary once the participants are comfortable being observed” (Lefton, Boyes, and Ogden 2000). I was aware of these effects but believe that they were reduced due to the length of the study, the length of time I was embedded in the environment, and the fact that the team, in general, was focusing on completing the project underway. There could also be bias in the analysis of the results for Scrum, pair programming, and test-driven development and continuous integration as there is only one researcher conducting the study. I have tried to reduce this bias through frequent and detailed consultations with my supervisor.

External Validity

The external validity is how well the findings can be generalized. The team in this case study was selected in large part due to the eagerness of both PetroSleuth and myself to change the previous process. The team was chosen as they were the developers were employees at PetroSleuth and the customers were the employees of the client company.

One threat to the external validity of the study is the team which is relatively small with many developers just out of university. This meant that the results might be closer to student expectations rather than that of experienced developers. However, the mix of some experienced and inexperienced members may be representative of other small teams using a mix of recent graduates and experienced team members. Other concerns to external validity are that the projects at PetroSleuth may not be similar to projects worked on by other small teams. I have provided detailed context of the software developers, the processes they use, and the projects they worked on. The information will assist the reader in determining if the study is valid in their context.

Other Limitations

One limitation of the study is the low number of data points for many of the pair programming and test-driven development quantitative results. This limitation is due to sprints taking one or more months to complete.

Two final limitations are that the team was not constant through out the entire study and the projects were different sizes and sometimes used different techniques. There were instances of developers leaving and new developers joining the company at all states of software development and across projects.

8.2 Summary

At the beginning of this thesis, I identified the main goal for this research as:

Explore the introduction and use of Scrum, pair programming, and test-driven development with continuous integration in a small company over the long term in the context of a small company environment.

This goal was achieved through exploring the following:

1) **The introduction and use of Scrum as described in Chapter 5.**

I explored the use of Scrum from its initial introduction in May 2004 until the end of the study in April 2005.

Scrum provided a sustainable pace for the developers through a reduction in the mean amount of overtime and in the variance in overtime worked. While providing a sustainable pace, Scrum also increased the satisfaction of the customer and developer group for the software produced.

Prior to Scrum, the customers felt “out of the loop” and were ambivalent and unsatisfied with the software being produced. After Scrum was introduced, the customers

felt more involved in the software being developed and in the process. They were more "in the loop" and had a more defined role in the software process. The use of Scrum has transformed the customers from being passive to active participants in the process.

Initially the accuracy of the estimates for backlog items was suspect and showed a wide range of misallocated hours. After the first sprint, the misallocated hours for the estimates decreased, but the variance in hours misallocated fluctuated significantly. This fluctuation is moderately correlated with the size of the backlog item, meaning larger items had more misallocated hours. There was no significant relationship however, between backlog item size and the percentage error in the backlog item estimates.

The development team and the customers adhered well to some of the Scrum practices. These practices were sprint planning, daily Scrum meetings, and sprint review. There was one main problem however. One practice not adhered to was having fixed length sprints. After extending two sprints (sprint 2 and 3) the developers and customers realized that they did not want to extend sprints any longer and decided to stay with fixed length sprints, as close to 30 days in length as possible.

There were two main issues encountered by the team during the use of Scrum. The first issue was that the customers were not well-enough prepared for the sprint planning meetings. This caused significant frustration with the developers and with the customers. The solution to the problem was found to be twofold. First, the team held a couple days of tutorials with the customer group. Second, the Scrum master worked with the customers during the sprint to get them prepared for the next planning meeting. After the project manager started working with the customers, the customers were more prepared for the planning sessions. A second issue was that the team lost focus in Scrum related meetings. The solution to this was to raise the issues during the retrospectives and remind people to try to stay more focused on keeping the meetings on track.

Prior to the introduction of Scrum, some of the developers and customers were unsatisfied with the products being produced. After the introduction of Scrum, the

developers had a better idea of what they were working on and when requirements needed to be completed. They were more satisfied with the products being produced and were more confident that they were producing the kind of functionality the customer requires.

2) The introduction and use of pair programming as described in Chapter 6

I explored the use of pair programming from its introduction at the end of January 2004 until the end of the study in April 2005.

Initially the developers did not use pair programming often and estimated that they used pair programming between 5 and 20% of the time. They characterized their use of pair programming as informal. Based on the results of the pair programming sheets (Appendix D), the developers used pair programming between 5 to 10% of office hours and 10 to 20% of sprint hours. The developers did not seem to actively try to pair program entire tasks, instead they used pair programming when they encountered a problem and needed assistance to solve it. This pattern changed over the next eleven months. As the study progressed, the developers used pair programming more often, with pairing ranging from 10 to 40% of office hours and 40 to 90% of sprint hours

Possible influencing factors for the use of pair programming provided by the developers included not enough time initially to pair program due to lack of management support and periods of uneven team sizes. These factors contributed to the differing amounts of time spent pair programming. In the beginning of the study, the largest obstacle to pair programming was the time pressures on the developers. This was due to management not giving them enough time to get acquainted with working with a partner. After a presentation by Dr. Frank Maurer, the use of pair programming at the company increased from 15% to 22% of office hours and from 20% to 50% of sprint hours.

When looking at other factors influencing pair programming, such as how tasks were assigned to developers, there was no significant relationship found between the

number of tasks assigned to pairs and the amount of pairing done by the team. However, there was a relationship between the percentage of the sprint hours that were estimated to pair tasks and the percentage of pairing the developers did.

Before starting pair programming, the developers thought pair programming might be useful for learning, debugging, and "figuring out" solutions. After using pair programming for four months, the developers thought pair programming was helpful for learning and commented they felt they learned more while paired. They discovered that they spent less time figuring out solutions as a pair versus working alone. The developers liked brainstorming ideas with their partners and found they were not getting stuck on a bad idea as long.

The developers did not like it when their pair would go too fast, too slow, or they had to work with the same partner all day long. After four months, the developers commented that they found working with another developer all day, very difficult. They held the same view after fifteen months. The developers said they felt pair programming was too intense and they wanted more personal time instead of working with the same person all day.

After using pair programming for four months the developers noticed a positive effect on quality. After fifteen months of pair programming, most of the developers still thought pair programming increased quality, but one developer thought that even with pair programming, the developers were not following good coding practices.

When discussing the effect pair programming may have had on the speed of software development, there was no firm consensus as to whether or not pair programming increased the speed of software development. The lack of consensus in these results, points to the need for more pair programming studies in industry.

The developers thought pair programming had assisted with transferring knowledge between each other. After the first four months of using pair programming,

the developers thought pair programming did have an effect on transferring knowledge between them. In April 2004 a new developer was added to the company and to get them started, they paired with the other developers. They thought that pair programming was very useful to them as it got them quickly up to speed with how the other developers worked. After fifteen months, the developers still thought that pair programming was useful at transferring knowledge between the developers and that it helped developers increase their skills and communicate better.

At the start of the study, the developers used paired programming for pair help and debugging rather than pair programming the entire task from start to finish. When tasks were assigned to an individual, the developers only did pairing when they encountered a problem. After tasks started to be assigned to pairs however, the developers used pair programming for the entire lifecycle of the backlog items and tasks. The developers then decided research tasks would be assigned to individuals.

The developers made four main recommendations:

1. Give developers more individual tasks that are not related to research, to break up the intensity of pair programming.
2. Paired developers should be assigned similar tasks.
3. Developers should create some base rules as to what should be pair programmed and what should not be pair programmed.
4. There should be someone who is experienced in using pair programming at the company to assist with the transition.

3) The use and introduction of test-driven development with continuous integration as described in Chapter 7

I explored the use of test-driven development with continuous integration from its introduction in June 2004 until the end of the study in April 2005.

Initially, the developers were not using test-driven development. They were creating unit tests after creating the production code. This pattern continued until a developer was added in November 2004. He and his pair partner started to use test-driven development frequently. The other developers tried test-driven development but usually forgot to write the tests first, they however did write unit tests for the software they created.

There were a number of factors that influenced the use of test-driven development. One factor was that, in many cases, the developers were trying to get the functionality finished and therefore did not want to write the tests before creating the code. One portion of writing tests, which the developers said took a lot of time, was that test data had to be created and then entered into a test database. Another factor influencing the use of test-driven development was that developers had trouble writing unit tests.

Prior to the introduction of test-driven development, the developers thought that writing test code before production code may provide some benefits in better understanding the problem and the tasks. However, writing tests first may be time consuming and may not be done if the developers are under time pressure. After using test-driven development for 8 months, the developers discovered three obstacles. First, the build would break due to the complexity of the build rather than a problem with the source code. Second, the developers sometimes had difficulty translating the task specification into a test. Third, the developers said they had difficulty learning how to write tests and use the testing framework.

The use of test-driven development reduced the defect density of the software produced. This result however, may not be valid due to poor reporting of defects by users and the poor recording of defects by the developers making it difficult to determine if defects actually have decreased. In addition, the third Windows application and the Website have not been used as much as Windows applications 1 and 2. Therefore, defect reports would not be as likely on the new application as with the previous applications, which were heavily used.

The developers used continuous integration regularly, since continuous integration is passive, in the sense the developer only has to monitor the build status and act only when the build breaks.

There were two main factors influencing the use of continuous integration. First, the continuous integration process is mostly passive in nature. This meant that developers do not have to change their development practices to use it .The second factor in the use of continuous integration was the fragility of the build process. Initially, the build would frequently break. Over time, this problem was fixed by making the build simpler so that build failures did not occur unless there was a test failure or compilation error.

The developers liked how testing, along with the continuous integration, gave them valuable feedback on the integrated system but they did not like having to wait for the product to build. Test-driven development also gave the developers more confidence in the code they were writing.

8.3 Concluding Summary

In summary, the use of Scrum, pair programming, and test-driven development over the long term at PetroSleuth can be considered a success. All of the techniques provided tangible benefits. The introduction of Scrum put controls on the chaos of software requirements and provided the developers with a sustainable work environment while increasing developer and customer satisfaction in the product. Pair programming assisted the team in transferring knowledge between each other and increased the quality of the software produced. Finally, test-driven development with continuous integration gave the team confidence that their software was working properly, through the use of unit testing and continuous integration, which integrated the system frequently.

8.4 Future work:

There is a significant amount of work to be done in the future in the area of agile software development methods. This study represents only a small step in addressing the issues involved with the use of agile practices and methods such as Scrum, pair programming, and test-driven development with continuous integration over the long term. It would be beneficial to attempt to replicate the study in a similar environment with a person experienced in introducing agile methods into companies contemplating the introduction of the practices. It would be helpful to attempt to conduct the study without time pressure to allow the participants to have time to learn and use the practices. In addition, future studies should be conducted on pair programming and coding standard adherence. The test-driven development portion of the study should be run again after improvements in the management of defects are implemented.

Other future work, as a result of the study, will be to test and validate the many hypotheses formulated as part of this study.

1. Scrum will provide both a reduction in the magnitude and in the variance of overtime hours of the developers while increasing customer satisfaction.

2. Creating small backlog items will have fewer misallocated hours.

3. The more task hours estimated for, and assigned to pairs the more pair programming will take place by the developers.

REFERENCES

Abrahamsson, P & Koskela, J 2004, 'Extreme Programming: A Survey of Empirical Data from a Controlled Case Study', *ACM-IEEE International Symposium on Empirical Software Engineering ISESE'04*, Redondo Beach, CA, USA, 2004.

Ambler, S W 2002, *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process***,** John Wiley and Sons Inc, New York.

Astels, D, 2003, *Test-driven Development: A Practical Guide,* Prentice Hall, Upper Saddle River, NJ.

Beck, K & Andres, C 2004, *Extreme Programming Explained: Embrace Change Second Edition*, Addison-Wesley, USA.

Cockburn A & Williams L 2001, 'The Costs and Benefits of Pair Programming', *Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering*, Italy.

CruiseControl, 2005, Accessed: May 20 2005, http:// ccnet.thoughtworks.com/

DeClue, T H 2003, 'Pair Programming and Pair Trading: Effects On Learning and Motivation in A CS2 Cours**e'**, *Journal of Computing Sciences in Colleges,* vol. 18, no. 5, pp. 49-56.

DevMetrics, 2005, Dev Metrics 1.0, Accessed: September 20 2004, http://www.anticipatingminds.com/Content/Products/devMetrics/devMetrics.aspx

ECMA, 2002, Standard ECMA-334 C# Language Specification, Accessed: May 20 2005, http://www.ecma-international.org/publications/standards/Ecma-334.htm

Fenton N E & Pfleeger, S L 1997, *Software Metrics: A Rigorous & Practical Approach*, 2nd edn, PWS Publishing Company, Boston, MA.

Fowler, M & Foemmel, M 2005, *Continuous Integration*, Accessed: May 20 2005, http :// www.martinfowler.com /articles/continuousIntegration.html

George, B & Williams, L 2003, 'An Initial Investigation of Test-Driven Development in Industry', in *ACM Symposium on Applied Computing (SAC),* March 2002.

Guy Elizabeth S 2000, 'Methodological constraints in researching systems development work', *Workshop Proceedings of ICSE 2000, 22nd International Conference on Software Engineering*, Limerick, Ireland, June 4 – 11.

Hanks, B & McDowell, C 2004, 'Program Quality with Pair Programming in CS1', *ACM SIGCSE Bulletin,* vol. 36, no. 3, pp. 176 - 180.

Hulkko, H & Abrahamsson,P 2005, 'A Multiple Case Study on the Impact of Pair Programming on Product Quality', Proceedings of ICSE'05, May 15-21, St.Louis,Missouri, USA

Junit Testing Framework, 2005, Accessed: May 20 2005, http://www.Junit.org/

Karlsson, EA, Andersson, LG, & Leion, P 2000, **'**Daily Build and Feature Development in Large Distributed Projects', *Proceedings of the 22nd International Conference on Software Engineering,* Limerick, Ireland, pp. 649 – 658.

Klein, H K & Myers, MD 1999, 'A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems', *MIS Quarterly*, vol. 23, no. 1, pp.67-94, Accessed: March 20, 2005, ABI/INFORM Global.

Lefton, LA, Boyes, MC, & Ogden, NA 2000, *Psychology*, Canadian Edition, Prentice-Hall Canada Inc, Scarborough, Ontario.

Lui, KM & Chan, KCC 2003, 'When Does When Does a Pair Outperform Two Individuals?', *Proceedings of XP 2003*, LNCS 2675, Springer-Verlag Berlin Heidelberg, pp. 225 - 233.

Manifesto for Agile Software Development, 2005, Accessed: May 20 2005, http://agilemanifesto.org/

Maximillien, M & Williams, L, *Assessing Test-Driven Development at IBM*, International Conference on Software Engineering, May 2003.

McDowell, C, Werner, L, Bullock, H, & Fernald, J 2003, 'The Impact of Pair Programming on Student Performance, Perception, and Persistence', *Proceedings of the 25th International Conference on Software Engineering ICSE'03*, IEEE, pp. 602 - 607.

McDowell, C, Werner, L, Bullock, H, & Fernald, J 2002, 'The Effects of Pair-Programming on Performance in an Introductory Programming Course', *Proceedings of 33rd SIGCSE'02*, Covington, Kentucky, USA, pp. 38-42.

Melnik, G & Maurer, F 2005, 'A Cross-Program Investigation of Students' Perceptions of Agile Methods', *Proc. 27th International Conference on Software Engineering (ICSE 2005)*, ACM Press, 2005, St. Louis, Missouri, USA.

MountainGoat, 2005, Mountain Goat Software Scrum Process, Accessed: June 1 2005, http:// http://www.mountaingoatsoftware.com/Scrum/

MSDN ReadLine Documentation, 2005, Accessed: July 22 2005, http:// http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemIOStreamReaderClassReadLineTopic.asp

Muller, M & Hagner, O 2002, 'Experiment about Test-first programming', *Software, IEEE Proceedings*, vol. 149, no. 5, pp. 131-136.

Nagappan, N, Williams, L, Ferzil, M, Wiebe, E, Yang, K, Miller, C, & Balik, S 2003, 'Improving the CS1 Experience with Pair Programming', *Technical Symposium on Computer Science Education Proceedings of the 34th SIGCSE'03*, Reno, Nevada, USA, pp. 359 - 362.

Nant Automated Build Tool, 2005, Accessed: May 20 2005, http://nant.sourceforge.net/

Nawrocki, J & Wojciechowski, A 2001, 'Experimental Evaluation of Pair Programming', *Proceedings of the 12th European Software Control and Metrics Conference*, ESCOM 2001, London, pp. 269-276.

Ncover Code Coverage Tool, 2005, Accessed: May 20 2005, http://www.ncover.org/

Nosek, J T 1998, ' The Case for Collaborative Programming', *Communications of the ACM,* vol. 41, no. 3, pp. 105 - 108

Nunit Unit Testing Framework, 2005, Accessed: May 20 2005, http://www.nunit.org/

Palmer, S & Felsing, J 2002, *A Practical Guide to Feature Driven Development*, Prentice Hall, Upper Saddle River, NJ

Rautiainen, K, Vuornos, L & Lassenius, C 2003, 'An Experience in Combining Flexibility and Control in a Small Company's Software Product Development Process', *International Symposium on Empirical Software Engineering (ISESE'03),* Rome, Italy, pp. 28-39.

Rising, L & Janoff, NS 2000, 'The Scrum Software Development Process for Small Teams', *IEEE Software,* vol.17, no. 4, pp.26-32.

Robillard, PN, Kruchten, P & d'Astous P 2003, *Software Engineering Process with the UPEDU*, Addison Wesley, USA.

Sanders, D 2003, 'Student Perceptions of the Suitability of Extreme and Pair Programming', *Extreme Programming Perspective*, Addison-Wesley, USA, pp. 261--271.

Schwabe,r Ken 2004, *Agile Project Management with Scrum*, Microsoft Press, Redmond, WA.

Sharp, H, Woodman, M & Robinson, H 2000, 'Using Ethnography and Discourse Analysis to Study Software Engineering Practices' , *Proc. ICSE'2000 Workshop on Beg, Borrow or Steal*, Limerick, Ireland, pp. 81-87.

Stapleton, J 1997, *DSDM Dynamic Systems Development Method: The Method in Practice*, Addison-Wesley, USA.

Succi, G., Marchesi, M., Pedrycz,W & Williams, L. 2002, 'Preliminary Analysis of the Effects of Pair Pprogramming on Job Satisfaction', *Fourth International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2002).*

Sutherland, J 2001, 'Inventing and Reinventing SCRUM in Five Companies'**,** *Cutter IT Journal*, vol. 14, pp. 5-11.

VanDeGrift, T 2004, 'Coupling Pair Programming and Writing: Learning About Students' Perceptions and Processes', *Proceedings of the SIGCSE Conference*, SIGCSE'04, Norflok, Virgina, US.

VersionOne, 2005, VersionOne Agile Planning Tool, Accessed: May 20 2005, http://www.VersionOne.net

Wake, WC 2004, *A Scrum Project-brief Experience Report***,** Accessed: May 20 2005, http:// www.controlchaos.com/module/practicing_wake.pdf

Williams, L & Kessler, R 2003, *Pair Programming Illuminated***,** Addison Wesley, USA..

Williams, L & Upchurch, R 2001, 'In Support of Student Pair Programming', *Technical Symposium on Computer Science Education, Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science,* pp. 327 - 331.

Williams, L & Kessler, R 2000, 'Experimenting with Industry's Pair-Programming Model in the Computer Science Classroom', *Journal on Software Engineering Education*, December 2000.

Williams, L & Kessler, R 2000a, 'The Effects of Pair-Pressure and Pair-Learning on Software Engineering Education', *Software Engineering Education & Training. Proceedings. 13th Conference,* pp. 59-65.

Williams, L, Kessler, R, Cunningham, W & Jeffries R 2000b, 'Strengthening the Case for Pair-Programming', *IEEE Software*, July/Aug 2000.

Yin, RK 2003, *Case Study Research: Design and Methods***,** Third Edition**,** Sage Publications Inc, Thousand Oaks, California.

APPENDIX A: RAW DATA

1. Questionnaires can be found on data CD-ROM in the questionnaire folder. They are titled Q#Dev#.doc where Q# is the questionnaire number and Dev# is the developer number of the developer who filled out the questionnaire.

2. Interview transcripts can be found on the data CD-ROM in the interviews folder. They are titled I#Dev#.doc, where I# is the interview number and Dev# is the developer who provided the interview.

3. Researcher Notes are available under ResearchNotes.doc

4. Charts and other raw data are provided under ResearchCharts.xls

5. Backlog and Task Information is provided under Sprint Excel Sheets folder

6. Dr.Maurer's Presentation Slides for August 6th 2004 are in PetroSleuth.ppt

7. TDD Presentation Slides used are in TDD.ppt

8. Backlog and task Information under Sprints

9. SQL Query for office hours under SQLQueries

10. Programs developed as to calculate SQL and HTML metrics in Programs vssFileparser.rar

APPENDIX B: RESEARCH METRICS

M11: Mean Percent Overtime by week

Mean Percent Overtime is the percentage of hours worked over the expected amount of hours per developer per week. For example, 20% overtime means on average each developer worked 20% more hours than they were supposed to for that week. The calculation is as follows.

$$\%\ \text{Overtime} = [(\ \frac{ActualOfficeHours}{ExpectedOfficeHours}\ ) - 1] * 100$$

Where:

ActualOfficeHours = Sum of office hours worked from (Sunday to Saturday) of a given week for the team. Office Hours are explained in appendix C.

ExpectedOfficeHours = Sum of office hours the developers should have worked as a team during a given week. The expected hours take into account statutory holidays, vacations, and sick days. Expected Hours is a calculated value.

To calculate the expected hours we first have to calculate how many person days should be included in the week. We start out by assuming that all the developers will work every day of the week, therefore a perfect work week is:

PDPW = # developers employed * 5 days.

From this perfect week, we then subtract the administrative days recorded in the Time Tracker system, as the developers are not expected to work on those days. In the case of the salaried employees, these administrative days are sickdays, vacation, courseday, flexday, compassionate leave.

In the case of the contract employees, such as two of the developers and summer students, the determination of administrative days is different. Persons on contract do not

enter administrative days into the system. If they have a sick day or want to take a holiday, then they would not enter time for the day. For this paper, if the contract employee did not enter a day then it will be counted as an administrative day.

The final expected hour calculation is as follows:

Expected Person Days = PDPW – Administrative Days

Expected Hours = Expected Person Days *7.5

M12: Misallocated Hours Over and Under Estimation

Misallocated Hours Over and Under Estimation is the number of hours over or under the estimated hours for a backlog item. The metric provides two results. First, the metric shows how many hours were over allocated to a backlog item, meaning the backlog item was completed faster than the estimate (negative value). Second, the metric shows how many hours were under allocated to a backlog item, meaning the backlog item took longer than the estimate to completed (positive value).

Misallocated Hours of Backlog= BacklogHoursCompleted– EstimatedBacklogHours

Where

BacklogHoursCompleted = Hours Recorded as completed on a Backlog Item

EstimatedBacklogHours = Hours Recorded as Estimated on a Backlog Item

M14: Percent Estimation Error

Percent Estimation Error is the error between EstimatedBacklogHours and BacklogHoursCompleted. The purpose is to show how close the actual and estimates are as a percentage of the actual completed work. A positive percentage means the estimate was smaller than the work completed meaning an under estimation. For example, if a backlog item was estimated to take 50 hours and the developers spent 40 hours, the calculation would be as follows:

$$\% \ \text{Error} = \left(\frac{\text{BacklogHours Completed - EstimateBacklogHours}}{\text{BacklogHours Completed}}\right) * 100$$

$$\% \text{ Error} = \left(\frac{40 \text{ hours} - 50 \text{ hours}}{40 \text{ hours}}\right) * 100$$

$$\% \text{ Error} = -25\%$$

The use of a negative number to represent over estimation was chosen as this method was used in Abrahamsson and Koskela (2004) to present % Estimation Error.

M15: Pairing Percentage per Sprint

The pairing percentage of the team per sprint based on the hours recorded in the sprint for software development. The percentage of pair programming per sprint is calculated as follows:

$$\% \text{ Pairing by Sprint} = \frac{\text{Number of Pair Hours}}{\text{Number of Sprint Hours}} * 100$$

Where:

Number of Pair Hours = Sum of hours spent pair programming by team. For example, if two developers pair for one hour, then the number of pair hours for the team would be two hours. See appendix C for pair hour calculation

Number of Sprint Hours = Sum of task hours completed on backlog items not related to meetings, sprint planning, sprint review, sprint retrospective, hardware support and configuration.

M16: Pairing Percentage per Month

The pairing percentage of the team per month based on the office hours recorded for that month. The percentage of pair programming per month is calculated as follows:

$$\% \text{ Pairing by Month} = \frac{\text{Number of Pair Hours}}{\text{Actual Office Hours}} * 100$$

Where

Number of Pair Hours from M15.

ActualOfficeHours from M11.

M17: % of pair tasks in Sprint

The percentage of pair tasks in a sprint is the percentage of tasks by task count that were assigned to a pair of developers. The purpose of this metric is to show how many tasks were assigned to pairs in relation to the total number of tasks assigned during the sprint. The calculation is as follows:

$$\%\ \text{Pair Assigned Tasks} = \left(\frac{\text{Pair Assigned Tasks}}{\text{Pair Assigned Tasks} + \text{Individual Assigned Tasks}}\right) * 100$$

Where

$\text{Pair Assigned Task} = \text{Count of how many tasks were assigned to two developers}$

$\text{Individual Assigned tasks} = \text{Count of how many tasks were assigned to only one developer}$

M18: % of sprint estimated for pair tasks

The percentage of the sprint estimated for pair tasks and is the percentage of the total sprint hours estimated for pair tasks verses the total number of hours estimated for all tasks. The purpose of this metric is to show how much of the sprint was assigned to pairs in relation to the total number hour estimated for all tasks.

$$\%\ \text{Sprint Estimated for Pair Tasks} = \left(\frac{\text{Pair Task Estimates}}{\text{Pair Tasks Estimates} + \text{Individual Tasks Estimates}}\right) * 100$$

Where

$\text{Pair Task Estimates} = \text{Sum of Estimates made on tasks assigned to two developers}$

$\text{Individual Tasks Estimats} = \text{Sum of Estimates made on tasks assigned to only one developer}$

M19: Test Code Percentage of System

The test code percentage of the system is the percentage of C# Lines, C# Statements, and C# Members which are test code compared to the application's code. Test code is code created for the purposes of testing the system. Test code percentage is an indication of the relative amount of test code rather than describing the codes effectiveness for testing.

$$\%\,\text{Lines} = \frac{\#\,\text{of Lines of Test Code}}{\#\,\text{of Lines of Application Code}}$$

$$\%\,\text{Statements} = \frac{\#\,\text{Test Statements}}{\#\,\text{Application Statements}}$$

$$\%\,\text{Members} = \frac{\#\,\text{of Test Members}}{\#\,\text{of Application Members}}$$

# of Lines, Statements and Members of Test Code= C# Lines of code, C# Statements and C# Members that are used for testing the system.

Application Lines, Statements and Members = C# Lines of code, C# Statements and C# Members for the entire system including test code.

The lines of code, statement and member counts are retrieved from the DevMetrics metrics program (DevMetrics 2005). The DevMetrics program has the following definitions for lines, statements and members.

Members: the number of members, includes all fields and member functions (events, properties, constructors, methods, etc.)

Lines: the total number of lines counted in C# files and includes blank lines and comments

Statements: Statements are not the same as a line of code. A line may contain more than one statement and there may be many lines in a file that are not statements. Statements are defined in (ECMA 2002)

M21: Test code coverage

Test code coverage is the percentage of code executed when the unit tests are run. Test code coverage shows how much of the system is executed and tested during automated unit testing.

$$\%\ \text{Coverage}\quad \text{for Unit}\quad \text{Tested}\quad \text{Classes}\ = \frac{\#\ \text{Statements}\quad \text{Covered}}{\#\ \text{of}\ \ \text{Total}\ \ \text{Statements}}$$

Where

Statements Covered= Sum of statements executed during unit testing of classes that have unit tests associated with them. Statements are reported by the Ncover test code coverage application (Ncover 2005)

Total Statements = Sum of statements in classes that have unit tests associated with them (as reported by NCover)

The overall system test code coverage is the percentage of the entire system executed during automated unit testing.

$$\%\ \text{Coverage Overall} = \frac{\#\ \text{Statements Covered}}{\#\ \text{Statements in System}}$$

Where

#Statements Covered = the sum of the statements executed during unit testing as reported by Ncover

# of statements in system = Sum of C# statements in the system as reported by the Dev metrics software metrics program (DevMetrics 2005).

NCover works on the compiled assemblies and the DevMetrics tool works on the raw source code.

To quantify the statement count difference between DevMetrics and Ncover (Ncover 2005), I randomly picked weekly code snapshots and ran the code coverage metrics against them using both Ncover and DevMetrics. Table B shows a sample of five different times when both processes were run and shows the comparisons and difference between them.

Table B Ncover Vs Dev Metrics Statement Count

	Ncover statement count	Dev Metrics statement count	% Difference
Oct 30 2004	5735	6061	6
Nov 27 2004	7672	7877	3
Jan 22 2005	17911	18237	2
Feb 12 2005	19717	19981	1
Mar 19 2005	26083	26304	1

$$\%\ \text{Difference} = \frac{\#\ \text{of NCover Statements} - \#\ \text{of Dev Metric Statements}}{\left(\frac{\#\ \text{of Ncover Statements} + \#\ \text{of Dev Metric Statements}}{2}\right)} * 100$$

When looking at the difference between the two there is a maximum of 6% , a minimum of 1%, and a median of 2% difference between the NCover and DevMetrics statement count.

M22: Defects Found per Product

The defects per product are how many user found defects were reported and recorded for a given product in VersionOne. There are three defect counts, one for each product (Windows Application 1/ 2, Website, and Windows Application 3). The number of user found defects was determined by counting the number of backlog items that were classified as a user found defects when entered into VersionOne.

M23: Defect Density by Product

Defect Density is usually considered as the number of defects in a product per line of code (Fenton and Pfleeger 1997). In this case, the defect density calculation is as follows:

$$\text{Defect Density} = \frac{\text{Number of User Reported Defects}}{\#\ \text{of statements in system}}$$

Where

Number of User reported Defects from M22.

# of statements in system from M21.

APPENDIX C: DATA SOURCES

In this section, I describe the data sources and the method of data collection. I also discuss the difficulties associated with the various data sources.

Office Hours

Purpose

Office hours represent the total amount of time the developers are in the office and are available to do work. The purpose of collecting office hours is to get an indication how much time the developers are in the office.

Definition

Office hours are hours spent at the workplace except for a lunch break. On a regular working day if the developer is at the workplace from 8 am until 4:30 pm, the number of office hours should be 7.5 hours. This calculation assumes they took a 1 hour lunch break (8.5 total hours – 1 hour for lunch).

How the Data is Captured

To record the office hours the developers were asked to use an internally developed application called “Time Tracker”. Time Tracker is a windows application that allows the developers to enter their office hours into a database. The hours entered into the database are used by the accountant for billing purposes. The tool gives the developers the ability to categorize their time into different categories: administrative (sick days, vacation, and statutory holidays), time working on client hardware support, and time spent working on software development tasks. For this study, the administrative days and holidays (both statutory holidays and days taken by the developers as part of their vacation time), are filtered out during the data collection process. The reason for removing the administrative hours is that the developers are not expected to work during those hours.

The frequency of when the developers enter their office hours into the system varies with each developer. Some developers enter their time daily while others record the hours in day planners and enter the time on a weekly basis.

Since office hour information is entered into Time Tracker, which uses an MS SQL database, the time information can be retrieved using database queries. In this case, a stored procedure has been created to extract the information, this procedure can be found in Appendix A.

Backlog and Task Information

Purpose

The purpose for collecting backlog and task information is to quantify the amount of time estimated and completed on backlog items and tasks. Backlog and task information also provides the ability to determine which tasks were paired or not.

Description

Backlog Information is made up of the following components:

Backlog Estimate: Estimate in hours for backlog completion

Backlog Actual: Actual hours completed on backlog item

Task Information is made up of the following components:

Task Owner: Developer responsible for task completion

Pair Developer: Developer to pair with task owner

Task Estimate: Estimate in hours for task completion

Task Actual: Actual hours completed on task

How the data is captured

VersionOne provides the ability to export information such as backlog and task lists from a sprint into an Excel work sheet.

Known and Possible Noise in Data

Since it is the responsibility of the developers to enter the update to the backlog and task information in VersionOne on a daily basis, the accuracy of the hours entered is determined by how well the developers keep track of their time (properly entering it into the system) and how much time they spent on a particular task or backlog item.

Pair Programming Sheets

At the beginning of the study, the developers were asked to record how much time they spent pair programming on a daily basis. Having the developers' record how frequently and for how long they paired was to get an idea of how much time the developers actually spent pairing with each other. The developers were asked to record their time as pairs as they did their work. In this way, it was felt that the record of when developers used pairing would be more accurate and representative than trying to ask the developers at the end of the study to determine how much time they spent working as pairs.

The pair programming sheets, which can be found in appendix B have four columns. The first column was for the developers to record what kind of pair encounter they were recording. For example:

1. If they were working on a programming task together, they would record programming in the tasks column
2. If they were pair reviewing each other's code, they would record reviewing
3. If they were doing a pair design session they would record planning

The splitting of the task into different categories was done to get information about the amount of different pairing types that were being used. Initially the developer's used the correct headings in tasks for the different task activities, but very quickly and for the rest of the study, the developers would record pair programming as the task. The second column on the pair programming sheet is titled Time Spent (this is recorded in minutes). The purpose of this column was to record the amount of time the developers spent together working as a pair. The granularity of the entry was asked to be recorded as

minutes because some pair encounters could be less than an hour in length. The developers initially recorded everything in minutes, but over time, they switched to mostly using hours, because the time they spent as pairs became longer. The developer's still record shorter encounters in minutes. The third column on the pair programming sheets was to record the date of the encounter. The date was written as month/ day/year. The fourth column was used to record who their pair partner was for the encounter.

Each developer was given their own pair programming sheet. The developers were asked to individually record all the different pairing encounters they had during the day. The reason for having each of the developers make a record on their own sheets was to have two records available that could be cross referenced to try to get a more accurate representation of the pairing time. After collecting the sheets, I compared the time recorded by each developer to generate the cumulative amount of pairing done by each developer per month and per sprint.

When comparing the records for two different developers, I looked for two differences. The first difference was, if one developer recorded a pairing event (an event is a recording on the pair programming sheet with the same pair, date and task) and the other partner did not. In that case, I would include that time in the other partner's hours, as it was common practice (based on observations and discussions with the developers) that only one of the developers would fill the sheet out and the other partner would copy the time over later in the day. The second difference was where two partners recorded an event, but the amount of time was different. In that situation, I would use the difference between these two numbers to figure out a **minimum amount of pairing** and a **maximum amount of pairing.** These numbers are used later in Chapter 6 to illustrate how much pairing was accomplished.

Source Code Snapshots

The source code is stored in two places. The first is a source control repository called Source Gear Vault. The source control repository stores the .net source code for all

of the on-going projects. The second location where the source code was stored is within the SQL database servers.

The source code is extracted from both of these repositories every night at 11:50 pm by an automated process. The process extracts the code and places it in a folder. This extracted code is called the “daily code snapshot”. The snapshots are compressed and stored for future use.

APPENDIX D: PAIR PROGRAMMING SHEET

Task (Programming, Reviewing, Planning)	Time Spent (in minutes)	Date	Partner

1) When checking code into visual source safe please put into the check in comments who you worked with if you worked with anyone.

2) Also please remember to comment at the top of a method any defects you fixed.

3) Make sure that the tasks you are working on are entered into on time.

When entering a defect into on time can you make a note in the defect description whether the defect was found by a user or by a developer

APPENDIX E: SCRUM STATISTICS

F-Test Two-Sample for Variances		
	Before	***After***
Mean	33.73188	11.95673
Variance	628.2524	68.93559
Observations	69	52
Df	68	51
F	9.113615	
P(F<=f) one-tail	6.31E-14	
F Critical one-tail	1.556272	
t-Test: Two-Sample Assuming Unequal Variances		
	Before	***After***
Mean	33.73188	11.95673
Variance	628.2524	68.93559
Observations	69	52
Hypothesized Mean Difference	0	
Df	87	
t Stat	6.742215	
P(T<=t) one-tail	8.17E-10	
t Critical one-tail	1.662557	
P(T<=t) two-tail	1.63E-09	
t Critical two-tail	1.987608	

APPENDIX F: PROJECT METRIC METHODOLOGY

Here I will discuss the values for the project metrics shown in chapter 3. The C# metrics were generated by running the DevMetrics program against the March 6, 2004 and the April 30, 2005 source code snapshots. The output of the program is an HTML file reporting source code by .net project. The HTML file not only contained the website (which is made up of multiple projects) but also contained project files from older projects also. After generating the report, I went though the list of projects in the html file and added up the metric values reported for each project that was part of the website.

The HTML metrics for the website were generated by a custom program I created, (Vss File Parser on CD-Rom). The metrics are very elementary and consist of physical lines of HTML after removing blank lines. Physical lines in this case meaning the number of end lines. The HTML files included in the count are only the html files are part of the secure portion of the site. There is a front-end marketing and greeting site that customer's access before logging into the secure portion of the site but this was done by an external website and marketing company and is not included in the project metric counts.

The SQL metrics included as part of the projects were calculated as follows:

The first step before calculating the metrics is to get the source code files into an internal structure so that the metrics can be calculated on them. The source code files are read into an internal array structure using the ReadLn() statement. The ReadLn() statement defines a line as the following: "*A line is defined as a sequence of characters followed by a line feed ("\n") or a carriage return immediately followed by a line feed ("\r\n")*." [Msdn 2005]. After the file is read into the program, blank lines are removed from the internal array by using the Trim() function on each lines and then checking if the lines is empty. The Trim() function in .net removes all occurrences of white space characters from the beginning and end of the string. The next step is to find all of the

comments in the file by looking for lines that are fully or partially commented out with the two dashed - - or the multi lines comment of /* */. The program will remove only the commented portion of the source. After the above processes are run the source codes representation in memory is a file without blank lines or comments.

Physical Lines: Physical Lines are the length of the internal array representation of the source code files. This length is equivalent to the number of lines as defined above. The physical lines count is done after the blank lines are removed but before the commented lines are removed.

Commented Lines: Commented lines are the count of the number of lines that are entirely “commented out” meaning that they either start with a - - or are contained entirely within a /* */ block.

Number of Procedures and Functions: The number of Procedures and Functions is the number of stored procedures or user defined functions in the SQL source file.

Number of Select, Insert, Update, and Delete Statements: This number is the summed total of the number of Select, Insert, Update and Delete occurrences in the SQL source code.

APPENDIX G: DEVELOPER CONSENT FORM

Research Project Title: A case study into the Effectiveness of Agile Methods in an Industrial Setting
Investigator: Christopher Richard Mann
Co-Investigator and Supervisor: Dr. Frank Maurer

This consent form, a copy of which has been given to you, is only part of the process of informed consent. It should give you the basic idea of what the research is about and what your participation will involve. If you would like more detail about something mentioned here, or information not included here, you should feel free to ask. Please take the time to read this carefully and to understand any accompanying information.

The purpose of the study is to gather evidence of the effectiveness of agile methods in an industrial setting. Agile methods are new software methodologies, which tend to be lightweight in terms of documentation and try to focus on adapting quickly as possible to changing software requirements.

The study to be conducted will introduce two agile techniques. The first technique to be introduced will be pair programming. Pair programming is where two developers sit at one computer to plan, develop and review code. The second agile technique will be test-driven development using unit tests with continuous integration.

If you volunteer as a participant in this study, you will be asked to participate in two ways. The first way is to record any time you spend designing, planning, programming or reviewing your tasks or code with a partner. This does NOT mean that you MUST program in a pair all the time. There should be approximately 4 to 5 months of pair programming before the second technique will be introduced.
The second way that you will be asked to participate will be when the second agile technique is introduced. During the second portion of the study, you will be asked to write unit test code before you write production code for the software you create.

The data, which will be collected for the study, takes the form of:

- Your Time Tracker time logs. This data will be used to determine who worked on what portion of the software and for how long.

- Software Source Code (including unit tests): This data will be used to perform some qualitative and quantitative measurements. Some of these measurements may include: number of lines of code, number of commented lines of code, number or classes, average size of a method, readability, testability, maintainability.

- Defect and Feature Reports: These reports will be used to perform calculations such as defect density (defects per thousand lines of code), defect rate (defects per week, month) and some other calculations dealing with defects and features.

- Questionnaire and Interview data: Before and after the introduction of each agile technique you will be asked to fill out a questionnaire and provide an interview. The purpose of the questionnaire and interview is to assess feelings before and after the introduction of the agile techniques. The interview may be recorded by an audio tape recorder. The time to complete the questionnaire and interview will be no greater then an hour.

You are being asked to participate in the study because you are a software developer at the company where the study is being conducted. If you do not wish to participate in the study or portions of the study, tell the investigator and they will provide some alternatives, where you would not participate in the tasks of the study but would allow your data to be used.

All information gathered as part of the study will be held in the strictest confidentiality. Raw data such as source code, time logs and defect/feature logs will only be viewed by the investigator Christopher Richard Mann. Only aggregate and process information which cannot be used to identify an individual will be released or published. The questionnaires and interviews will be kept for a period of 1 year, after that period of time the interview and questionnaire sheets will be destroyed. If the interview is recorded by an audio tape recorder then the tapes will only be used to transcribe the interview into an electronic format. Once the transcription is complete, the tapes will be erased. The transcribed data will then be destroyed after 1 year. The aggregate data which is extracted from the raw data will be kept indefinitely on a cd-rom for future use by researchers. Since the investigator is embedded at the company any updates can come through verbal requests for more information may be made to him or you may contact the principal investigator at his Email.

Your signature on this form indicates that you have understood to your satisfaction the information regarding participation in the research project and agree to participate as a subject. In no way does this waive your legal rights nor release the investigators, sponsors, or involved institutions from their legal and professional responsibilities. **You are free to withdraw from the study at any time for any reason even while employed at PetroSleuth**. **You are also free to withdraw at any time from the study even if you are programming with a partner or creating unit test code. In addition the company will not be notified of your status within the study, this means that your status (participating or not) will not be given to the company.** Your continued participation should be as informed as your initial consent, so you should feel free to ask for clarification or new information throughout your participation. If you have further questions concerning matters related to this research, please contact:

Principal Investigator: Christopher Mann, 220-7140 mannc@cpsc.ucalgary.ca

Supervisor: Dr. Frank Maurer, 220-3531 maurer@cpsc.ucalgary.ca

If you have any issues or concerns about this project that are not related to the specifics of the research, you may also contact the Research Services Office at 220-3782 and ask for Mrs. Patricia Evans.

Participant's Signature

Date

Investigator and/or Delegate's Signature

Date

Witness' Signature (optional)

Date

A copy of this consent form has been given to you to keep for your records and reference.

APPENDIX H: CUSTOMER CONSENT FORM

Research Project Title: A case study into the Effectiveness of Agile Methods in an Industrial Setting
Investigator: Christopher Richard Mann
Co-Investigator and Supervisor: Dr. Frank Maurer

This consent form, a copy of which has been given to you, is only part of the process of informed consent. It should give you the basic idea of what the research is about and what your participation will involve. If you would like more detail about something mentioned here, or information not included here, you should feel free to ask. Please take the time to read this carefully and to understand any accompanying information.

The purpose of the study is to gather evidence of the effectiveness of agile methods in an industrial setting. Agile methods are new software methodologies which tend to be light weight in terms of documentation and try to focus on adapting quickly as possible to changing software requirements.

The study has introduced many agile techniques such as Scrum, pair programming, test-driven development.

If you volunteer as a participant in this study, you will be asked to fill out a questionnaire and possibly provide a follow up interview about the Scrum software development process which has been used since May 2004. The purpose of the questionnaire and possible interview is to get some feedback from your perspective as to how the software process is working for you. The questionnaire should take no longer then 30 minutes to complete. If a follow-up interview is needed and you agree to participate it, the interview should take no longer then 1 hour.

You are being asked to participate in the study because you are party of the customer group who works closely with the software developers. If you do not wish to participate in the study, tell the investigator and they will provide some alternatives where you would not participate in the tasks of the study but would allow your data to be used.

All information gathered as part of the study will be held in the strictest confidentiality. No personal or identifying information will be associated with the questionnaire or Interview. The questionnaires and transcribed interviews will be stored on a cd-rom for use by future researchers. If the interview is recorded by an audio tape recorder then the tapes will only be used to transcribe the interview into an electronic format. Once the transcription is complete the tapes will be erased. Since the investigator is embedded at the company any updates can come through verbal requests for more information may be made to him or you may contact the principal investigator at his Email.

Your signature on this form indicates that you have understood to your satisfaction the information regarding participation in the research project and agree to participate as a subject. In no way does this waive your legal rights nor release the investigators,

sponsors, or involved institutions from their legal and professional responsibilities. **You are free to withdraw from the study at any time for any reason. In addition the company will not be notified of your status within the study, this means that your status (participating or not) will not be given to the company.** Your continued participation should be as informed as your initial consent, so you should feel free to ask for clarification or new information throughout your participation. If you have further questions concerning matters related to this research, please contact:

Principal Investigator: Christopher Mann, 220-7140 mannc@cpsc.ucalgary.ca

Supervisor: Dr. Frank Maurer, 220-3531 maurer@cpsc.ucalgary.ca

If you have any issues or concerns about this project that are <u>not</u> related to the specifics of the research, you may also contact the Research Services Office at 220-3782 and ask for Mrs. Patricia Evans.

Participant's Signature

Date

Investigator and/or Delegate's Signature

Date

Witness' Signature (optional)

Date

A copy of this consent form has been given to you to keep for your records and reference.

APPENDIX I: ETHICS AND CO-AUTHOR APPROVAL

UNIVERSITY OF CALGARY

MEMO

CONJOINT FACULTIES RESEARCH ETHICS BOARD
c/o Research Services
Room 602 Earth Science
Telephone: (403) 220-3782
Fax: (403) 289 0693
Email: plevans@ucalgary.ca
Monday, January 26, 2004

To: **Christopher R. Mann**
Computer Science

From: Dr. Janice P. Dickin, Chair
Conjoint Faculties Research Ethics Board (CFREB)

Re: **Certification of Institutional Ethics Review:** A Case Study into the Effectiveness of Agile Methods in an Industrial Setting

The above named research protocol has been granted ethical approval by the Conjoint Faculties Research Ethics Board for the University of Calgary.

Enclosed are the original, and one copy, of a signed **Certification of Institutional Ethics Review**. Please make note of the conditions stated on the Certification. A copy has been sent to your supervisor as well as to the Chair of your Department/Faculty Research Ethics Committee. In the event the research is funded, you should notify the sponsor of the research and provide them with a copy for their records. The Conjoint Faculties Research Ethics Board will retain a copy of the clearance on your file.

Please note, an annual/progress/final report must be filed with the CFREB twelve months from the date on your ethics clearance. A form for this purpose has been created, and may be found on the "Ethics" website, http://www.ucalgary.ca/UofC/research/html/ethics/reports.html

In closing let me take this opportunity to wish you the best of luck in your research endeavor.

Sincerely,

Patricia Evans
Executive Secretary for:
Janice Dickin, Ph.D., LLB., Faculty of Communication and Culture and
Chair, Conjoint Faculties Research Ethics Board

Enclosures(2)
cc: Chair, Department/Faculty Research Ethics Committee
Supervisor: Dr. F. Maurer

UNIVERSITY OF CALGARY

CERTIFICATION OF INSTITUTIONAL ETHICS REVIEW

This is to certify that the Conjoint Faculties Research Ethics Board at the University of Calgary has examined the following research proposal and found the proposed research involving human subjects to be in accordance with University of Calgary Guidelines and the Tri-Council Policy Statement on *"Ethical Conduct in Research Using Human Subjects"*. This form and accompanying letter constitute the Certification of Institutional Ethics Review.

File no:	**CE101-3819**
Applicant(s):	**Christopher R. Mann**
Department:	**Computer Science**
Project Title:	**A Case Study into the Effectiveness of Agile Methods in an Industrial Setting**
Sponsor (if applicable):	

Restrictions:

This Certification is subject to the following conditions:

1. Approval is granted only for the project and purposes described in the application.
2. Any modifications to the authorized protocol must be submitted to the Chair, Conjoint Faculties Research Ethics Board for approval.
3. A progress report must be submitted 12 months from the date of this Certification, and should provide the expected completion date for the project.
4. Written notification must be sent to the Board when the project is complete or terminated.

Janice Dickin

Janice Dickin, Ph.D, LLB,
Chair
Conjoint Faculties Research Ethics Board

23 January 2004

Date:

Distribution: (1) Applicant, (2) Supervisor (if applicable), (3) Chair, Department/Faculty Research Ethics Committee, (4) Sponsor, (5) Conjoint Faculties Research Ethics Board (6) Research Services.

UNIVERSITY OF CALGARY

MEMO

Conjoint Faculties Research Ethics Board (CFREB)
Research Services Office
Main Floor, Energy Resources Research Building
Research Park
Telephone: (403) 220-3782
Fax: (403) 289-0693
Email: plevans@ucalgary.ca

To: Mr. Christopher Mann
Department of Computer Science

Date: February 24, 2005

From: Dr. Janice P. Dickin, Chair
Conjoint Faculties Research Ethics Board

Re: Approval of Modification for: A Case Study into the Effectiveness of Agile Methods in an Industrial Setting
Original Approval Date: 23 January 2004; extended to 2006/02/28
File No: 3819

The Certificate of Institutional Ethics Review issued on 23 January 2004 continues in force and extends to the modifications as set out in your email and attachments dated 16 February 2005. Your request to add a new questionnaire for the developers and broaden your recruitment pool to include customer opinion is approved as described.

You should attach a copy of the documentation you provided in order to request the modification, together with a copy of this memorandum, to the original Certification in your files.

Sincerely,

Janice Dickin

Janice Dickin, Ph.D., LLB., Professor
Faculty of Communication and Culture
Chair, Conjoint Faculties Research Ethics Board

cc: Chair, Department/Faculty Ethics Committee
Supervisor: Dr. F. Maurer

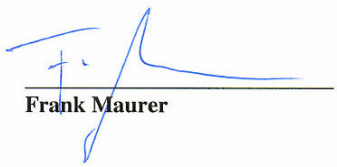
Co-Author Permission

June, 2005

University of Calgary
2500 University Drive N.W
Calgary, Alberta
T2N 1N4

I, Frank Maurer, give Christopher Mann permission to use co-authored work from our published paper, "A Case Study on the Impact of Scrum on Overtime and Customer Satisfaction" in this thesis.

Sincerely,



Frank Maurer