

Pair Programming Unearthed

M.Sc. Thesis by Hudson Gevaert
Supervisor: Dr. Vojislav B. Mišić

September 21, 2006

THE UNIVERSITY OF MANITOBA
FACULTY OF GRADUATE STUDIES

COPYRIGHT PERMISSION

Pair Programming Unearthed

BY

Hudson Gevaert

**A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of
Manitoba in partial fulfillment of the requirement of the degree
OF**

MASTER OF SCIENCE

Hudson Gevaert © 2006

Permission has been granted to the Library of the University of Manitoba to lend or sell copies of this thesis/practicum, to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film, and to University Microfilms Inc. to publish an abstract of this thesis/practicum.

This reproduction or copy of this thesis has been made available by authority of the copyright owner solely for the purpose of private study and research, and may only be reproduced and copied as permitted by copyright laws or with express written authorization from the copyright owner.

ABSTRACT

Extreme Programming (XP) is an emerging development methodology. To get a better understanding of XP, the practices which constitute this process need to be critically analyzed. In particular, the practice of pair programming is analyzed as it is both an interesting practice and one that provides a large area of research.

The goal of this thesis, therefore, is to analyze pair programming in an effort to determine how this practice relates to solo (traditional) programming, the personality characteristics of the developers and the XP practice of refactoring. By undertaking a thorough examination of pair programming, a deeper understanding of this important XP practice can be gained and a deeper understanding of XP can also be gained.

To accomplish this goal, two experiments were designed and implemented using fourth year computer science students at the University of Manitoba. The first experiment compared solo programming with pair programming while using personality characteristics, as determined by Eysenck Personality Questionnaire (EPQ-R), as factors. The second experiment compared solo programming and pair programming in terms of what kind of refactorings the subjects carried out while working individually as opposed to in pairs. The second experiment compared solo programming and pair programming in terms of what kind of refactorings the subjects carried out while working individually as opposed to in pairs.

These experiments were designed in an attempt to confirm or deny five hypotheses. These hypotheses are as follows:

1. H1

- (H1null) Pair programmers are not more efficient at software development than solo programmers.
- (H1a) Pair programmers are more efficient at software development than solo programmers.

2. H2

- (H2null) Personality characteristics play no role in how efficiently an individual develops code when working in a pair programming situation.
- (H2a) Personality characteristics do play a role in how efficiently an individual develops code when working in a pair programming situation.

3. H3

- (H3null) The individual efficiency of each partner in a pairing has no effect on how well the two individuals pair program.
- (H3a) Pair programming efficiency can be seen as a function of the solo programming efficiency of each of the individuals in the pairing.

4. H4

- (H4null) Individuals and pairs undertake refactorings at the same difficulty level.
- (H4a) Pair programmers undertake refactorings at higher difficulty levels than individuals.

5. H5

- (H5null) Pair programming satisfaction is unrelated to the personality characteristics and software development skill levels of the developers.
- (H5a) Pair programming satisfaction is linked to the personality characteristics of developers and their software development skill level.

The findings were such that the null hypothesis was demonstrated for H1 and H2, although some interesting statistics were still recorded. In particular, it is thought that with a larger sample size, the alternate H2 could be shown to be true. H1 yielded data which could suggest that pairs develop higher quality code while individuals develop code more quickly. In both cases, however, no statistically significant correlations were found.

For H3 and H5, the alternate hypotheses were adopted. In the case of H3, the higher the individual ability of the subjects, the higher the pair programming efficiency. For H5, it was clear that the more extroverted an individual tends to be, the more satisfied that individual tends to be with the pair programming experience.

The data for H3 was a little harder to analyze. While there was no statistical difference in the difficulty levels of refactorings individuals chose as opposed to pairs, there was a negative correlation in that a subject who undertook difficult refactorings individually would undertake less difficult refactorings when paired and vice versa.

In general, I believe this thesis was successful in broadening and deepening our understanding of the pair programming practice as the experiments yielded a large amount of interesting data in how the practice of pair programming relates to solo programming, personality characteristics, and refactoring.

CONTENTS

1. <i>Introduction</i>	1
2. <i>Literature Review</i>	7
2.1 Extreme Programming (XP)	8
2.2 Pair Programming	11
2.2.1 Pair Programming Experiments & Surveys	12
2.3 Refactoring	14
2.4 JUnit and Eclipse	14
2.5 Revised Eysenck Personality Questionnaire	15
3. <i>Problem Statement</i>	17
3.1 Hypothesis I	19
3.2 Hypothesis II	19
3.3 Hypothesis III	20
3.4 Hypothesis IV	21
3.5 Hypothesis V	22
4. <i>Proposed Solution</i>	23
4.1 Phase I	23
4.1.1 Experimental Environment	24
Subjects	25
4.1.2 Experimental Plan	25
4.1.3 Data Collection Protocol	29
4.2 Phase II	30
4.2.1 Realization of Experiment	30
4.2.2 Experimental Plan	30
4.2.3 Data Collection Protocol	35
5. <i>Analysis of Experimental Results</i>	37
5.1 Hypothesis I	37
5.2 Hypothesis II	41
5.3 Hypothesis III	43
5.4 Hypothesis IV	44
5.5 Hypothesis V	47

LIST OF FIGURES

4.1	Flow chart for pair programming experiment	25
4.2	Flow chart for refactoring experiment	31
5.1	T-test to compare the means of the solo scores and pair scores	39
5.2	Correlation of solo score versus pair score	40
5.3	Mean and standard deviation of pair efficiency score versus solo efficiency score	40
5.4	Mean and standard deviation of pair time and pair quality versus solo time and solo quality	41
5.5	Correlation of extroversion with pair and solo scores	42
5.6	Correlation of the log of the solo scores and neuroticism	43
5.7	A scatter plot graph showing the relationship between pair efficiency scores and the average solo efficiency scores of each partner	45
5.8	The ANOVA results from comparing the means of the pair programming efficiency scores with the average of the two programming solo efficiency scores	45
5.9	Correlation of refactoring weights for solo versus pair programmers	46
5.10	Correlation of pair programming satisfaction with personality characteristics	48
5.11	Correlation of pair programming satisfaction with solo efficiency scores	49
A.1	Class Hierarchy	59
A.2	Experiment Survey	61
A.3	List of Possible Refactorings	62
A.4	The data collected from phase I	68
A.5	The data collected from phase I	69
A.6	The data collected from phase II	70

LIST OF TABLES

A.1	Account Limits	60
A.2	Refactoring Weights	67

1. INTRODUCTION

The field of software engineering is slowly evolving. The methods and practices which were the mainstay of software development are beginning to give way to a newer, more dynamic method of software development.

Traditionally, software development has followed a sequential pattern of analysis, design, implementation, and testing [36]. The design stage of this type of methodology, widely known as the Waterfall method, is often labor intensive and tends to produce large documents which are supposed to represent all of the system requirements.

The purpose of this type of extensive requirements gathering is to mitigate the risk of any software engineering project. These risks refer to the chances of not delivering a workable project on time and under budget. After years of following this methodology, businesses have found out first hand that this type of software development process leads to problems when changes have to be made. The risks are therefore increased, rather than decreased.

In response to this situation, newer development methodologies, known as Agile Methodologies, have been designed which seek to eliminate the inherent overhead of the traditional system and provide a more 'light-weight' alternative. The term light-weight is used to describe these newer methodologies because they focus more on programming and less on documentation. The underlying theory of which is that a programmer knows how to program best, so let him or her program and not spend an

undue amount of time on creating cumbersome documentation.

At the forefront of these newer methodologies is Extreme Programming (XP), which was first advocated by Kent Beck [10]. Beck thought that if the cost of ongoing change can be mitigated then the waterfall method could be 'thrown in a blender' [9]. The result of this blending? Extreme Programming.

As with the traditional development methodologies, XP was also designed with the aim of reducing risk [10]. It just approaches this task in a substantially different manner attempting to take the programming best practices to the extreme. Some of the more essential practices include test-driven development, refactoring, small iterations, and pair programming.

The practices are as follows:

- The Planning Game
- Small releases
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming
- Collective code ownership
- Continuous integration
- 40-hour week
- On-site customer
- Coding standards

As XP is still a relatively young methodology, as well as being a complete departure from traditional methodologies, it has yet to gain widespread acceptance in the

business community. Such acceptance will not happen until the practices which constitute XP are thoroughly understood and managers are given the knowledge they need to adapt each practice for their own particular benefit. To date, there are few detailed reports that examine the relationship between XP practices and XP efficiency.

Pair programming, together with test-driven development, is often cited as one of the most important practices of XP [45] and yet little research has been done to understand *why* this practice is or is not effective. Pair programming, as can be inferred, is the act of two people programming at one computer doing the work that would customarily only be done by an individual. These two developers switch roles so each has a chance to control the keyboard.

Pair programming is thought to be responsible for a lot of the efficiency that XP professes to contain. Efficiency, in this case, is measured through productivity, expressed as the amount of code produced per time unit, and quality, expressed as the number of defects per some unit of code size. While pair programming may cause a drop in productivity compared to solo programming, the difference is expected to be small. The reason XP claims to be more efficient despite this productivity drop is because the practice is expected to produce a noticeable increase in quality with respect to solo programming. Thus, the productivity drop is offset and an overall increase in programming efficiency is attained.

There is evidence, however, that this does not happen in all situations, e.g., as noted in [24]:

Not everyone makes an ideal pair. It only works if the pair is reasonably evenly matched. If one person is quiet and does not contribute, their presence is wasted. Also, if a person is really disorganized and does not work in a cooperative way, the frustration can [disturb] the other participant!

The goal of this thesis, therefore, is to analyze pair programming to try and determine how it relates to solo programming, the personality characteristics of the developers and the other XP practice of refactoring. By undertaking a thorough examination of pair programming, beyond simply measuring whether it results in quicker code development, a more nuanced and useful understanding of this XP practice can be gained.

For example, it is often assumed that people who are more compatible work better together than those who are less compatible. In a pair programming environment, this compatibility takes on an even larger role as any two developers can be working in a very close workspace under pressure. Is it possible that certain personality characteristics of the developers play a role in how effectively they work in this type of pair programming situation?

Furthermore, in the XP methodology, each practice is meant to work in conjunction with the other XP practices. In an agile development environment where pair programming is practiced, other XP practices are likely also being utilized. Thus, to gain a better understanding of pair programming in an XP environment, an analysis of how pair programming is interconnected with other XP practices is desirable. For this thesis, the other XP practice that will be examined in conjunction with pair programming is refactoring.

The reason refactoring was chosen is because a practice was needed which, by its very nature, would be affected by pair programming. Practices like collective code ownership or small iterations would be tough to measure with regard to pair programming. Since refactoring is still development, however, its relationship with pair programming should be easier to measure. For example, does a developer make more complicated refactorings when paired as opposed to individually? Additionally, refactoring has not been the subject of as much research as some of the other XP practices, like test-driven development, making it a more novel area of research

If pair programming affects how code is developed, then there should be noticeable effects on how code is refactored between pairs and individuals. Together, these questions form the basis for this thesis. To attempt to answer these questions, I designed and carried out two experiments in which students at the University of Manitoba participated. The first experiment focused on the correlation between personality characteristics and pair programming efficiency. The second experiment focused on the relationship between pair programming and refactoring.

This analysis is essential if XP and its practices are to be embraced. Without evidence one way or the other as to the effectiveness of pair programming, it is unlikely many organizations will radically change the structure of their development environment. As noted by [32], ‘an empirical, quantitative investigation is beneficial for demonstrating XP’s efficacy.’ Only with the correct analysis and evidence can informed decisions be made.

In the next chapter I will review the Extreme Programming methodology with particular emphasis on the practices of pair programming and refactoring. In addition, I will give an overview of the resources I used in my experiments which include JUnit, Eclipse, and the Revised Eysenck Personality Questionnaire.

2. LITERATURE REVIEW

The first type of actual process development methodology is the Waterfall model. It is characterized by having four stages: Analysis, design, implementation, and testing. [9]. The goal of the Waterfall model is to move in a logical order from one stage to the next in a sequential manner. Once all of the system requirements are gathered, the system can be designed. Once the entire system is designed, the actual code can be written. Once the code is written, the system can be tested to see how well it works.

In the Waterfall model, these phases are undertaken sequentially in one long iteration. As the methodology evolved, it moved to more iterative variations such as the Spiral model or the Rational Unified Process (RUP). These slightly differing types of methodologies all have the same structure except they have more iterations of the same process. The classical life-cycles of these approaches require that a single activity be completed before one stage is complete and the next stage can begin [50].

This formality and rigid structure is the result of the prevailing view that the cost of changing a piece of software rises dramatically over time [9]. The logical response, therefore, was to try and ensure that every piece of code written is thought out well in advance of it ever being written.

The heavyweight, document-centric traditional methodologies just mentioned are no longer universally thought to be the best way to create software. Newer methodologies are being proposed which are the opposite of these classical development processes as they try to eliminate the formality and create a process which instead focuses

on being dynamic and evolutionary [15].

These methodologies, referred to as agile methodologies [6] because of their light-weight approach, arose as a response to the fact that even with all of the rules and formality in place, most software was still being delivered late, over budget and of questionable quality. Extreme Programming (XP) is one of these newer methodologies.

2.1 *Extreme Programming (XP)*

XP, as developed and advocated by Kent Beck [10] with the participation of Ward Cunningham and Ron Jeffries [48] is a development methodology which takes the supposed programming ‘best practices’ to the extreme. As noted by [50]:

Extreme Programming is the oldest and most popular agile software development methodology. As presented in the Agile Manifesto [2], the agile development values individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; and responding to change over following a plan.

To achieve the goal of small iterations and dynamic flexibility, XP is grounded in a philosophy defined by a set of values and principles which act as a guide when implementing the twelve main XP practices. The values specified by [10] originally and further explained by [27] and [49] are as follows:

- *Communication:* Most problems that arise in projects can be traced back to failures in communication [10]. It is for this reason that communication is the most overriding value [49]. Examples of communication include programmer-programmer communication, as with pair programming; programmer-manager

communication as with task estimation [27]; and customer-programmer communication that arises from user stories.

- *Simplicity*: Think about what the simplest way to complete your task is today and don't think about what the future may hold. It is better to do a task simply today and make changes later if need be [10]. By thinking of today and not of what might be useful in the future, it is easier to avoid code bloat and focus on the essential aspects of the system.
- *Feedback*: XP practices such as short releases, continuous integration, pair programming and test driven development are designed to encourage feedback as often as possible [27]. Feedback provides a sound foundation to move the project forward by working hand-in-hand with communication and simplicity to ensure everybody is aware of the projects status.
- *Courage*: For XP to be effective, the participants have to have the courage to do what is necessary to win. For example, if something does not work, throw it away and start again. If a flaw has to be fixed which compromises half of the already correctly running tests, then fix the flaw and make the tests work [10].

These values are the core of XP. When taken further, they can be distilled into four fundamental principles which are listed as follows:

- *Rapid Feedback*: When designing, coding, or testing the system, feedback should be immediate, measured in seconds, days, or weeks as opposed to months or years.
- *Assume Simplicity*: Best said by Kent Beck [10]: 'Treat every problem as if it can be solved with ridiculous simplicity'

- *Incremental Change*: Take baby steps toward solving a problem because making big changes all at once is ineffective.
- *Embracing Change*: Solve the most pressing problem first while leaving as many options available as possible.
- *Quality Work*: The developers have to enjoy their work, and good software will ensue [27].

These core values and principles result in a different way of looking at software development. Taken further, these principles can lead to a methodology based on a set of fundamental activities which in turn can lead to a set of practices which define this new type of software development process. These fundamental activities are coding, testing, listening, and designing. A description of the practices based on these fundamental activities is as follows [49]:

- *Planning*: Work with customers to determine the scope of the next iteration while developers provide the technical estimates.
- *Small Releases*: Get a prototype developed rapidly and then keep iterations small so that feedback can be given sooner.
- *Metaphor*: A way for the customer and developers to understand how the whole system works.
- *Simple Design*: Keep it simple by only designing the system for the features that need to be implemented today.
- *Testing*: Since feedback is essential with this methodology, unit tests are written by the developers while the customers write acceptance tests. In both cases, these tests act as a barometer for completion.

- *Refactoring*: Developers improve the design of the system when needed without changing the behavior of the system. Refactoring is an every day activity for the developers.
- *Pair Programming*: Two individuals work together at one computer when developing code.
- *Collective Ownership*: Any developer on the team can change any part of the system.
- *Continuous Integration*: Developers integrate and build the software system many times a day.
- *40-hour week* Developers are not to overwork themselves but are instead encouraged to work until they feel less productive.
- *On-site Customer*: The customer is part of the development team and is thus around at all times to answer questions and write acceptance tests.
- *Coding Standards*: As communication is a key value, coding standards are important to ensure consistency across the system.

2.2 Pair Programming

Pair programming is the practice of two developers working together at the same computer, creating the same code. In such a situation one developer acts as the 'driver' while the other developer acts as the 'navigator.' The driver is responsible for typing in the code while the navigator is responsible for observing the work of the driver and thinking more strategically and with a broader view of the whole application. Moreover, the navigator is also responsible for thinking tactically, whereby he or she can

check for syntax errors, typos, etc... [66]. Basically, the driver types while the navigator thinks.

Some of the purported benefits of pair programming include increased code quality and an increase in the transfer of knowledge. Furthermore, working in pairs, rather than individually, should provide inherent inspection capabilities and thus improve the quality of the product [51].

While the anecdotal evidence for pair programming is strong, there is also research that has been conducted on the effectiveness of pair programming as well as its effect on the actual developers. In the next section, a number of papers are reviewed which attempt to verify different aspects of pair programming through either experiments or surveys.

2.2.1 Pair Programming Experiments & Surveys

There have been experiments conducted on pair programming which attempt to understand pair compatibility and derive quantitative evidence from qualitative or anecdotal evidence [67]. Williams et. al [67] measured the ability of pairs to gel together and then surveyed the subjects about pair satisfaction after the experiment was conducted. The focus of the experiment was to find out which was better: pair programming or solo programming.

Lui et al. [34] undertook a similar experiment. The goal was to determine whether a pair outperformed two individuals. The method they used to measure productivity involved getting the subjects to solve algorithm-style aptitude tests. The results of the experiments were favorable to show that pairs have a higher level of productivity than two individuals due to XP practices such as simple design and refactoring.

Another type of experiment which has been conducted on pair programming focuses on the differing skill levels of the students in the study. Thomas et. al [58]

categorized their subjects into two groups; code-a-phobes and code warriors. The purpose of which was to determine which kind of student would pair better. Their research indicates that code warriors enjoyed the pair programming experience less than the less advanced programmers and that students of similar levels were more aptly suited as pairs than those with differing skill sets.

A more novel approach to understanding why pair programming is more effective than solo programming was conducted by Johnston and Johnson [26]. Basically, their hypothesis is that pair programming can lead to a more productive work environment in much the same way as a musician playing in an ensemble can increase the overall performance in a manner that is greater than simply adding his skill to the band. In other words, musicians grow from interacting and playing with other musicians. For a developer to grow in a similar way, that developer could interact closely with fellow developers in a way exemplified by pair programming.

Experiments have also been carried out which attempt to use personality tests to break students into groups and then propose roles for them based on those groupings [23]. Gifford and Henry use the Myers-Briggs Type Indicator [47] to determine group roles in their experiment. The programming groups they consider, however, are classroom programming teams and are unrelated to XP or pair programming.

The Myers-Briggs Type Indicator [47] has also been used by Katira et. al [30] to group students and then place them in a pair programming environment. The goal being to determine the compatibility of students with similar or different personality types and how well they work together. The results were such that while some evidence did exist, more research needs to be done.

2.3 Refactoring

Refactoring, as noted by Martin Fowler [19], is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. To the end user, what they see before code is refactored is the exact same as what they would see after code is refactored. The purpose of refactoring, therefore, is not to directly benefit the end user but to benefit the developers and the application itself.

In the context of XP's core values, refactoring is essential for promoting both simplicity and communication. If a design is flawed (too difficult) or the code is not communicative (redundant, overly complicated, etc..) then refactoring should take place. It is a general assumption that as the life cycle of a project increases, the time needed to implement changes increases exponentially [10]. In such a methodology refactoring is an essential practice as it ensures that the software system continues to be dynamic and evolutionary throughout the project life-cycle.

2.4 JUnit and Eclipse

JUnit is a standard set of open source java classes that can be used to test methods and classes. The purpose of using the JUnit libraries is that unit testing becomes easier and more fun, which encourages developers to write more tests before they write the code. This kind of Unit testing facilitates test-driven development which is itself a major practice of XP [33].

JUnit works by having the developers make assertions which are declarations of what the code should be doing. Those assertions are then compared with the actual output of the code to see if that code does indeed pass the unit tests.

In conjunction with IBM's Eclipse platform, JUnit can be used as a plugin giving

visual feedback to the developers as to the status of their code in terms of how many unit tests it passes. When a suite of JUnit tests is run in Eclipse, the subjects can easily see if they passed (indicated with a green check mark) or failed (indicated with a red x).

2.5 Revised Eysenck Personality Questionnaire

The revised Eysenck Personality Questionnaire (EPQ-R) was designed by H. J. Eysenck and S. B. G. Eysenck from the University of London to measure the dimensions of personality. The particular dimensions of personality gauged were specifically chosen because they have attained the highest level of consensus in the field [17]. The purpose of the EPQ-R, therefore, is to isolate these dimensions of personality in order to gain an insight into the characteristics of an individual.

The first dimension that the EPQ-R focuses on is the level of extroversion - introversion for a subject. An extrovert is considered somebody who is more outgoing and prefers to have people around and engage others. An introvert, on the other hand, is someone who appears reserved and distant and takes 'matters of everyday life with proper seriousness' [17].

The second dimension of personality the EPQ-R measures is referred to as neuroticism or stability/instability. A person with a typically high score on the neuroticism index tends to be a worrying individual who is anxious, moody, and frequently depressed. Someone who is low on the neuroticism index tends to be a much more calm individual who responds to stimuli slower and with less emotion.

The other main dimension of the EPQ-R is that of psychoticism. Less is known about the experimental correlates of psychoticism than the much more common extroversion - introversion and neuroticism dimensions. In general, psychoticism does not

imply a level of psychosis but simply refers to an underlying personality trait which, to some degree, is present in all persons.

These personality characteristics play a central role in the choices people make. As noted by [17],

personality differences relate closely to both vocational guidance and occupational selection...It would be difficult to think of any area of human endeavor in which individual differences in personality are unimportant and where some knowledge of a person's degree of extroversion, neuroticism or tough-mindedness would not be of great help.

Thus, the EPQ-R is intended to be used as a tool for helping to explain and predict human behaviour. The test itself contains one hundred yes/no questions. Depending on the responses to these questions, a deeper understanding of the factors driving human conduct and interaction can be gained.

3. PROBLEM STATEMENT

The purpose of XP is to take the software development best practices to the extreme. As noted by Beck [10], if a practice is assumed to be a good development practice, then it should be a good development practice all of the time.

If code that is reviewed often is considered better than un-reviewed code, then there should be somebody constantly evaluating the code as it is created. This idea is the basis of pair programming as two people working together on the same computer will be part of a constant peer review process. Code reviews therefore take place in small pieces as the code is written instead of at a later time when large amounts of code would need to be reviewed.

Furthermore, if it is accepted that the cost of making design changes increases with the lifespan of a project, then design changes should be made as early as possible. This is the basis of refactoring.

Although Extreme Programming is still a relatively new methodology, its proponents are ardently trying to get it accepted into the business community. Such acceptance would be greatly accelerated if the practices which constitute XP are thoroughly understood and managers are given the knowledge they need to manipulate each practice for their own particular benefit. Pair programming, in particular, needs to be analyzed in detail. To date, there are few detailed reports that examine the relationships between pair programming and solo programming, the personality characteristics of the developers, and other XP practices.

Understanding these relationships is essential if a broad analysis of the effectiveness of XP is to be undertaken. The problem is that there is not enough information available explaining why some people work better together in a pair programming situation than others. A main goal of this thesis, therefore, is to try and determine if there is a link between certain personality characteristics and pair programming success.

The three personality characteristics that will be discussed in this thesis are extroversion, neuroticism, and psychoticism. These characteristics are based on the Eysenck Personality Questionnaire [17] and are described in more detail in the literature review.

Furthermore, none of the practices of XP operate in a vacuum, but are interconnected in a complex relationship. To understand pair programming fully not only the practice itself has to be analyzed, but how that practice influences other XP practices like refactoring also has to be examined. If individuals tend to refactor differently than pairs, then there is no point in examining refactoring with regards to XP without also determining whether or not the act of pair programming impacts upon what kind of refactoring decisions are made by pairs as opposed to individuals.

Refactoring and pair programming are two of the core practices of XP [10]. Understanding how these practices interrelate is essential to furthering our assessment of XP as a development methodology. While studies exist that analyze both refactoring [29; 37; 41] and pair programming [5; 30; 32; 36; 39; 43; 44; 51; 59; 67; 64], there is a lack of empirical investigation into how these particular practices relate to each other.

Kent Beck, the originator of the XP methodology, advocates that XP practices are meant to work with each other as both weight and counterbalance. Experiments and studies that attempt to analyze XP, therefore, should also take into account how multiple practices function together. To further this end this thesis offers the following five hypotheses to be validated or refuted.

3.1 Hypothesis I

One of the central concepts of extreme programming is that pair programming is more efficient than solo programming. That does not mean that a pair programs faster than an individual, but when quality is factored in the overall efficiency of development favors pair programming. For instance, while an individual may program faster than a pair, the individual will have more defects in his or her code and design choices than the pair. Thus, after taking into account the time needed to fix the defective code the advantage would fall to the pair programming method.

The first hypothesis of this thesis is that there is a significant difference between the overall efficiency of individuals as opposed to pairs when developing software in terms of both speed and quality. The null and alternate hypothesis are formally defined as follows:

1. (H1null) Pair programmers are not more efficient at software development than solo programmers.
2. (H1a) Pair programmers are more efficient at software development than solo programmers.

3.2 Hypothesis II

Development teams are made up of individual developers with individual personality characteristics working together to create a piece of software. This hypothesis examines how these characteristics play a part in how well an individual works in a pair situation as opposed to individually. If developers are working individually, then their individual personality characteristics play only a small role in how efficient they are at software development. If two developers are working in a pair environment, personal-

ity characteristics play a larger role in their efficiency as the rate of interaction between the developers is increased greatly.

In particular, an individual is suspected to work better in a pair development situation if he or she is more extroverted while someone who is better at programming solo is less extroverted. Neuroticism and psychoticism are also suspected to play a role but what that role is uncertain. The null and alternate hypotheses are as follows:

1. (H2null) Personality characteristics play no role in how efficiently an individual develops code when working in a pair programming situation.
2. (H2a) Personality characteristics do play a role in how efficiently an individual develops code when working in a pair programming situation.

3.3 Hypothesis III

The focus for H2 is on how an individual's personality characteristics affect his or her ability to pair program. The next step is to try and determine if the individual skill level of each of the pair programmers affects how well they program in pairs.

Specifically, the question is whether pair programming efficiency can be seen as a function or relation of how well the two pair programmers worked individually.

1. (H3null) The individual efficiency of each partner in a pairing has no affect on how well the two individuals pair program.
2. (H3a) Pair programming efficiency can be seen as a function of the solo programming efficiency of each of the individuals in the pairing.

3.4 Hypothesis IV

The main purpose of this thesis is to understand the XP practice of pair programming. In other words, to unearth the hidden factors behind pair programming success or failure. As noted, however, no XP practice works in a vacuum. Not only is it necessary to understand the details of why individuals work better or worse in a pair programming situation, but an understanding of how this pair relationship relates to other XP practices is also needed.

For this thesis, the other XP practice that will be examined in conjunction with pair programming is refactoring. For pair programming, an analysis is conducted of pair efficiency based on quality of work and speed of development. Refactoring can be analyzed along these same lines by determining what kind of refactorings are undertaken by pairs as opposed to individuals. For this reason refactoring was chosen.

An analysis is needed to try and determine whether the act of pair programming causes a different kind of refactoring to take place than that undertaken by individuals. Since refactoring affects the whole application, it would be useful to know whether or not pairs tend to take on more difficult or abstract refactorings as opposed to individuals. The null and alternate hypothesis are as follows:

1. (H4null) Individuals and pairs undertake refactorings at the same difficulty level.
2. (H4a) Pair programmers undertake refactorings at higher difficulty levels than individuals.

3.5 Hypothesis V

It is a common thought that the best software development is undertaken by developers who enjoy the work they do. Since developer resistance is a major obstacle to the practice of pair programming, an attempt to discern in what situations pair programming would meet with the most minimal resistance is desirable.

Is there a certain personality characteristic that leads to an individual enjoying the pair programming experience more than solo development? Since pair programming is a very interactive experience between two people, perhaps individuals with higher levels of extroversion will enjoy the pair programming experience more. Furthermore, perhaps those who tend to develop better by themselves enjoy the pair programming experience less. The null and alternate hypothesis are as follows:

1. (H5null) Pair programming satisfaction is unrelated to the personality characteristics and software development skill levels of the developers.
2. (H5a) Pair programming satisfaction is linked to the personality characteristics of developers and their software development skill level.

4. PROPOSED SOLUTION

The goal of this thesis is to analyze the XP practice of pair programming in order to determine what effect personality characteristics have on pair effectiveness and how this relates to the other XP practice of refactoring. Thus, I devised a pair of experiments with the intent of corroborating or refuting my hypotheses. The first experiment focused on the personality characteristics of developers and the relationship those traits have with pair or solo effectiveness.

The second experiment was designed to analyze the relationship between pair programming and refactoring. In particular, I wanted to see if the act of pair programming resulted in different refactoring decisions being made by the pairs as opposed to those decisions made by the individuals.

The practices of XP cannot be analyzed in isolation. To understand one practice (like pair programming) we have to examine how the act of working in pairs affects the way other XP practices are performed. In this case, refactoring is the other practice.

The first experiment, which from now on will be referred to as Phase I, will try to confirm or deny H1, H2, H3 and H5. The second experiment, Phase II, will focus on proving or disproving H4. The experimental design for each phase is described below.

4.1 Phase I

This phase is intended to examine the manner in which the personality characteristics and programming skills of individual programmers influence the productivity and

quality of software development in a pair programming environment compared to a solo programming environment. As noted by [4], if pair programming is to be truly understood, more behavioral experiments are needed that involve students or professionals. As this experiment was conducted at the University of Manitoba, a practical decision was made to gain the participation of students enrolled here.

For each of the subjects taking part in the experiment, there were four distinct pieces of information we needed to ascertain:

1. The personality characteristics of the subjects as determined by the EPQ-R
2. The relative skill level of each subject as compared with the skill level of the other subjects taking part in the experiment.
3. How well each of these individuals worked in a pair situation as opposed to individually.
4. The opinions of the subjects regarding the experiment and their pair programming satisfaction.

4.1.1 Experimental Environment

This experiment was conducted in a lab with limited seating. It was therefore divided into two parts. Each part was conducted in the exact same manner but on separate days. Subjects on the first day simply had the letter A next to their id while those on the second day had the letter B next to their ID.

The lab computers each had a copy of the IBM Eclipse platform and a preloaded workspace which contained the two problems pre-loaded. The Eclipse platform also had the JUnit plugin installed which allowed me to create the associated suite of JUnit test cases for the application.

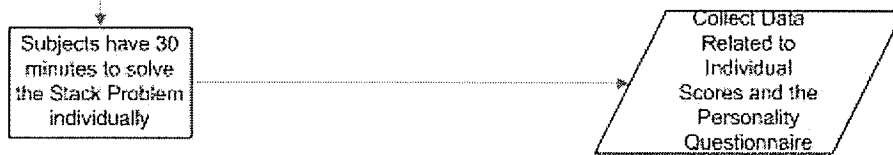
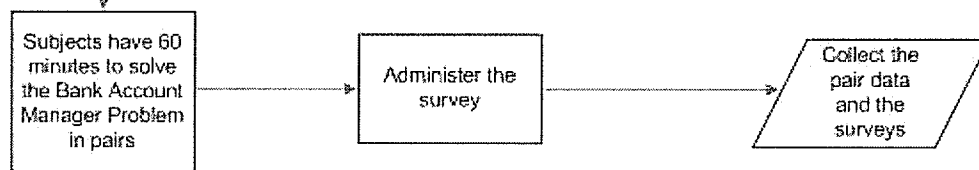
Part 1: 30 minutes**Part 2: 45 minutes****Part 3: 75 minutes**

Fig. 4.1: Flow chart for pair programming experiment

Subjects

In this experiment we had 28 fourth year computer science students from the University of Manitoba take part. A sample size of 28 was used because that was the number of students that were available. Future work would include similar experiments with a larger sample size to attempt to analyze the more subtle correlations.

4.1.2 Experimental Plan

Figure 4.1 shows an overview of the experimental process for Phase I.

I began by giving an explanation of the methodologies and concepts used in the experiment. The purpose was to educate the subjects on the context of the experiment

so they would be somewhat knowledgeable in the concepts this experiment addresses. Since the subjects had different levels of familiarity with XP, it was desirable to ensure that each subject was at least familiar with the basic principles behind XP in general, and pair programming in particular. In addition, each subject was given three numbered sticky tags which were used for keeping related materials together.

After this brief overview of the experiment, I administered the Eysenck Personality Questionnaire (EPQ-R) [17] to all of the subjects. The subjects were given fifteen minutes to answer the one hundred yes or no questions of the EPQ-R.

Once the EPQ-R part of the experiment was complete the subjects had to solve two programming problems that will from now on be referred to as the stack problem and the bank account manager problem. How long it took a subject or pair of subjects to solve these problems and the number of defects were the determining factors in the efficiency of each subject.

While timing the subjects was straightforward, measuring quality was a little more difficult. To handle this problem, a prepared suite of JUnit test cases accompanied each programming task as a way to gain a quantitative assessment of each subject's performance in terms of code defects. The more tests the code passed, the higher the skill level of the subject and the higher the quality of the program.

JUnit was used for the purpose of measuring defects. When a suite of JUnit tests is run in Eclipse, the subjects can easily see if they passed or failed. Thus, for the stack and bank account manager problem, a subject's efficiency was based on how many JUnit test cases he or she could get to pass combined with how long it took them to do so.

The stack problem was designed to test the relative skill level of individual subjects by giving them a programming task and evaluating the productivity and quality of their work. To determine the relative skill level of each subject, a simple problem that all

of the subjects would be familiar with but still require them to expend some effort on solving was required. For this part a simple stack class was used which the subjects had to implement in half an hour.

Thus, the subjects were required to solve the stack problem as quickly as possible in such a way that the code passed all of the quality tests. The subjects were provided with the class definition for a stack called `MyStack` written in Java. The written question given to the subjects is listed in Appendix A.1.

There were eight JUnit tests associated with this problem and a time limit of 30 minutes. Initially, all of the JUnit test cases failed, so the subjects had to succeed in getting all of the test cases to pass in as little time as possible without altering the test cases in any way. The test cases were designed to ensure the subjects would check for null cases and that all of the data got added and removed from the stack in a consistent way.

At this point the subjects were given a fifteen minute break to have some refreshments and relax. I took the opportunity to record the number of JUnit tests passed by each subject and ensure that none had begun early on the second problem. After the break, the subjects were randomly paired to begin work on the second problem.

While the first problem was designed to assess the general skill level of each subject individually, the second problem was designed to get an idea of how successful pairs were at solving a programming problem.

As noted, the subjects were asked to create part of an account management system while working in pairs. The subject pairings were randomly assigned and therefore unrelated to the scores from the first problem or the personality characteristics as determined by the EPQ-R.

Three types of bank accounts had to be created: a checking account, a savings account, and a credit card account. For the credit card account, there were three types

of credit cards: Visa, AMEX, and MC. Each account had a customer associated with it. In addition, money could be withdrawn from each account, deposited into each account, or transferred between two accounts.

I provided the subjects with a table of interest rates. Also, the credit cards had an associated variable `limit` which prevents the card from going over a certain limit. I provided the subjects with a table of `limits` for each type of credit card.

The question given to the subjects can be seen in Appendix A.2. As with the stack problem there was a suite of JUnit tests associated with this problem. For this problem nine JUnit tests were used because there was more code that needed to be tested.

Since the amount of work required to solve this problem was more than that for the stack problem, the subjects were given a full hour time to find a solution. I provided them with the definition for each class and some of the method definitions with the intended purpose of the subjects creating an application that fulfilled all of the criteria as given to them in the problem statement.

The class and method definitions were provided so that the subjects would have an idea of the class hierarchy. Ideally, this problem would be at the same difficulty level as the stack problem so all that was required of the subjects was for them to put the appropriate code in the appropriate places.

After the time expired on the second programming problem, a survey was distributed to determine whether the subjects enjoyed the pair programming experience, what they thought were the pros and cons of pair programming and whether they would want to pair program again. In particular, the survey was designed to gain insight into how each of the subjects appreciated pair programming and the experiment. Appendix A.3 contains all of the questions on the post-experiment survey.

4.1.3 Data Collection Protocol

To capture the results of each subjects performance, I used a spreadsheet containing only the id's of each subject. These ID's were unrelated to their names as the personal information of the subjects in this experiment was treated with all possible confidentiality. This confidentiality was due to the fact that each subject was asked to answer an in depth personality questionnaire. To further ensure that every effort was made to conduct a responsible experiment, I had my experimental method and plan approved by the University of Manitoba's Research Ethics Board (REB).

To ensure confidentiality, each subject was given a set of sticky tags with a number on them. One tag went on each subject's EPQ-R, one tag went on the survey, and one tag was associated with his or her computer. In this manner there was no way for a subject's name to be identified with the results from the questionnaire, the survey, or the programming problems. Thus, there was a level of disconnection which prevented even the experimenter from knowing which subject was responsible for which result.

To collect the programming information I waited until the allotted time had expired and then went to each computer and downloaded all of the results for later analysis. I also ensured that the subjects informed me as soon as they solved a problem such that all of the JUnit test cases passed. This information was recorded in my spreadsheet to keep a record of each subject's time and tests passed.

The same procedure was used for both the stack and bank account manager problem. After the experiment, the results of the personality questionnaire and the survey were tabulated and recorded. Thus, at the conclusion of the experiment, for each subject it was known what personality characteristics they possessed and to what degree, how long it took them to solve each problem, how many JUnit tests they passed for each problem, and what their answers were for the survey without having to relate this information to the actual names of the subjects.

4.2 Phase II

To understand the relationship between pair programming and refactoring, I designed an experiment (Phase II) that would identify which kind of refactorings would be undertaken by individuals as opposed to pairs. The strategy I incorporated was to provide a large enough problem domain such that the subjects would have a multitude of refactorings to choose from. To run an experiment on refactoring I created an already functional program in consultation with my colleague Michael W. Rennie (MSc) which contained as many *bad smells* as possible.

4.2.1 Realization of Experiment

I had 22 subjects take part in this experiment working both individually and in a pair situation. A sample size of 22 is not a lot of subjects but was the most that could be utilized with the limited funds and time available. Future work would include a similar experiment with a larger sample size to attempt to analyze the more subtle correlations.

The same lab as for Phase I was used as each computer had the IBM Eclipse platform and a preloaded workspace which contained the problem application already loaded. The Eclipse platform also had the JUnit plugin installed which allowed me to create an associated suite of JUnit test cases for the application. As each subject gained experience with Eclipse and JUnit from Phase I, there was no need to give them another introduction to this environment.

4.2.2 Experimental Plan

Figure 4.2 shows an overview of the experimental process for Phase II.

I began by giving an explanation of refactoring. It was important to impart to the subjects the importance of their feedback with regards to the refactorings they

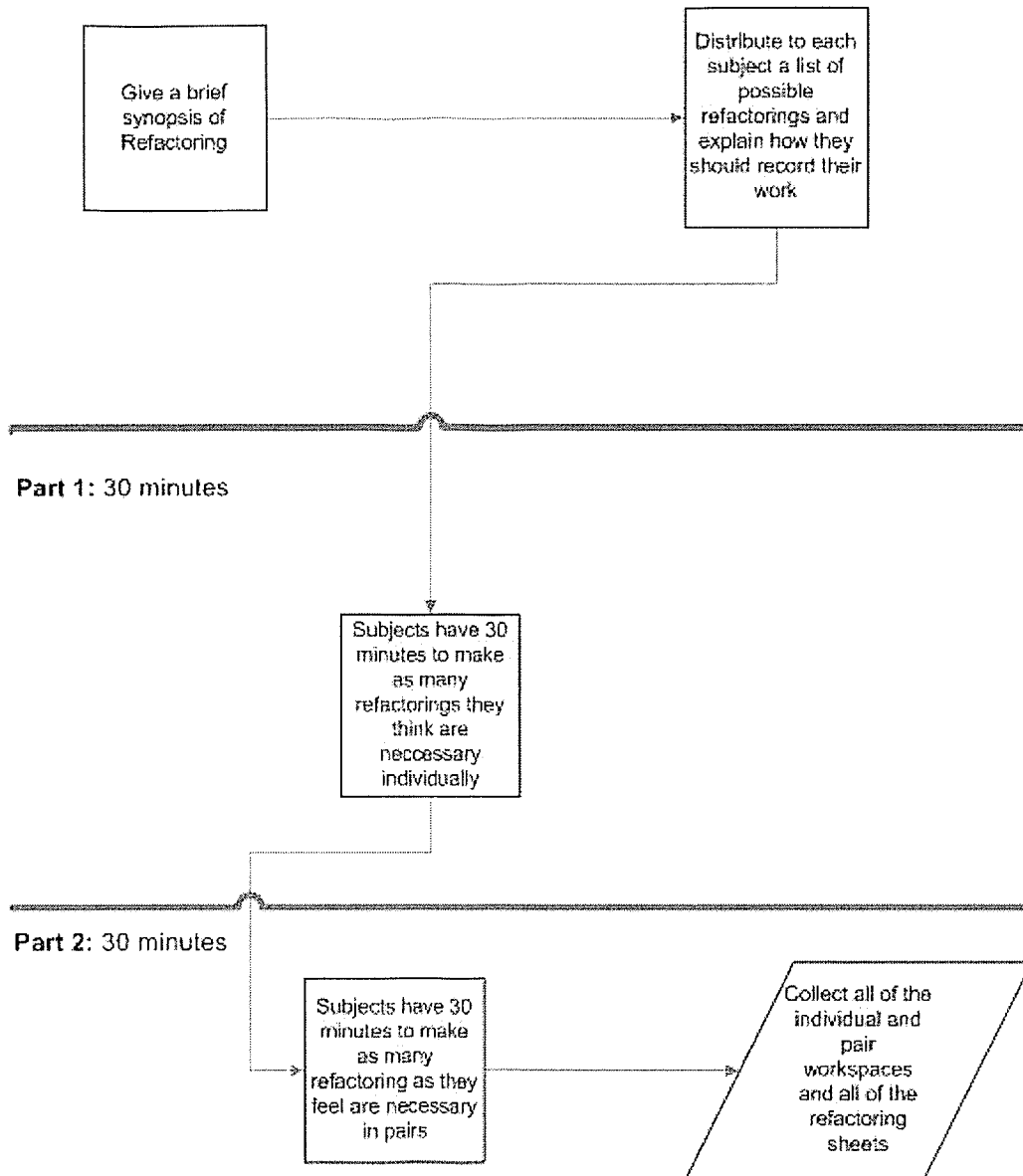
Introduction: 15 minutes

Fig. 4.2: Flow chart for refactoring experiment

performed. Since the JUnit tests were supposed to pass both before and after the refactorings, it was dependent upon the subjects to indicate where the refactorings took place and the order in which they occurred.

As with Phase I, this experiment required the subjects to work both individually

and in pairs. In the first part of the experiment the subjects worked individually on the assigned problem, henceforth referred to as the car problem, and attempted to make as many refactorings as they felt were necessary. For this part the subjects were given half an hour to commit as many refactorings as possible such that the associated JUnit tests continued to pass.

In the second part of the experiment, the subjects were randomly paired and again asked to make as many refactorings as possible. Since there were enough potential refactorings to ensure that the subjects would always have something to improve, there was no need to create two separate problem domains. Furthermore, since I did not want each pair rehashing the same refactorings they did individually, they were required to carry on from at least one of the partner's Eclipse workspaces from the first half of the experiment.

To calculate how many refactorings were completed by the subjects, a table was provided for them to fill out each time they completed what they considered to be a refactoring. This table outlined what piece of code was refactored, how it was refactored, and what was gained by the refactoring (i.e. fewer lines of code, less redundant code, increased efficiency, etc.).

In addition, a list of possible refactorings, as compiled by Fowler [19] originally and synthesized by Wake [62], was provided to the subjects to give them an idea of what possible refactorings to look for. This extra information provided the subjects with an idea of what constituted a *bad smell* without giving them any specific information on how many *bad smells* were in the program or how often they occurred.

The problem given to the subjects was designed to include as many refactoring availabilities as possible. The problem domain was made large enough so that the subjects would not run out of refactorings in the allotted time. With a large problem domain in terms of potential refactorings, it was possible to gather a larger breadth of

data on the refactoring habits of individuals as opposed to pairs.

The car problem was designed as a fully functional Java application with an associated set of JUnit tests that were designed to pass in the applications initial state. Appendix A.5 is the question as it was given to the subjects. In refactoring the application, it was up to the subjects to ensure that after each refactoring the JUnit test cases still passed. A layer of abstraction, in the form of a gateway class, was created so that the subjects could change how methods and classes were called without changing the actual JUnit test cases.

I used the fairly straightforward example of a doubly linked list being used to keep track of different car models. The doubly linked list was sorted in ascending order based on the car models. The doubly linked list had functionality implemented that allowed a car to be added, deleted, or found based on the model number.

To make such a large problem with as many bad smells as possible, a number of classes were defined and implemented which have no effective design aspect, are inefficient, and yet still work. To determine the workability of the design, a test suite using JUnit was also developed to work in conjunction with the IBM Eclipse platform.

All of the JUnit tests succeeded before the problem was presented to the subjects. Only the refactorings done by the subjects that resulted in all of the test cases still passing were recorded because the purpose of refactoring is to change the design aspects of a system without changing how the system functions.

There were two types of car that were used in the program: Luxury and Economy. Each car had a model, a make, engine specifications, etc. Each type of car also had some type specific information such as whether it came with leather seats or cup holders.

The application, by design, was not a very well designed application. It contained plenty of potential refactorings in the form of useless or redundant code, non-

descriptive variable names and methods and variables placed in inappropriate places. The important aspect was that the program still worked the same both before and after the subjects had their chance to make a more efficient, well designed, communicative application. It was up to the subjects to make whichever refactorings they wished in the allotted amount of time such that all of the JUnit test cases still passed.

In the course of refactoring, the subjects deleted and added classes and in effect altered the entire design of the application. Since the JUnit test cases still had to pass after all of these changes, a gateway class was created which allowed the subjects to change the test parameters. The subjects were allowed to change this class in order to ensure that the JUnit tests would run but they were not allowed to alter the underlying JUnit test suite. This layer of abstraction is what allowed the same JUnit test cases to function on both the original and refactored application.

The problem given to the subjects was already solved (in terms that it already functioned fully) so a lot of the information gained was dependent on how well the subjects communicated their intentions. With an almost unlimited amount of potential refactorings it was up to the subjects to indicate what changes they made and in what order. Moreover, as I wanted to compile statistics on each running version of the application, the subjects were required to export the whole application after each refactoring.

To help facilitate this extra work, the subjects were provided with two documents. The first document contained a list of all the potential refactorings as compiled by [62] and is shown in Appendix A.4. The second document was simply a numbered list with space for the subjects to write what refactorings they had undertaken. Ideally, the subjects would correlate whatever change they made to one of the given refactorings, although this step was not essential.

Once the pairs ran out of time, the data from the computers was collected along with the refactoring sheets which recorded what refactorings were undertaken. The

data from the computers consisted of each saved Eclipse workspace after each refactoring by both the individuals and the pairs.

4.2.3 Data Collection Protocol

The subjects were encouraged to make as many refactorings as they felt were necessary. I compiled this data into a spreadsheet which recorded what type of refactorings the subjects made and how many times they made those type of refactorings. I recorded each refactoring by name up to what turned out to be a maximum of 25 different types of refactorings. For a complete record of these refactorings, consult Appendix A.6.

Furthermore, I divided the refactorings into three categories. Each refactoring was determined to be at the method level, class level, or application level. If a refactoring only impacted a method, it was considered the method level. If a class variable was changed or code moved between methods, I considered that refactoring to be at the class level. If a class was deleted or added, or methods or variables moved between classes, I considered that refactoring to be at the application level.

The second way of dividing the refactoring into categories was to assign them weights. In consultation with another Java Instructor at the University of Manitoba, John Braico, we gave each refactoring a relative weight of 1, 2 or 3 depending on the level of difficulty of each refactoring. These weights can be seen in Appendix A.7

5. ANALYSIS OF EXPERIMENTAL RESULTS

Overall, the experiment described in the previous chapter yielded some interesting results. Appendix A.8 displays the data collected from Phase I and Phase II. The following sections will discuss how the results from the experiments tend to confirm or deny the alternate hypothesis for each of the five hypothesis described in Chapter 3.

5.1 *Hypothesis I*

1. (H1null) Pair programmers are not more efficient at software development than solo programmers.
2. (H1a) Pair programmers are more efficient at software development than solo programmers.

At the conclusion of Phase I of the experimental suite, the subjects had worked on two programming problems. The first problem they did individually while they worked in tandem on the second problem. Since the problems were different, both in terms of how much time the subjects were given to complete the problems and how many test cases were created for each problem, a method for comparing the results of each part of Phase I had to be devised.

To accomplish this synthesis, formulas for the pair programming and individual results were created which would give an efficiency score for each subject taking part in the experiment. Efficiency being a combination of how productivity (speed) related

to quality (tests passed). These efficiency scores will henceforth be referred to as solo score and pair score. The formulas are thusly listed in Figure 5.1 below for the solo programming part of the experiment and Figure 5.2 for the pair programming part.

$$SoloEfficiency = (TSt/St) * (Sp/TSp) \quad (5.1)$$

$$PairEfficiency = (TPt/Pt) * (Pp/TPp) \quad (5.2)$$

where:

TSt The total time given to the subjects to solve the solo problem which was equal to 30 minutes.

St The amount of time it took for the subjects to pass all of the tests for the solo problem. This value would be equal to TSt if an individual was unable to pass all of the tests in the allotted time.

Sp The number of test cases passed during the solo programming part of the experiment.

TSp The total number of test cases that were provided for the subjects to pass while working individually. This value is equal to 8.

TPt The total time given to the subjects to solve the pair problem which was equal to 60 minutes.

Pt The amount of time it took for the subjects to pass all of the tests for the pair problem. This value would be equal to Tp if a pair was unable to pass all of the tests in the allotted time.

One-Sample Test						
	Test Value = 0					
	t	df	Sig. (2-tailed)	Mean Difference	95% Confidence Interval of the Difference	
					Lower	Upper
SoloScore	6.086	27	.000	.95824	.6352	1.2813
PairScore	10.961	27	.000	1.14143	.9278	1.3551

Fig. 5.1: T-test to compare the means of the solo scores and pair scores

Pp The number of test cases passed by the pair programmers in the second half of phase I.

TPp The total number of test cases that were provided for the subjects to pass while working in pairs. This value is equal to 9.

Figure 5.1 shows the results of a t-test on the values for solo score and pair score. There was no level of statistical significance found in the difference between the two means. The next test, the results of which are shown in Figure 5.2, attempts to find a correlation between any of the individual scoring metrics and the pair scoring metrics. Again, no statistical correlation was found.

A more thorough investigation, however, does result in some more interesting findings. Figure 5.3 shows the mean values for both the solo efficiency scores and the pair efficiency scores. The pair scores in these results are, on average, higher than the solo scores.

The solo and pair efficiency scores are, as mentioned, a result of how long it took the subjects to complete the tests and how many tests the subjects correctly passed. Figure 5.4 shows the means for how long it took individuals and pairs to finish the tests and how many test they passed. While not statistically significant, it is interesting how the mean value for solo time is less than that for pair time, indicating that on average

Correlations							
		SoloScore	SoloQuality	SoloTime	PairScore	PairQuality	PairTime
SoloScore	Pearson Correlation	1	.880**	-.925**	.181	.173	-.165
	Sig. (2-tailed)		.000	.000	.358	.378	.402
	N	28	28	28	28	28	28
SoloQuality	Pearson Correlation	.880**	1	-.643**	.161	.093	-.181
	Sig. (2-tailed)	.000		.000	.414	.639	.358
	N	28	28	28	28	28	28
SoloTime	Pearson Correlation	-.925**	-.643**	1	-.138	-.147	.116
	Sig. (2-tailed)	.000	.000		.492	.457	.558
	N	28	28	28	28	28	28
PairScore	Pearson Correlation	.181	.161	-.138	1	.838**	-.949**
	Sig. (2-tailed)	.358	.414	.492		.000	.000
	N	28	28	28	28	28	28
PairQuality	Pearson Correlation	.173	.093	-.147	.838**	1	-.659**
	Sig. (2-tailed)	.378	.639	.457	.000		.000
	N	28	28	28	28	28	28
PairTime	Pearson Correlation	-.165	-.181	.116	-.949**	-.659**	1
	Sig. (2-tailed)	.402	.358	.558	.000	.000	
	N	28	28	28	28	28	28

** Correlation is significant at the 0.01 level (2-tailed).

Fig. 5.2: Correlation of solo score versus pair score

One-Sample Statistics

	N	Mean	Std. Deviation	Std. Error Mean
SoloScore	28	.9582	.83318	.15746
PairScore	28	1.1414	.55102	.10413

Fig. 5.3: Mean and standard deviation of pair efficiency score versus solo efficiency score

solo programmers finished quicker. Since the efficiency scores are a combination of speed and quality, then the quality average must be higher for pairs than for individuals. And this is indeed the case.

Thus, while the alternate hypothesis can not be proved to a statistically significant level, the trend indicates that pair programming results in slightly more efficient code production. A possible conclusion, therefore, is that while solo developers tend to develop faster, the pair developers create code of a high enough caliber that the pair efficiency scores are still higher, on average, than the solo efficiency scores.

One-Sample Statistics

	N	Mean	Std. Deviation	Std. Error Mean
SoloTime	28	.8333	.21525	.04068
PairTime	28	.8155	.19186	.03626
SoloQuality	28	.6384	.45177	.08538
PairQuality	28	.8333	.25210	.04764

Fig. 5.4: Mean and standard deviation of pair time and pair quality versus solo time and solo quality

5.2 Hypothesis II

1. (H2null) Personality characteristics play no role in how efficiently an individual develops code when working in a pair programming situation.
2. (H2a) Personality characteristics do play a role in how efficiently an individual develops code when working in a pair programming situation.

An important aspect of this experiment is to determine how well each subject does in relation to each other. Some subjects did poorly compared with their peers during the skills exercise while some did very well and most fell in the middle. This is representative of all development environments as it can be expected that some developers are very experienced while some are poor, and most usually fall in the middle.

One of the main factors of the EPQ-R is that of extroversion - introversion. This hypothesis supposes that individuals who tend to be more extroverted will get a better score at pair programming than those who tend to be more introverted. Figure 5.5 shows the results of a single-tailed correlation analysis between extroversion and pair/solo score.

Correlations

		PairScore	SoloScore	Ext
PairScore	Pearson Correlation	1	.181	-.034
	Sig. (1-tailed)	.	.179	.432
	N	28	28	28
SoloScore	Pearson Correlation	.181	1	-.134
	Sig. (1-tailed)	.179	.	.248
	N	28	28	28
Ext	Pearson Correlation	-.034	-.134	1
	Sig. (1-tailed)	.432	.248	.
	N	28	28	28

Fig. 5.5: Correlation of extroversion with pair and solo scores

These results indicate that there is no statistically significant correlation between how extroverted a subject is and how well they perform at pair programming and solo programming. The indication, therefore, is that regardless of how introverted a person is, he or she is just as capable of pair programming at an effective level as someone who is more extroverted. Thus, for Hypothesis II the null hypothesis will have to be accepted.

The scope of the Hypothesis II, however, does lead to some interesting findings with regard to the personality characteristics of neuroticism and solo programming effectiveness. For example, Figure 5.6 shows that when the natural logarithm of the subject's solo scores is taken, there is a negative correlation. This finding suggests that as an individual's level of neuroticism increases, their solo programming effectiveness declines.

Despite the fact that no correlation was discovered between extroversion and pair programming efficiency, there is still a possibility that personality characteristics can play a role in how well a person develops software. Perhaps with a more detailed

Correlations

		SoloScoreLog	Neur
SoloScoreLog	Pearson Correlation	1	-.392*
	Sig. (1-tailed)	.	.019
	N	28	28
Neur	Pearson Correlation	-.392*	1
	Sig. (1-tailed)	.019	.
	N	28	28

*. Correlation is significant at the 0.05 level (1-tailed).

Fig. 5.6: Correlation of the log of the solo scores and neuroticism

study encompassing a larger breadth, more substantial data on this hypothesis can be uncovered.

5.3 Hypothesis III

1. (H3null) The individual efficiency of each partner in a pairing has no effect on how well the two individuals pair program.
2. (H3a) Pair programming efficiency can be seen as a function of the solo programming efficiency of each of the individuals in the pairing.

As was discussed for H1, the solo efficiency ratings for each individual in a pairing were, on average, less than the efficiency ratings for that individual when working in a pair programming situation. The next step is to determine if there is a link between the actual individual efficiency scores of each partner in a pairing and the overall efficiency score of the pair.

For each individual, two data points were added: one which gives the solo efficiency score for that individual's partner and one which calculates the average solo

efficiency score for each of the partners. Figure 5.7 is a graph which shows how pair programming efficiency scores are related to the solo efficiency scores of each individual in the pairing.

The trend line in Figure 5.7 shows that, in general, as the average scores of the two individuals in a pairings go up, so to does the pair efficiency score. This data is reinforced by an ANOVA test carried out on the same two variables and which can be seen in Figure 5.8. The level of significance is $< .05$ and thus we can say that the pair programming efficiency is directly related to how well the two individuals who make up the pairing programmed solo.

Thus, the higher the individual skill levels of each individual in a pairing, the higher the pair efficiency they will attain. This analysis is interesting as it has also been shown that, on average, pair efficiency and pair quality scores were higher than individual efficiency and quality scores. The alternate H3 is therefore the result that is chosen for this hypothesis.

5.4 Hypothesis IV

1. (H4null) Individuals and pairs undertake refactorings at the same difficulty level.
2. (H4a) Pair programmers undertake refactorings at higher difficulty levels than individuals.

In studying pair programming, it was an integral aspect of this thesis that this practice be analyzed in an Extreme Programming context. The goal of this kind of study is to determine what kind of connection pair programming has to other XP practices. None of the XP practices were designed to work in isolation, but were instead meant

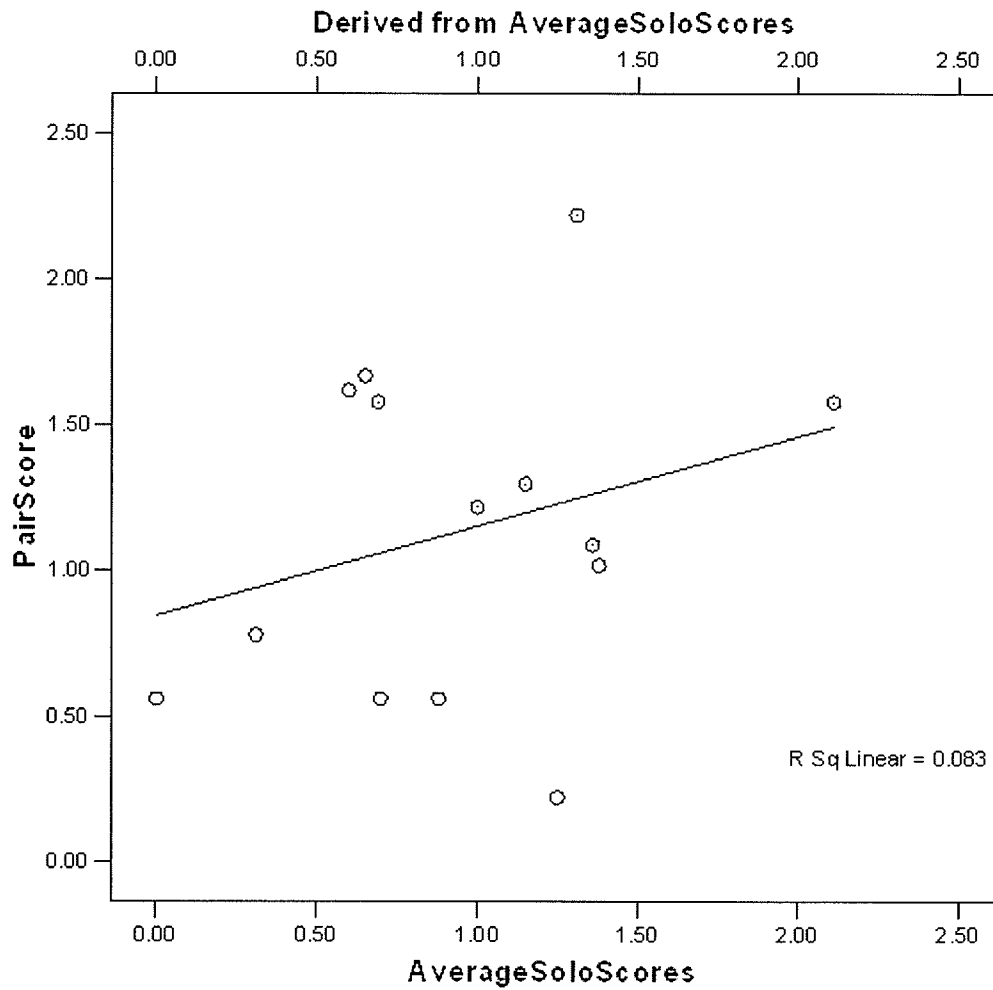


Fig. 5.7: A scatter plot graph showing the relationship between pair efficiency scores and the average solo efficiency scores of each partner

ANOVA

AverageSoloScores

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	4.359	10	.436	2.572	.042
Within Groups	2.881	17	.169		
Total	7.239	27			

Fig. 5.8: The ANOVA results from comparing the means of the pair programming efficiency scores with the average of the two programming solo efficiency scores

Correlations

		SoloWeight	PairWeight
SoloWeight	Pearson Correlation	1	-.528**
	Sig. (1-tailed)	.	.006
	N	22	22
PairWeight	Pearson Correlation	-.528**	1
	Sig. (1-tailed)	.006	.
	N	22	22

** . Correlation is significant at the 0.01 level (1-tailed).

Fig. 5.9: Correlation of refactoring weights for solo versus pair programmers

to weave a cohesive fabric. Thus, to give more depth to the study of pair programming, an experiment that studied pair programming and how it affects the practice of refactoring was devised and explained in Chapter 4.

Figure 5.9 shows the results of this experiment in terms of the difficulty level of each refactoring undertaken by individuals as opposed to pairs. To measure the difficulty level of the refactorings undertaken by the subjects, each kind of refactoring was divided into a category of either simple, moderate, or difficult and given the associated weights of 1, 2 or 3. Three difficulty levels were chosen because the domain of the problem included refactorings at the method level, class level, and application level. Thus, a comparison of the total weight of all the refactorings each subject undertook while working individually and while paired was made.

The correlation from Figure 5.9 is quite a strong negative correlation. The inference is that if a developer tends to choose more difficult refactorings while working individually, they will choose less difficult refactorings to work on when in a pair situation and vice versa.

This evidence is indicative of a larger pattern at work. It highlights the fact that if a subject is comfortable with programming individually, he or she may not be so comfortable in a pair. The reverse could also be said. Unfortunately, since Phase I and Phase II were designed as independent experiments, there is no way to compare the refactoring weights to the individual scores attained from Phase I.

While it is evident that the act of pair programming affects the practice of refactoring, it is not yet clear how that relationship works. Thus, the null hypothesis will have to be adopted. While there is a relationship between pair and solo refactoring choices, there is no statistical correlation indicating pair programming had a positive or negative affect on the difficulty levels of refactorings undertaken by the subjects.

5.5 Hypothesis V

1. (H5null) Pair programming satisfaction is unrelated to the personality characteristics and software development skill levels of the developers.
2. (H5a) Pair programming satisfaction is linked to the personality characteristics of developers and their software development skill level.

There is a direct link between how well a developer programs and how satisfied he or she is with her job. It is therefore in the best interests of any software development firm to keep their employees content. As aforementioned, a major hurdle for the XP methodology is convincing the actual developers that practices such as pair programming can make their job more satisfying.

To address this topic, a simple survey was provided to the subjects which attempted to gauge how much satisfaction they derived from the pair programming experience compared with the solo programming experience. The results clearly indicate the alternate hypothesis should be accepted for H4. Figure 5.10 shows that there is a direct

Correlations					
		Neur	Psy	Ext	Pair Satisfaction
Neur	Pearson Correlation	1	-.021	-.339*	-.054
	Sig. (1-tailed)	.	.458	.039	.392
	N	28	28	28	28
Psy	Pearson Correlation	-.021	1	.145	.412*
	Sig. (1-tailed)	.458	.	.231	.015
	N	28	28	28	28
Ext	Pearson Correlation	-.339*	.145	1	.374*
	Sig. (1-tailed)	.039	.231	.	.025
	N	28	28	28	28
PairSatisfaction	Pearson Correlation	-.054	.412*	.374*	1
	Sig. (1-tailed)	.392	.015	.025	.
	N	28	28	28	28

*. Correlation is significant at the 0.05 level (1-tailed).

Fig. 5.10: Correlation of pair programming satisfaction with personality characteristics

correlation between pair programming satisfaction and the level of extroversion of a subject.

These findings support the anecdotal evidence suggesting that the more outgoing an individual is, the more likely they are to enjoy pair programming. Another interesting finding, although not investigated deeply, was that the personality characteristic of psychoticism is also positively correlated with pair satisfaction. Thus, individuals with a higher level of psychoticism and extroversion as defined by the EPQ-R tend to enjoy the pair programming experience more than those with lower scores for those personality characteristics.

Pair satisfaction may also be related to how well a subject performs individually. If someone performs poorly on an individual basis, they may be more likely to enjoy pair programming. Figure 5.11 shows the correlation between solo scores and pair satisfaction. In this case there is no correlation between how a subject performed individually and how much they enjoyed the pair programming experience. This evi-

Correlations

		SoloQuality	SoloTime	SoloScore	Pair Satisfaction
SoloQuality	Pearson Correlation	1	-.643**	.860**	-.169
	Sig. (1-tailed)	.	.000	.000	.194
	N	28	28	28	28
SoloTime	Pearson Correlation	-.643**	1	-.925**	.126
	Sig. (1-tailed)	.000	.	.000	.261
	N	28	28	28	28
SoloScore	Pearson Correlation	.860**	-.925**	1	-.168
	Sig. (1-tailed)	.000	.000	.	.197
	N	28	28	28	28
PairSatisfaction	Pearson Correlation	-.169	.126	-.168	1
	Sig. (1-tailed)	.194	.261	.197	.
	N	28	28	28	28

** . Correlation is significant at the 0.01 level (1-tailed).

Fig. 5.11: Correlation of pair programming satisfaction with solo efficiency scores

dence leads to a conclusion that how much an individual enjoys pair programming has more to do with that individual's personality characteristics than with how good of a programmer he or she is.

6. CONCLUSION

The purpose of this thesis is to more deeply understand pair programming as it relates to the development methodology of Extreme Programming. As the title suggests, I wanted to 'unearth' the relationships between pair programming and developer efficiency, developer personality characteristics, and the other XP practice of refactoring.

The reason for this research is that XP is still a relatively new methodology which has yet to gain widespread acceptance in the business community as a valid development process. The reason for this hesitance can be attributed to the lack of research and basic understanding of what exactly XP is meant to be.

In particular, for XP to be better understood, the practices which constitute the methodology need to be researched more fully. For this thesis the XP practice of pair programming was chosen to be examined in great detail in order to provide a fuller understanding of the practice.

To this end, I designed and conducted two experiments which attempted to prove or disprove a set of five hypothesis described in Chapter 3. The first experiment focused on pair programming efficiency versus solo programming efficiency and what role the personality characteristics of the subjects played in those efficiency scores. The second experiment attempted to find what links exist between pair programming and refactoring.

As there were five hypotheses, there was a myriad of interesting results which I will recount here. The first is related to pair efficiency as opposed to solo efficiency.

While no statistically significant result was found, there were definite trends in the data which, given a larger sample size, may prove to be significant. In particular, the average pair score was higher than the average solo score. Also, individuals tended to be quicker in developing the software but had a higher rate of defects. The quality rating was enough in the favor of the pairs that it pushed the average pair scores higher than the average individual scores.

A major purpose of this thesis was to attempt to find a link between personality characteristics and pair programming efficiency. In particular, I had hypothesized that subjects who tended to be more extroverted would be more efficient pair programmers. This link was not proved. There is evidence, however, that personality characteristics do play a role in software development performance. The logarithm of the solo scores of the subjects was correlated to their level of neuroticism. This indicates that perhaps the size of this experiment was not large enough to find the necessary correlations to prove statistically significant.

The relationship of pair programming efficiency and personality characteristics, however, may not be as strong as the relationship between pair programming and solo efficiency. As was shown in H3, the tendency is that the higher two individuals constituting a pairing scored while working solo, the higher the pair programming efficiency would be. Thus, the skill level of a subject's partner directly affects how well the two programmed overall.

This finding is particularly interesting because it indicates that if you put a lesser skilled individual with a more highly skilled individual, the lesser skilled individual's efficiency raises. Since the average efficiency rating for pair programming is shown to be higher than for solo programming, there is a net overall increase in efficiency. In other words, the lesser skilled individual becomes more efficient to a degree that outweighs the amount the more highly skilled individual becomes less efficient when

the two are paired.

It was also important to understand what kind of personality characteristics are inherent in a satisfied pairing. The findings of this thesis suggest that the more extroverted an individual is, the more he or she will enjoy the pair programming experience. So while it is not clear how personality characteristics relate to pair programming efficiency, it is clear how they relate to pair programming satisfaction.

The more satisfied a developer is, the more likely he or she is to be a more dedicated developer. If pairing a person results in a high level of satisfaction, it can be indirectly said that personality characteristics do indeed play a role in how efficiently a pair develops code. Two introverts working together are much less likely to enjoy the pair programming experience and are thus more likely to not be as efficient as two highly extroverted people at developing code.

The final finding of this thesis is in regards to pair programming as it relates to refactoring. The more high level refactorings a subject undertakes individually, the less high level refactorings he or she will undertake in a pair situation and vice versa. Thus, the subjects tended to either work better by themselves or in a pair but not both.

Overall, I believe this thesis was successful in broadening and deepening our understanding of the pair programming practice as the experiments yielded a large amount of interesting data in how the practice of pair programming relates to solo programming, personality characteristics, and refactoring.

6.0.1 Future Work

The next step in this research is to widen the size and scope of the experiments. There was some initial data suggesting that personality characteristics can play a role in pair programming efficiency. With a larger sample size, it might be possible to uncover a more substantial link between personality characteristics like extroversion and pair

programming efficiency.

Furthermore, with a larger sample size it would be possible to divide the subjects into two groups where one group works individually and the other group works in pairs. In this manner each group could answer the same question instead of similar questions as was done in this experiment.

Along with an increase in sample size, it would also be interesting to pair a lesser skilled programmer with a more highly skilled programmer. After working together for a while, the two developers could be separated and asked to program another task individually to see how the pair programming experience affected their skill levels after the pair experience had ended. It would be useful to know if a lesser skilled individual will be buoyed by working closely with a more highly skilled individual.

The final future research that I would consider valuable would take this kind of experimental methodology and enlarge it to examine more practices of XP. By broadening the scope, it would be possible to conduct an experiment that would test a myriad of XP practices individually and in conjunction with other XP practices. For example, what was done in this thesis with pair programming and refactoring could be enlarged to include test driven development or metaphor or any number of other XP practices.

The end goal of this enlarged experiment would be to eventually get a deep understanding of XP in the same way this thesis attempted to unearth a deep understanding of pair programming

APPENDIX

A. SUPPORTING DOCUMENTS

A.1 The Stack Problem

Basically, your job is to create a standard stack –where the elements operate on a first in, last out basis - which has the four common methods push, pop, first, and last. The stack will be designed for strings only. The push method pushes a string onto the end of the stack. The pop method takes the last element of the stack, removes it from the stack, and returns it. The first method simply returns the first element of the stack while the last method returns the last element of the stack.

In all cases it is essential to test for null, both when items are placed into the stack and removed from the stack. In the case of pop, first, and last, if the stack is empty, return the String "Empty". If you accomplish all of these methods there is a reverseStack() method as well which should reverse all of the elements in the stack.

Your class name should be MyStack and the method names should be as specified or they will fail the automated tests. For the implementation, feel free to use any method you want, but I recommend using a Vector as it contains everything you will need.

To use a Vector, import java.util.Vector. You have half an hour to finish the assignment. The score you receive on the test will be determined by how many of the pre-determined tests your code passes.

A.2 The Bank Account Manager Problem

Account Management Problem

For this problem you are being asked to create part of an account management system. There will be three types of account you will be required to create: A checking account, a savings account, and a credit card account. For the credit card account, there are three types of credit cards: Visa, AMEX, and MC. You will have to create most of the classes. Some of the method definitions are provided, but most are absent (In cases where the methods are absent, it is important to think about which class the method belongs in). As with the Stack question, there is a suite of JUnit tests which will act as the account manager. All of these tests will be provided for your use so all that you have to focus on is getting the accounts to set and return the correct values. I encourage everybody to use inheritance and polymorphism wherever possible to minimize redundant code. The following are the methods you have to create:

- A constructor for each type of account
- A `getBalance()` method ' All of the money values are floats. To indicate a number is a float use an 'f' at the end of the number. Eg. 3.02f indicates that it is a float.
- A `GetCustomerName()` method
- A `GetAccountNumber()` method
- A `deposit(float amount)` method
- A `withdraw(float Amount)` method: Returns a string: Either 'Success' or 'Failure' depending on whether there is enough money in the account.
- A `getAccountType()` method which will return a string for the type of account: 'Checking Account',

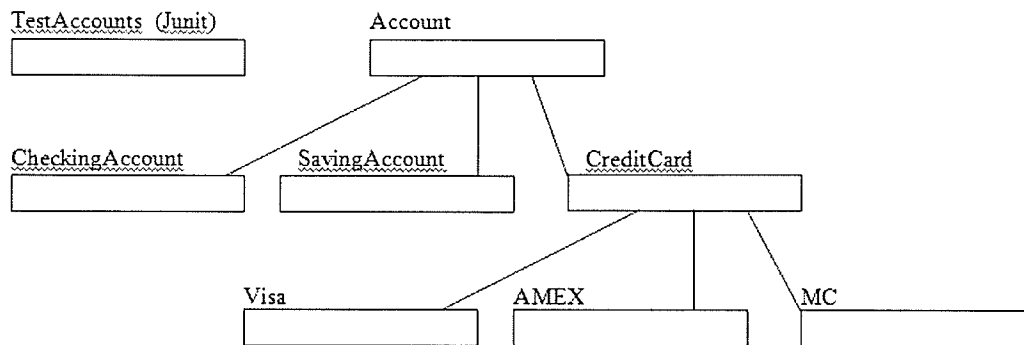


Fig. A.1: Class Hierarchy

- ‘Saving Account’, ‘Visa’, ‘AMEX’, or ‘MC’. If any other string is returned an error will ensue.
- A `calculateInterest()` method: This method will use the interest rate for the particular account and calculate how much interest has accumulated based on the `balance()` and returns a float.
- A `transferFundsTo(AccountTo, amount)` method. The `accountTo` parameter is the account to transfer the funds to from the current account. Returns the String ‘Success’ if transfer successful and ‘Failure’ if not.
- Finally, in the case of the credit card accounts, there will be a variable called `Limit` which can’t be exceeded by how much money is withdrawn from the credit card. Similarly, the checking and savings accounts can not go below zero.

`refHierarchy` should give you an idea of what the system should look like.

`refCLT` lists the limits on each account.

Account Type	Interest Rate	Credit Limit
Checking Account	5%	
Saving Account	10%	
Visa	10%	5000.00\$
AMEX	15%	3000.00\$
MC	20%	1000.00\$

Tab. A.1: Account Limits

A.3 The Survey

The survey questions for the subjects

General Survey of Subject Attitudes & Opinions	Strongly Disagree		Neut ral	Strongly Agree	
1. I consider my programming skills to be good	1	2	3	4	5
2. I am an outgoing person	1	2	3	4	5
3. I work well with others	1	2	3	4	5
4. I prefer to work with others, rather than alone	1	2	3	4	5
5. It is important to have high quality code instead of simply functional code	1	2	3	4	5
6. I have a firm understanding of Extreme Programming	1	2	3	4	5
7. I have practiced XP before	Yes			No	
8. I have a firm understanding of pair programming	1	2	3	4	5
9. I have practiced pair programming before	Yes			No	
10. My programming experience (in years) is	0-1		2-3	4 or more	
11. I enjoyed the pair programming experiment	1	2	3	4	5
12. I find that I'm more productive with pair programming than solo programming	1	2	3	4	5
13. I find that pair programming results in higher quality code than solo programming	1	2	3	4	5
14. I would like to work with pair programming in the future	1	2	3	4	5

Fig. A.2: Experiment Survey

Refactoring	Description
Unnecessary comments	
Large method	A method has a lot of lines of code
Large class	Too many methods/instance variables
Method has too many parameters	
Type imbedded in name	Method name includes type
Uncommunicative name	
Dead Code	Code that isn't used
Speculative Generality	Code that adds functionality that is never used
Magic Number	A constant placed in the middle of some code
Duplicated code (identical)	
Duplicated code (similar effects)	
Null check	Null is used instead of an appropriate object
Complicated Boolean expression	
Special cases	Complex if statements
Simulated inheritance	Sequential ifs, switches, or <u>instanceOf</u>
Public data members	
Data Clump	Several lines of code always appear together
Temporary fields	
Refused bequest	Subclass doesn't implement some methods of parent or should not be <u>subclass</u>
Inappropriate intimacy	Subclass accesses information that should be private in the parent
Lazy Class	A class without a purpose
Feature Envy/Inappropriate intimacy (general)	A method manipulates data of other classes
Message chains	<u>a.b().c().d()</u> etc...
Middle Man	A class simply routes commands to a different class
Divergent Change	A class goes in two directions
Shotgun surgery	Something that should be simple to change requires modifying several classes

Fig. A.3: List of Possible Refactorings

A.4 The Refactor List

A.5 The Refactor Problem

Refactoring Experiment – The Problem Domain

This experiment is being conducted to determine what kind of refactoring, given a problem with many possible refactorings, will be done by 4th year students on an already functional Java program. The program at each stage of development, i.e. after each refactoring, will be saved and analyzed at a later date using a metrics suite and the JUnit tests.

Everybody taking part in the experiment should be familiar with Java and JUnit. In the initial state, all of the JUnit tests will pass. In refactoring the application, it is up to the subjects to ensure that after each refactoring the JUnit test cases still pass. A layer of abstraction, in the form of the `toRefactor` class, exists so that the subjects can change how methods and classes are called without changing the actual JUnit test Class (`RefactorTest`).

The problem itself is a fairly straightforward example of a doubly linked list being used to keep track of different car models. The doubly linked list is sorted in ascending order on the `Car.model` field. For example, the list would be ordered `benz`, `civic`, `temp`, `Volvo` if those Strings were entered as the model of car when the car was created. The doubly linked list has the functionality implemented so that it can add a car, delete a car, or find a car based on the model.

As for Car, there are two types of car that are actually created for this program: `Luxury` and `Economy`. It is fair to assume that only `Luxury` and `Economy` cars will be created. Each car has a model, a make, engine specifications, etc Each type of car also has some type specific information.

This program is not pretty, but it is completely functional. Your task is to refactor as much as you feel necessary to make the program into an efficient, communicative

application. Along with the running program, each of you will get a paper listing of all the refactorings that you should be looking for. These refactorings were taken from the book 'Refactoring Workbook' by William C. Wake. When you refactor something in the program, you should be able to correlate that change with one of the refactorings mentioned by Wake.

You will also be given a document that is basically a numbered list. Each time you complete a refactoring, you will be expected to write down which refactoring you carried out and a short description of how. You will then export the project file to a zip file which says the computer name and the number which matches what you wrote for your refactoring. If you do the same minor refactoring then you can lump it together as just one refactoring. For example, if you change a few variable names to be more communicative. Everything else, however, you should export after each time you've finished refactoring. Remember, you haven't finished refactoring if any of the JUnit tests fail

The experiment itself will be divided into two phases. In the first phase everybody will refactor as individuals. In the second phase, everybody will refactor in pairs. The experimenter will randomly pair everybody. It will be up to each particular pairing at this point to decide whose computer to use. It is expected that the pairs will simply continue working on a program already started by one of the partners. The pairs will be given a second sheet of refactorings, just like the first, and then continue to refactor as a pair.

A.6 The Total Refactorings

- VarA Changed uncommunicative name
- VarB Speculative generality (removed hatchback)
- VarC Dead code (removed unneeded methods, variables, or classes)
- VarD Removed public data members (Use getter and setter)
- VarE Make abstract class (Car)
- VarF Deleted unnecessary comments
- VarG Bypassed Luxury and Economy classes
- VarH Removed Super() in constructor
- VarI Simplified method (general)
- VarJ Move variables & methods to more appropriate class
- VarK Chagned variable type
- VarL Add super() in constructor
- VarM Minimize Null usage
- VarN Add Constructor
- VarO Added Comments
- VarP Modified (not specified)
- VarQ Added methods in Luxury or Economy Classes
- VarR Removed duplicated class

- VarS Removed classes Luxury & Economy
- VarT Inappropriate intimacy
- VarU Removed Duplicated Code
- VarV Include proper OO principles (Added inheritance)
- VarW Added ComparteTo method
- VarX Removed all references to *this*
- VarY Simplified Complicated boolean expression

Variable ID	Weight	Description
VarA	1	Changed Uncommunicative name
VarB	2	Speculative Generality (removed hatchback)
VarC	2	Dead Code (removed unneeded methods, variables, or classes)
VarD	2	Removed Public Data Members (Use getter and setter)
VarE	3	Make Abstract Class (Car)
VarF	1	Deleted Unnecessary Comments
VarG	2	Bypassed Luxury and Economy classes
VarH	2	Removed Super() in constructor
VarI	1	Simplified Method
VarJ	2	Move variables & methods to more appropriate class
VarK	1	Changed variable type
VarL	2	Add super() in constructor
VarM	1	Minimize Null usage
VarN	3	Add Constructor
VarO	1	Added Comments
VarP	3	Modified (not specified)
VarQ	1	Added methods in Luxury or Economy
VarR	2	Removed duplicated class
VarS	2	Removed Lazy classes Luxury & Economy
VarT	2	Inappropriate intimacy
VarU	3	Removed Duplicated Code
VarV	3	Include proper OO principles (Added inheritance)
VarW	2	Added CompartTo method
VarX	1	Removed <i>this</i> .
VarY	2	Simplified Complicated boolean expression

Tab. A.2: Refactoring Weights

A.7 The Refactoring Weights

Subject	Solo Quality	Pair Quality	Solo Time	Pair Time	Solo Score	Partner Solo Score	Average Solo Scores	Pair Score
a368	1	1	0.37	0.63	2.73	1.5	2.11	1.58
a329	0.25	0.56	1	1	0.25	1.15	0.7	0.56
a344	0	0.56	1	1	0	0	0	0.56
a757	0	0.78	1	1	0	0.63	0.31	0.78
a749	0	0.56	1	1	0	1.76	0.88	0.56
a375	0	1	1	0.6	0	1.3	0.65	1.67
a732	1	1	0.57	0.98	1.76	1	1.38	1.02
a710	0.63	0.78	1	1	0.63	0	0.31	0.78
a721	0	0.56	1	1	0	0	0	0.56
a351	0.88	1	1	0.77	0.88	1.43	1.15	1.3
a380	1	1	0.67	0.63	1.5	2.73	2.11	1.58
a409	1	0.56	0.57	1	1.76	0	0.88	0.56
a337	1	1	0.77	0.6	1.3	0	0.65	1.67
a415	1	1	0.7	0.77	1.43	0.88	1.15	1.3
a420	1	1	1	0.98	1	1.76	1.38	1.02
a127	1	0.56	0.87	1	1.15	0.25	0.7	0.56
B902	0	1	1	0.62	0	1.2	0.6	1.62
B803	1	1	0.53	0.45	1.88	0.75	1.31	2.22
B305	0	1	1	0.92	0	2.73	1.36	1.09
B898	1	0.22	0.77	1	1.3	1.2	1.25	0.22
B871	1	1	1	0.63	1	0.38	0.69	1.58
B853	0.75	1	1	0.45	0.75	1.88	1.31	2.22
B953	0	1	1	0.82	0	2	1	1.22
B929	1	1	0.83	0.62	1.2	0	0.6	1.62
B296	1	1	0.5	0.82	2	0	1	1.22
B254	1	1	0.37	0.92	2.73	0	1.36	1.09
B214	1	0.22	0.83	1	1.2	1.3	1.25	0.22
B278	0.38	1	1	0.63	0.38	1	0.69	1.58

Fig. A.4: The data collected from phase I

A.8 Experimental Data

Psy	Ext	Neur	Solo Score Log	Pair Score Log	Pair Better	Solo Better	Relative Score	Absolute Score	Pair Satisfaction
0.21	0.41	0.29	1	0.46	0	2.73	-1.15	-2.73	4
0.06	0.45	0.13	-1.39	-0.58	0.56	0	0.31	0.56	3
0.18	0.64	0.58	-9.21	-0.58	0.56	0	0.56	0.56	4
0.24	0.45	0.58	-9.21	-0.25	0.78	0	0.78	0.78	3
0.3	0.41	0.46	-9.21	-0.58	0.56	0	0.56	0.56	3
0.12	0.27	0.96	-9.21	0.51	1.67	0	1.67	1.67	4
0.15	0.55	0.5	0.57	0.02	0	1.76	-0.75	-1.76	5
0.27	0.77	0.33	-0.46	-0.25	0.78	0	0.15	0.78	4
0.18	0.82	0.21	-9.21	-0.58	0.56	0	0.56	0.56	5
0.21	0.91	0.25	-0.13	0.26	1.3	0	0.43	1.3	5
0.3	0.77	0.33	0.41	0.46	1.58	0	0.08	1.58	3
0.03	0.55	0.75	0.57	-0.58	0	1.76	-1.21	-1.76	2
0.12	0.32	0.5	0.26	0.51	1.67	0	0.36	1.67	2
0.21	0.59	0.38	0.36	0.26	0	1.43	-0.12	-1.43	5
0.15	0.5	0.46	0	0.02	1.02	0	0.02	1.02	1
0.15	0.27	0.42	0.14	-0.58	0	1.15	-0.6	-1.15	4
0.33	0.86	0.29	-9.21	0.48	1.62	0	1.62	1.62	5
0.18	0.64	0.13	0.63	0.8	2.22	0	0.35	2.22	3
0.45	0.27	0.46	-9.21	0.09	1.09	0	1.09	1.09	4
0.3	0.95	0.58	0.26	-1.51	0	1.3	-1.08	-1.3	5
0.3	0.64	0.33	0	0.46	1.58	0	0.58	1.58	3
0.27	0.59	0.33	-0.29	0.8	2.22	0	1.47	2.22	5
0.24	0.91	0.54	-9.21	0.2	1.22	0	1.22	1.22	4
0.03	0.91	0.13	0.18	0.48	1.62	0	0.42	1.62	3
0.09	0.41	0.25	0.69	0.2	0	2	-0.78	-2	3
0.21	0.41	0.13	1	0.09	0	2.73	-1.64	-2.73	3
0.3	0.86	0.17	0.18	-1.51	0	1.2	-0.98	-1.2	5
0.15	0.82	0.08	-0.97	0.46	1.58	0	1.2	1.58	3

Fig. A.5: The data collected from phase I

Subject	Solo Number	Solo Method	Solo Class	Solo Application	Pair Number	Pair Method	Pair Class	Pair Application	Solo Weight	Pair Weight
F101	2	0	1	1	1	0	0	1	3	11
F102	5	3	2	0	2	1	1	0	12	0
F103	5	2	3	0	0	0	0	0	13	4
F104	4	1	1	2	1	0	0	1	12	13
F105	3	1	1	1	7	4	2	1	6	13
F107	5	3	2	0	7	3	2	2	7	13
F108	7	1	4	2	7	3	2	2	16	4
F109	4	2	2	0	2	1	1	0	9	0
F110	6	0	4	2	0	0	0	0	11	9
F111	5	1	3	1	7	5	2	0	9	6
F112	6	4	2	0	3	2	1	0	10	11
F113	6	0	6	0	7	4	2	1	12	11
F116	1	0	0	1	7	4	2	1	3	6
F120	1	0	0	1	3	2	1	0	1	11
F121	1	1	0	0	2	1	1	0	2	15
F122	2	1	1	0	7	1	2	4	5	20
F123	5	1	2	2	10	3	4	3	7	9
F124	5	5	0	0	7	5	2	0	7	20
F125	6	1	4	1	10	3	4	3	11	13
F126	6	2	4	0	7	4	2	1	11	4
F127	3	1	2	0	2	1	1	0	5	15
F144	6	1	4	1	7	1	2	4	15	0

Fig. A.6: The data collected from phase II

BIBLIOGRAPHY

- [1] T. Abdel-Hamid and S. E. Madnick. *Software Project Dynamics: An Integrated Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
- [2] agilemanifesto.org. Manifesto for Agile Software Development, 2001.
- [3] J. Aiken. Technical and human perspectives on pair programming. *ACM SIG-SOFT Software Engineering Notes*, 29(5):1–14, Sept. 2004.
- [4] M. Ally, F. Darroch, and M. Toleman. A framework for understanding the factors influencing pair programming success. In *XP 2005*, pages 82–91, 2005.
- [5] M. Alshayeb and W. Li. An empirical study of relationships among extreme programming engineering activities. *Information and Software Technology*, pages www.sciencedirect.com/science/article/B6V0B-4JD1130-3/2/3279b82f8ab8e2d7c643a1417a06103b, 2006.
- [6] S. Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley and Sons, 2002.
- [7] D. Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall, Upper Saddle River, NJ, 2003.
- [8] K. Auer and R. Miller. *Extreme Programming Applied: Playing to Win*. Pearson Education, Inc., Boston, 2002.

- [9] K. Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.
- [10] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley Longman, Inc., Reading, Massachusetts, 2000.
- [11] K. Beck and B. Boehm. Agility through discipline: A debate. *Computer*, 36(6):44–46, 2003.
- [12] G. Canfora, A. Cimitile, and A. Visaggio. Working in pairs as a means for design knowledge building: an empirical study. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC'04)*, pages 62–68, June 2004.
- [13] T. DeMarco and B. Boehm. The agile methods fray. *Computer*, 35(6):90–92, 2002.
- [14] A. J. Dick and B. Zarnett. Working in pairs as a means for design knowledge building: an empirical study. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC'04)*, pages 62–68, June 2004.
- [15] A. English. Extreme programming: It's worth a look. *IT Professional*, 4(3):28–50, 2002.
- [16] H. Erdogmus and L. Williams. The economics of software development by pair programmers. *The Engineering Economist*, 48(4):283–319, 2003.
- [17] H. J. Eysenck and S. B. G. Eysenck. *Manual of the Eysenck Personality Questionnaire*. San Diego, California, 1993.
- [18] W. J. Forrester. Industrial dynamics: A major breakthrough for decision makers. *Harvard Business Review*, 36(4):37–66, 1958.

-
- [19] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley, Boston, MA, 2000.
- [20] H. Gallis, E. Arisholm, and T. Dybå. Extreme programming and its development practices. In *Proceedings of the 22nd International Conference on Information Technology Interfaces (ITI 2000)*, page 97, Pula, Croatia, June 2000.
- [21] H. Gallis, E. Arisholm, and T. Dybå. An initial framework for research on pair programming. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE'03)*, pages 132–142, Oct. 2003.
- [22] B. George and L. Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337–342, Apr. 2004.
- [23] S. S. Gifford, S. M. Henry, and P. K. Schoenhoff. Personality and type considerations for programming teams in computer science classes. In *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2003)*, pages 91–95, Orlando, Florida, USA, July 2003.
- [24] R. Gittins, S. Hope, and I. Williams. Qualitative studies of xp in a medium sized business. *UPGRADE The European Online Magazine for the IT Professional*, 3(2), Apr. 2002.
- [25] ISEE systems, Inc. *ithink v. 8.0*. Lebanon, NH, 2004.
- [26] A. Johnston and C. S. Johnson. Extreme programming: A more musical approach to software development? In *Proceedings of the Fourth International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2003)*, pages 325–327, Genova, Italy, May 2003.

- [27] R. Juric. Extreme programming and its development practices. In *Proceedings of the 22nd International Conference on Information Technology Interfaces (ITI 2000)*, page 97, Pula, Croatia, June 2000.
- [28] G. Kahen, M. M. Lehman, and J. F. Ramil. *System Dynamics Modelling for the Management of Long Term Software Evolution Processes*. www.doc.ic.ac.uk/deptechrep/DTR00-16.pdf, 2000.
- [29] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 576 – 585, 2002.
- [30] N. Katira, L. Williams, E. Wiebe, C. Miller, S. Balik, and E. Gehringer. On understanding compatibility of student pair programmers. In *Proceedings of the ACM Special Interest Group on Computer Science Education (SIGCSE 2004)*, pages 7–11, Norfolk, Virginia, Mar. 2004.
- [31] S. Kuppuswami, K. Vivekanandan, P. Ramaswamy, and P. Rodrigues. The effects of individual xp practices on software development effort. *ACM SIGSOFT Software Engineering Notes*, 28(6):6–6, Nov. 2003.
- [32] L. Layman, L. Williams, and L. Cunningham. Exploring extreme programming in context: An industrial case study. In *Proceedings of the Agile Development Conference (ADC'02)*, pages 32–41, 2004.
- [33] P. Louridas. Junit: Unit testing and coding in tandem. *IEEE Software*, July 2005.
- [34] K. M. Lui and K. C. Chan. When does a pair outperform two individuals? In *Proceedings of the Fourth International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2003)*, pages 225–233, Genova, Italy, May 2003.

-
- [35] K. M. Lui and K. C. Chan. A cognitive model for solo programming and pair programming. In *Proceedings of the Third International Conference on Cognitive Informatics (ICCI 2004)*, pages 363–367, Reno, Nevada, USA., Feb. 2004.
- [36] F. Macias, M. Holcombe, and M. Gheorghe. A formal experiment comparing extreme programming with traditional software construction. In *Proceedings of the fourth Mexican international conference on Computer Science (ENC'03)*, pages 73–80, Sept. 2003.
- [37] M. Mantyla. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In *2005 International Symposium on Empirical Software Engineering*, pages 287–296, 2005.
- [38] C. McDowell, B. Hanks, and L. Werner. Experimenting with pair programming in the classroom. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2003)*, pages 60–64, Thessaloniki, Greece, July 2003.
- [39] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald. The impact of pair programming on student performance, perception and persistence. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 602–607, May 2003.
- [40] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald. The impact of pair programming on student performance, perception and persistence. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 602–607, May 2003.
- [41] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126 – 139, Feb. 2004.

- [42] V. Mišić, H. Gevaert, and M. Rennie. Extreme dynamics: Modeling the extreme programming software development process. In *The Fifth International Workshop on Process Simulation and Modeling (PROSIM 2004)*, pages 237–242, Edinburgh, Scotland, May 2004.
- [43] M. M. Müller. Two controlled experiments concerning the comparison of pair programming to peer review. *The journal of Systems and Software (Article in Press)*, 2005.
- [44] M. M. Müller. A preliminary study on the impact of a pair design phase on pair programming and solo programming. *Information and software technology*, 2006.
- [45] M. M. Müller and F. Padberg. On the economic evaluation of XP projects. In *Proceedings of the 2003 European Conference on Software Engineering (ESEC 2003)*, pages 168–177, Helsinki, Finland, Sept. 2003.
- [46] M. M. Müller and F. Padberg. An empirical study about the feelgood factor in pair programming. In *Proceedings of the 10th International Symposium on Software Metrics (METRICS'04)*, pages 151–158, Sept. 2004.
- [47] myersbriggs.org. Myers and Briggs Foundation website, 2002.
- [48] H. Natsu, J. Favela, and A. L. Morán. Distributed pair programming on the web. In *Proceedings of the Fourth Mexican International Conference on Computer Science (ENC'03)*, pages 81–88, Sept. 2003.
- [49] J. Newkirk. Introduction to agile processes and extreme programming. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 695–696, 2002.

-
- [50] J. Noble, S. Marshall, S. Marshall, and R. Biddle. Less extreme programming. In *Proceeding of the Sixth Australasian Computing Education Conference (ACE 2004)*, pages 217–226, Dunedin, New Zealand, Feb. 2004.
- [51] F. Padberg and M. M. Müller. Analyzing the cost and benefit of pair programming. In *Proceedings of the Ninth International Software Metrics Symposium (METRICS 2003)*, pages 166–177, Sydney, Australia, Sept. 2003.
- [52] M. M. Padberg, Frank Müller. Modeling the impact of a learning phase on the business value of a pair programming project. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pages 142–149, Nov. 2004.
- [53] A. Parrish, R. Smith, D. Hale, and J. Hale. A field study of developer pairs: productivity impacts and implications. *IEEE Software*, 21(5):76–79, 2004.
- [54] L. J. Perelman. Time in system dynamics. *TIMS Studies in the Management Sciences*, 14:75–89, 1980.
- [55] D. Pfahl and K. Lebsanft. Integration of system dynamics modelling with descriptive process modelling and goal-oriented measurement. *The Journal of Systems and Software*, 46(2-3):135–150, Apr. 1999.
- [56] H. Stanislaw, B. Hesketh, S. Kanavaros, T. Hesketh, and K. Robinson. A note on the quantification of computer programming skill. *International Journal on Human-Computer Studies*, 41(3):351–362, Sept. 1994.
- [57] G. Succi, M. Stefanovic, and W. Pedrycz. Quantitative assessment of extreme programming practices. In *Canadian Conference on Electrical and Computer Engineering, 2001*, pages 81–86, May 2001.

- [58] L. Thomas, M. Ratcliffe, and A. Robertson. Code warriors and code-a-phobes: A study in attitude and pair programming. In *Proceedings of the ACM Special Interest Group on Computer Science Education (SIGCSE 2003)*, pages 94–102, Hong Kong, China, Aug. 2004.
- [59] J. E. Tomayko. A comparison of pair programming to inspections for software defect reduction. *Computer Science Education*, 12(3):213–222, Sept. 2002.
- [60] T. VanDeGrift. Coupling pair programming and writing: Learning about students' perceptions and processes. In *Proceedings of the ACM Special Interest Group on Computer Science Education (SIGCSE 2004)*, pages 2–6, Norfolk, Virginia, Mar. 2004.
- [61] W. C. Wake. *Extreme Programming Explored*. Addison Wesley, Boston, 2002.
- [62] W. C. Wake. *Refactoring Workbook*. Addison Wesley, Boston, 2004.
- [63] P. Wernick and T. Hall. The impact of using pair programming on system evolution: a simulation-based study. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 422–426, Sept. 2004.
- [64] L. Williams, C. McDowell, N. Nagappan, J. Fernald, and L. Werner. Building pair programming knowledge through a family of experiments. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE 2003)*, page 143, Rome, Italy, Sept. 2003.
- [65] L. Williams, A. Shukla, and A. Antón. An initial exploration of the relationship between pair programming and Brooks' law. In *Proceedings of the 2004 Agile Development Conference (ADC'04)*, pages 11–20, June 2004.

-
- [66] L. Williams and R. Kessler. *Pair Programming Illuminated*. Addison Wesley, Boston, 2003.
- [67] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair programming. *IEEE Software*, 17(4):19–25, July/August 2000.
- [68] L. A. Williams and R. R. Kessler. Experiments with industry's
pair programming
model in the computer science classroom. *Computer Science Education*,
11(1):7–20, Jan. 2001.