TO TEST BEFORE OR TO TEST AFTER – AN EXPERIMENTAL INVESTIGATION OF THE IMPACT OF TEST DRIVEN DEVELOPMENT

by

VIKRAM S BHADAURIA

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2009

ACKNOWLEDGEMENTS

When I started working towards my PhD, I did not realize that this endeavor would not only help me accomplish my research goals, but would help me become a better human being as well. The valuable lessons I have learned during the process of completing my dissertation work will continue to be my guiding light for the future. I consider myself to be fortunate that I received an opportunity to learn from some of the luminaries in the field of research.

My dissertation committee chair, Dr. Radha Mahapatra, was a constant source of inspiration during the whole PhD program. His able guidance, sound advice and continuous encouragement helped me immensely, especially during the dissertation phase. I also wish to express my deep gratitude to Dr. Sridhar Nerur. His keen insight in programming helped me in identifying the task for the experiment. His enthusiasm for this project and readiness to support were motivational. He also helped me in recruiting students for the experiment. Dr. Kenneth Price provided his expert help in building the model. I greatly benefited from his expertise in the area of small group research. Through his unique sense of humor, Dr. Craig Slinkman always encouraged me to keep working hard. I audited some of his classes in which he used TDD to teach programming. It helped me learn more about Eclipse IDE and that knowledge was very useful in the project. Dr. Mark Eakin provided his expert guidance that helped me deal with statistics related issues in data analysis. I am very grateful to all

of the members of my dissertation committee for the much needed support and guidance. I also wish to thank additionally Ms Andrea Webb who was always ready to run the extra mile to help PhD students.

I also wish to thank several of my friends who have helped me at different stages of the dissertation. I thank my seniors, Dr. Venugopal Balijepally and Dr. George Mangalaraj for their advice and support. Ralph Yeh and Lulu Zhang helped in grading the experimental tasks, and I am thankful to them for their help. I also wish to thank Dr. Anil Singh for his words of encouragement and Kamphol and Jairo for logistics support. Special thanks to Dr. Gayle White and Dr. Stacy M. Clanton for the editorial help.

Most importantly, I wish to thank my parents and family for their support. My mother has been the inspiration and guiding force behind all my endeavors. It was her faith in me that helped me realize my potential. My technology savvy brother, whom I am proud of, was always there for discussion. Last but not least, I am thankful to my wife and my daughter who had to bear the inattention that resulted from my preoccupation with the dissertation.

May 12, 2009

ABSTRACT

TO TEST BEFORE OR TO TEST AFTER – AN EXPERIMENTAL
INVESTIGATION OF THE IMPACT OF
TEST DRIVEN DEVELOPMENT

Supervising Professor: Dr. Radha Mahapatra

Test Driven Development (TDD) requires the developer to create the test suite before designing and writing the application program. Unlike traditional software development practices, in TDD test development precedes application development. Such a practice also redefines the role of the developer. Lately, TDD is growing in popularity as a part of Agile methodologies. There is a critical need for rigorous empirical research to understand the role and impact of TDD as a software development practice. The goal of this dissertation research is to fill this gap. A laboratory experiment was conducted to understand the influence of TDD on the outcomes of the software development process. Software quality, learning and task satisfaction were examined as outcome variables. In the experiment, groups that used the traditional method of software development were compared with those that used TDD. Individual programmers were also compared with paired programmers, when both used TDD.

Individual programmers using TDD were found to produce a higher quality of software as compared to programmers using the traditional method of software development. Comprehension, as a part of the learning measure, was also found to be significantly higher in case of programmers using TDD as compared to those using the traditional method of software development. Programmers using TDD were also found to achieve higher task satisfaction as compared to those using the traditional method of software development. Another important finding of this study was that collaborating pairs outperformed second best programmers in nominal groups in terms of software quality, when both groups used TDD.

TABLE OF CONTENTS

Appendix

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

Increasingly, businesses are adopting IT solutions to streamline their processes. Rapid proliferation of the internet has enabled integration and globalization of businesses. The effect of change at one place can now be felt almost anywhere on the globe. Simultaneous operations at co-locations add to the complexity of the processes. Dealing with such a dynamic and complex environment thus becomes a major challenge for managers and IT professionals today.

The Standish report has often been cited to claim that the majority of the Software projects are not successful in meeting targets. The Standish group studied over 10,000 global software projects between October 2005 and November 2006 and in their latest Chaos report they claim that by deploying improved development processes, 35% of the software projects were successfully completed within the targeted time, budget and functionality, which is a significant improvement over the corresponding figure of 29% in the year 2004 (PMI Report 2007). Though it is encouraging to note, yet it is far from satisfactory as almost two thirds of the software projects are still delayed or cancelled.

Software development is perhaps the most important part of any Software project. A significant amount of time and effort are expended in checking for, and eliminating,

errors in the code. A recent study found that even at high levels of process maturity, software size remains the most significant factor that affected the development effort, quality and cycle time (Agrawal, Chari 2007).

1.1 Software Development Process Improvements

To deal with size, complexity and rapid change, IT organizations are increasingly looking for better Software development methodologies. Agile methodologies have been keenly employed by various organizations. Using agile methodologies, researchers claim, organizations can enhance and better utilize personnel capability and deliver a higher quality of software. For instance in Pair programming, though it may require more man power to accomplish the design task, the time spent in removing errors is greatly reduced. Researchers have stressed the cost savings through quality enhancements. Krishnan et al (2000) found that higher personnel capability, early deployment of resources in the design phase of development and improved process factors aid in achieving a higher quality in software product (Krishnan et al. 2000). One of the most prevalent techniques to assure quality of software is through testing. Although to be fully assured of the quality of software, exhaustive testing is needed, a study claims that a finite number of tests can potentially assure highest level of quality in a software product (Kuhn, Wallace & Gallo Jr. 2004). Regardless of the number of the tests required, the quality of the software product is widely accepted to be correlated to testing.

Test Driven Development (TDD) is an agile methodology that has caught the attention of the IT industry in recent times. Traditionally, the code has been inspected for errors after the developer has completed writing the code. Typically, the coder and the tester were two different people. TDD reverses the paradigm with test being written even before writing the first line of code. In TDD the role of the developer also changes as now he is also the tester of the code. Since quality is ascertained by testing the code, TDD holds the promise of a high quality tightly written code. In this study, the terms the Test First approach and Test Driven Development (or TDD) are used interchangeably.

Though TDD has been claimed to improve performance, yet there is no consensus among findings; Some found significant improvements, some did not find any change, and while some actually found deterioration in performance, both in the industrial setting and in the academic world (Jeffries, Melnik 2007). Erodgmus et al (2005) have tried to measure the effectiveness of the Test First approach to Programming. They evaluate effectiveness in terms of productivity (defined as number of stories per unit effort) and the software quality (defined as per number of defects) in their experiment. They find support for their hypothesis that more tests leads to better quality. However, the sample size of 24 limits the generalizability of their findings (Erdogmus, Morisio & Torchiano 2005). Some studies have tried to combine both pair programming and TDD. One researcher investigated effects of Pair programming on the external code quality using TDD (Madeyski 2005). He found that using TDD in place of the

conventional Test Last approach actually caused deteriorated performance in individuals and did not have any significant change in the case of pairs. The study did not use any theory to explain the observation, nor did the research explore any outcome variables other than Quality. There is a gap in the literature, as there is no consensus among academicians and practitioners whether TDD works or not. And, there are no theoretical explanations that help in understanding the phenomenon of TDD.

There is a clear lack of empirical evidence as regards the effectiveness of TDD in the case of developers working individually versus those working as collaborating pairs. Researchers have called for a new research to be done in this area that examines software quality and productivity while investigating adoption issues such as learning, suitability and fit, and motivation (Janzen, Saiedian 2005). This study aims to address this gap in the literature. It strives to measure effectiveness of TDD using constructs drawn from theory. This research explains the 'why' question from a theory lens while analyzing the findings. And, most importantly, it aids in a more fine grained understanding of the phenomenon through additional process variables. Additionally, this research seeks to answer the question concerning whether TDD improves learning.

1.2 Research Questions

This study aims to examine the following research questions:

1. Do programmers develop software of a relatively higher quality when they use TDD instead of the traditional method of software development?

2. When programmers use TDD, do they develop software of a relatively higher quality when they work as collaborating pairs rather than as individuals?

3. Do programmers learn more when they use TDD in place of the traditional method of software development?

4. When programmers use TDD, do they learn more when they work as collaborating pairs rather than as individuals?

5. Do software developers feel more satisfied when they use TDD instead of the traditional method of software development?

6. While using TDD, are software developers more satisfied when they work as collaborating pairs as compared to when they work individually?

1.3 Overview of the Dissertation

A laboratory experiment was conducted to examine with precision and rigor the causality in the hypothesized relationships. Chapter 2 is composed of comprehensive literature review. Chapter 3 is devoted to developing the Research Model and hypotheses. The research methodology is discussed in Chapter 4. Research results are

discussed in Chapter 5. Finally, conclusions, implications to business and academics, and limitations of the study are discussed in Chapter 6.

1.4 Contributions of the study

This study is likely to make significant contributions to the field, some of which are –

1. Organizational learning is an important area of research and it is a major determinant of performance especially in the case of knowledge intensive organizations. For IT companies, the issue of organizational learning becomes even more critical because of the high attrition rate among software developers attributed to varied factors by researchers (Ahuja et al. 2007). This study provides insights into the organizational learning process and will help the IT managers make an informed decision on appropriate software development methodology that can result in higher learning, thereby leading to better performance.

2. This study aims to provide a theory driven model that explains the process of the TDD approach as compared to the traditional approach of software development. Since the existing studies have not yet used theory driven constructs to understand the process of TDD, this study can contribute significantly to enhance our understanding of the phenomenon.

3. This study examines the claim that the TDD approach provides better software quality as compared to the traditional method of software development where testing is done after writing the code.

4. This study also empirically examines whether software developers learn more and are more satisfied with their performance when using the Test Driven approach as compared to the traditional method of software development.

5. This study also provides further evidence to the ongoing debate amongst researchers on whether programmers yield better solutions when working as collaborating pairs as compared to when working individually.

6. This study is also aimed to provide more insight into whether software developers learn more and feel more satisfied with their performance when they work as collaborating pairs as compared to working as individual programmers.

CHAPTER 2

LITERATURE REVIEW

This study has three core components, and the literature review focuses on each of these areas to identify issues or gaps that need to be addressed.

1. Software Engineering – Specifically, literature pertaining to agile methodologies, the Test First and Test Last approaches in software development, and performance issues such as the software quality and the task satisfaction.
2. Learning – Specifically, from a constructivist cognitive perspective.
3. Groups and Teams literature – Specifically, as applicable to pair programming in software development.

2.1 Software Engineering Literature

2.1.1 Software Development Process

Researchers have explained the process of software development using different models. With six distinct phases, Systems Development Life Cycle (SDLC) is generally agreed upon as an accepted way of modeling the process of software development (Blum 1994). As shown in Figure 2.1, SDLC has six distinct phases. The first phase,

requirement analysis, involves gathering information about the needs of the customer or client who will mostly be the end user of the developed product.

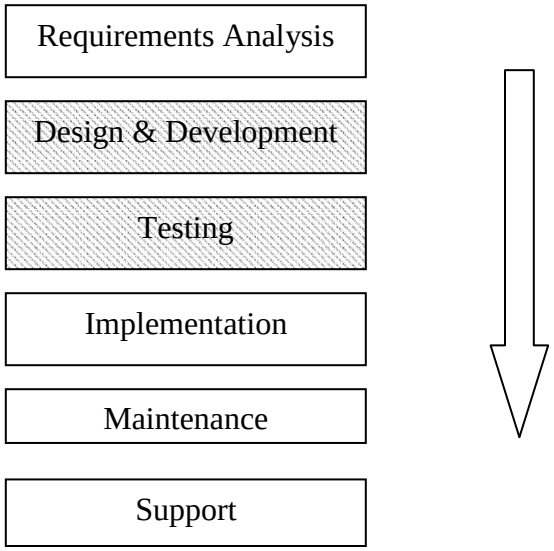


Figure 2.1 – System Development Life Cycle (SDLC)

The second phase deals with analysis of the requirements gathered at the first phase to come up with the problem definition and subsequent designing of the solution. The third phase of the software development is testing to check for errors in the code. Implementation of the solution forms the fourth phase. The fifth phase involves maintenance which is important in updating the systems so as to keep them robust and effective against any new developments. The final phase consists of customer services support that includes training the new users and helping the end user with technical issues.

This study deals with the phases of design and development and testing as shown by the shaded region in Figure 2.1. Research indicates that these phases are critically important phases of the Systems Development Life Cycle (Guindon 1990).

Generally, the 'Waterfall Model' is cited by researchers to depict a systematic sequential approach to software development. As shown in Figure 2.2, the process begins with requirements gathering where the emphasis is on identification of the needs of the client that the software solution will aim to fulfill. It is followed by planning where the emphasis is on estimation and scheduling. Planning is followed by modeling where the analysis and design are the critical activities. The construction phase focuses on coding and testing. Since this is a widely used, perhaps the oldest model, it is also referred to as the classical life cycle model (Pressman 2005).

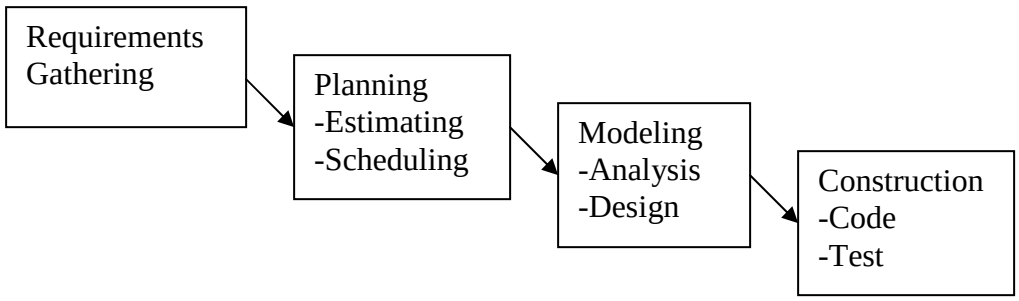


Figure 2.2 – The Waterfall Model
(adapted from Pressman, 2005 p.79)

Other models of software development like Incremental Process Model (McDermid, Rook 1993) adopt a similar approach but the cycle is repeated for each requirement and thus the software is developed in iterative manner. Different variants of the incremental software development model are being suggested such as Rapid Application

Development, Evolutionary Models, etc. In all the models suggested, the analysis and design phase invariably appears before the coding phase. Testing is always preceded by coding.

Much as with the traditional approach, modern approaches also follow the steps of planning, design, coding and then finally testing. But the modern agile methodologies emulate the incremental process of software development and the same sequence is followed iteratively. Extreme Programming uses the process as depicted in Figure 2.3. The software development process progresses incrementally, Figure 2.3 shows one such iteration. Upon testing, one such iteration is completed.

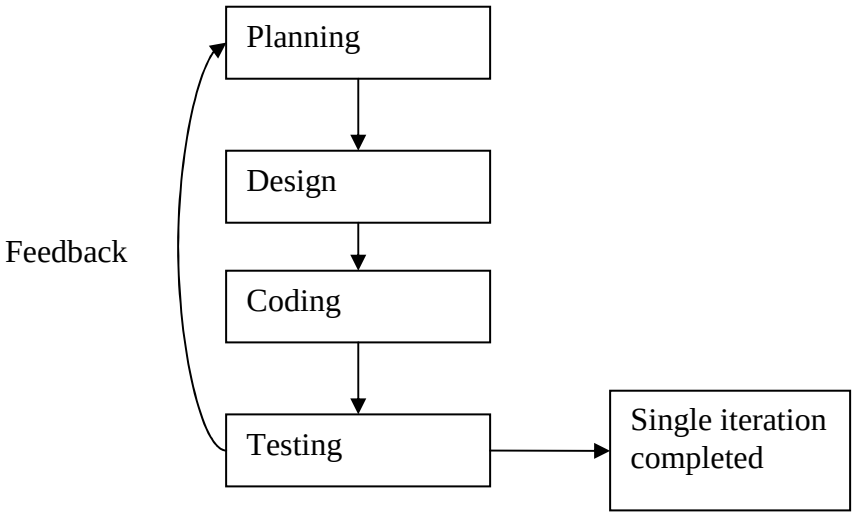


Figure 2.3 – Feedback Loop in Iterative Process of Extreme Programming

In all the software development process models, one common theme is that since the testing phase is always preceded by the coding phase, the developers are not able to get

any feedback on their codes until at least a single iteration is not completed. Figure 2.4 shows the feedback loop in the case of TDD process.

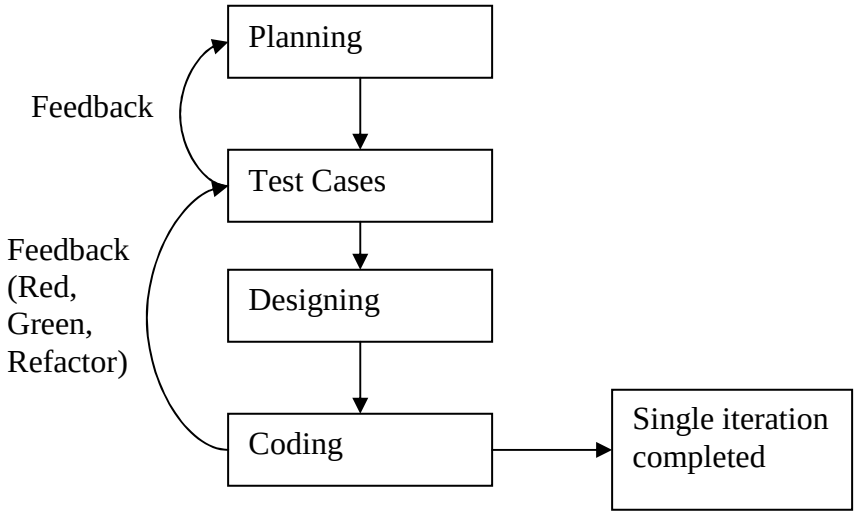


Figure 2.4 – Multiple Feedback loops in the TDD Process

2.1.2 Software Development Skills

Broadly speaking, there are two types of skill sets required for a software project - One is the technical skill set (computer related) and the other set is of functional skills (task related). Technical skills are more grounded in logic and algorithm while functional skills are more oriented towards abstract conceptual thinking with focus on real world business applications. The skills required for the design and development phase are different from those required at the testing phase. While the priority in the design and development phase is on the technical skills, in testing more emphasis is on the task related skills. It is argued that the role of software tester is much closer to the end user,

in the sense the tester has to look at the code from an end user's perspective with the view to find possible scenarios when the code could fail. Hence, both phases are important in the systems development process and they emphasize different skill sets. Consequently, in a typical traditional setting, the coder and the tester are two different people.

2.1.3 Test Driven Development (TDD)

Agile methodologies were developed in an attempt to simplify the process of software development and to make it more adaptable to the change in requirements. These new methodologies seem to be gaining wider acceptance in the software industry (Ambler 2006). Extreme Programming (XP) is one such methodology that uses pairs in place of solo programmers (Beck 1999). TDD is one of the core practices of XP that uses testing to guide the process of software development. TDD represents an evolution in the software development process, and it is being tried by the software industry as the new approach to code development (Jeffries, Melnik 2007).

In the traditional approach of writing software the sequence of the two steps remains as shown in SDLC graphic (Figure 1), when testing follows the design and development phase. However, in TDD the sequence is reversed as the developer now has to write testsuites even before writing the first line of code. The TDD approach forces the programmer to look beyond the algorithm and logic side and pay more attention on the functionalities and behavior of the code, even when not a single line of code is written.

Researchers claim that the hardest part of TDD is changing the way software developers think about writing the code (Hammell 2005).

This change translates into a paradigm shift. The role of the programmer is now completely reversed; now he/she is not thinking in terms of algorithm and structure; however, he/she has to conceptualize the situation more from the perspective of the end user. The focus therefore shifts from the algorithmic logic to the functional utility. This paradigmatic shift entails many benefits, which are enumerated as under and subsequently followed by discussion.

- Since the structure follows the functional utility, it yields a less imposing, more adaptable robust structure of the code.
- It yields a cleaner code as chances of bugs and errors are addressed right from the beginning.
- Programmers have a much better understanding of the behavior of their code.
- The resulting code is a tighter code as redundancies are reduced through continuous refactoring.
- Software written using the TDD approach is of a higher quality.
- It provides a better mechanism for more objective assessment of the programming progress.

- Test themselves can serve as documents that can help the developers to quickly recapture what they have been doing once they have been away from the project for a while.
- Codes written using TDD have much less computational complexity as compared to codes written using the conventional approach.
- TDD yields higher test volume and coverage.

Several researchers have studied TDD to verify the claims made by its proponents. Lui and Chan (2004) found that TDD greatly improved the process of software development. They claim that the TDD approach makes it easier for the developers to check if they stray from following the frameworks and guidelines. It also enables better task estimation and more objective progress tracking. Overall, this results in a better quality software.

Another study conducted by George and Williams (2004) involved pair programmers working in the industry, they found that though the initial productivity went down, the developers using TDD produced code of a higher external quality that passed 18% more functional black box tests as compared to the codes developed by using the traditional Test Last approach (George, Williams 2004). Janzen's (2005) finding also supports a higher external quality software. He found that the codes by professional developers written using TDD could pass from 18% to 50% more external tests than the codes written by the control group using the traditional approach. He also found that the

Test First projects had much lower computational complexity while having higher test volume and coverage as compared to the Test Last approach (Janzen, Saiedian 2005, Janzen, Saiedian 2006). Crispin (2006) also reports reduction in the defects rate by as much as 62% in the projects she worked on when using TDD (Crispin 2006).

TDD has also been studied when applied under different environments. In two case studies conducted at Microsoft, Bhat and Nagappan (2006) found that the TDD approach reduced the number of defects per KLOC (thousand lines of code) by almost four times, while the effort required went up by only 15%. Test coverage enhanced significantly by 88%, thus enhancing software quality.

Hence, the findings of different researchers remain mixed and there is no consensus among them on the effectiveness of TDD in Software Development. The results from different studies done in the area of TDD are tabulated as in Table 2.1.

Table 2.1 – Comparison of different studies done on TDD

Study	S/W Quality	Productivity	Setting
George et. al, 2004	+18%	-16%	Industry
Erdogmus et. al, 2005	No difference	+22%	Academic
Madeyski et. al, 2005	-25% to -45%	N/A	Academic
Edwards, 2004	+40%	-90%	Academic
Canfora et al, 2004	Inconclusive	-65%	Industry
Bhat & Nagappan 2006	+88%	-15%	Industry
Janzen et. al, 2006	+16%	+57%	Academic

2.1.4 Software Quality

Different researchers have elucidated different aspects of software quality. Traditional views on the quality as applied to manufacturing could not be applied to software products because of their peculiarities that do not find any parallels in manufacturing. Prahalad and Krishnan (1999) discuss a framework to judge software quality. They explain quality from the perspective of a customer and argue that the expectations of the customers from a software product vary widely depending upon the functionalities and features of the software. While a customer may be very precise in expecting how a spreadsheet should use a formula, for instance, he/she might be more forgiving in the layout of Graphic User Interface (GUI) components. Also, updates, patches, ease of

installation, ease of use, reliability, robustness etc. can be considered to be important indicators of quality of software (Prahalad & Krishnan 1999). This perspective on quality cannot be standardized as the customer needs are likely to vary widely, though it may provide important inputs towards evolving best practices.

Software quality can be judged from a technical perspective. The number of bugs reported is widely considered as a parameter to judge quality. Generally, errors are reported per KLOC (kilo, or thousand, lines of code). Errors can be found from Black Box testing techniques. There are white box testing techniques as well that provide insights about the reasons of problems in code execution. Research have also used inverse error rate, the ratio of software size and the number of unique problems identified by customer during acceptance testing (Ramasubbu et. al 2008). From a resource perspective, quality can be judged in terms of the amount of memory required to execute code, the amount of processing power needed, or the amount of CPU time used. Code optimization in terms of usage of resources is an important area of research, pursued in the field of Computer Science.

Software metrics have been used for a long time to judge software quality. Researchers have compared efficacy of metrics in codes written with different approaches. Subramanyam and Krishnan (2003) found that use of OO metrics derived from Chidamber and Kemerer Metrics gave consistent results in identifying defects in codes. Structural complexity has been investigated using the criteria of coupling and cohesion.

Coupling can be measured by finding Cyclomatic Complexity number that determines the dependence of one class on other classes in a program written using an Object Oriented approach (Gill & Kemerer 1991, Nurminen 2003). Findings indicate that both coupling and cohesion should be taken jointly rather than independently in complexity analysis (Darcy et al. 2005). Three metrics, the Chidamber and Kemerer (CK) metrics, Abreu's Metrics for Object-Oriented Design (MOOD), and Bansiya and Davis' Quality Metrics for Object-Oriented Design (QMOOD) for Object Oriented Programming have been examined in the case of agile software development methodologies (Olague et al. 2007).

Software quality is closely associated with productivity. Literature suggests productivity measured from a resource based perspective. Manpower is the critical resource to measure productivity and hours required by an individual software developer provide a good indicator of productivity from a project management perspective. Brook’s Law has long been discussed by the researchers to explain concepts related to productivity (McCain, Salvucci 2006).

Krishnan et al. (2000) have found a direct link between software development process and quality. Drawing parallels between manufacturing and software development they used a mathematical model to examine the effect of antecedent factors on quality and productivity in case of software development. They operationalized quality as the number of customer reported defects per thousand lines of code. They found that

among the factors such as personnel capability, product size, software process factors, and usage of tools, improvement in the development process can significantly improve the quality of the software products. They also pointed out the importance for further investigating the effect of personnel capability and software development process on productivity and quality of software products. Such a study would be a relevant area of research, given the widespread adoption of IT in businesses and the high growth in companies that provide IT solutions.

2.1.5 Task Satisfaction

Researchers have explained the concept of satisfaction in terms of fulfillment of expectations. It can be expressed as the difference between actual amounts of rewards received and the amount anticipated by workers (Robbins 1998). According to Locke (1976), job satisfaction is the "positive emotional response to a job resulting from attaining what the employee wants and values from the job" (Locke 1976). The emotional response is thus dependent upon an individual's appraisal of the task outcomes. The activity performed and its resultant outcome determines the perceived value from the task. The amount of time that the activity requires to get completed can alter the emotional response of the individual. The delay between the activity and finding the outcome can cause a change in the emotional state as well. The anticipation of rewards is also closely associated with feedback. As per Job Characteristics Theory, five task characteristics, namely, personal significance, variety, autonomy, feedback, and identity, drive satisfaction in the individual (Hackman et al. 1975).

The quantum of work that an activity takes and the feedback are thus the key drivers of satisfaction. Feedback is an important factor to consider in measuring the task satisfaction in case of software developers. As per the Flow theory (Csikszentmihalyi 1990), the feedback and clearly defined goals are imminent for experiencing flow. When experiencing flow, developers are likely to be oblivious to their surroundings and be highly engaged, and they may even lose track of time. Thus, it is argued that feedback allows people to experience flow and it also leads them to derive greater job satisfaction.

Apart from feedback, goal setting is also related to the satisfaction of an individual worker. The goals serve as a benchmark to judge amount of satisfaction that a worker would experience. Locke and Lantham (1990) argue that goal commitment and feedback moderate the relationship between goals and performance. The goal orientation is equally important as the employees with learning orientation will derive more satisfaction in solving complex tasks as compared to performance oriented employees who would prefer simple tasks (Steele-Johnson et al. 2000).

Employee satisfaction could greatly enhance when people work in groups. The individuals experience higher satisfaction and also produce greater quality work when working in groups (Campion, Medsker & Higgs 1993). Group members also have higher goal commitment and a more positive attitude towards goal attainment that leads to higher satisfaction (Hinsz & Nickell 2004). Also, some time individuals relate more

to the group performance than to their own, as group participation additionally fulfills their emotional and social needs (Levine, Moreland 1998), a fact that is well explained by the Social Identity Theory (Jetten, Spears & Postmes 2004).

Satisfaction and Performance have been perceived of as related but distinct measures in literature. The literature suggests that the most satisfied employees may not be the best performers (Iaffaldano, Muchinsky 1985, Judge et al. 2001, Petty, McGee & Cavender 1984). Therefore, it seems pertinent if the two constructs are measured separately in a study, as shown in Figure 2.5

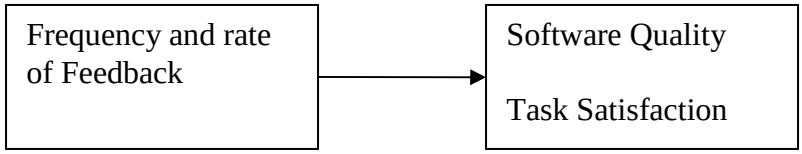


Figure 2.5 – Effect of Feedback on Software Quality and Task Satisfaction

2.2 Literature on Learning

It has been debated over many years now whether groups are better than the individual. Are two heads better than one? Researchers time and again find that two heads are not better than the best individual (Balijepally et. al 2009). Then, the question arises why the organizations go for teams at all? Are there more critical issues to examine than just measure performance?

One of the problems is that we have a too narrow definition of performance. The performance measured in the traditional sense of efficiency and effectiveness will not reflect the amount of learning that can be achieved by groups. The theories of learning are generally explained through three perspectives – Behaviorism, Cognitivism, and Constructivism. Behaviorism is grounded in Skinner's operant conditioning. The cognitivism stream of research argues that it is impossible to achieve learning purely on operant conditioning; there have to be some internal mental states to facilitate learning. This body of knowledge is more focused on information processing view at the cognitive level. The third stream of research, Constructivism combines the social aspects and is built on the foundation that truth can be discovered through social interaction. From the group and team literature, Constructivism seems to be the most relevant guidance to research. From the IS perspective, both Cognitivism and Constructivism hold promise to lead the researchers.

A discussion on different theoretical perspectives on learning that are relevant to the software development area follows.

2.2.1 Action Learning Perspective

Action learning or learning by doing is an important perspective for the field of software development. Different researchers have tried to define Action Learning, some of which are given below:

"An approach to working with, and developing, people that use working on an actual project, or problem, as a way to learn. Participants work in small groups to take action to solve their problem and learn how to learn from action. Often a learning coach works with the group in order to help the members learn how to balance their work with the learning from that work (Yorks, O'Neil & Marsick 1999, p.3)."

"Action Learning is both a process and a powerful program that involves a small group of people solving real problems while at the same time focusing on what they are learning and how their learning can benefit each group member and the organization as a whole (Marquardt 1999, p. 4)."

"Action Learning is a product not of teaching but of tackling problems to which there is no right answer and is about acquiring the ability to ask good questions of oneself, of others, and of situations which lead to an increased ability to tackle problems in the future. (Pedler 1991, p.63)"

Since software development is essentially a hands-on activity, Action Learning provides a useful framework. When action and reflection on those actions proceed almost simultaneously, double loop learning is said to occur (Nerur & Balijepally 2007). The term Experiential Learning can be used when both action and reflection are included in the learning process.

Argyris and Shon (1978) tied learning to Dewey's (1960) idea of inquiry. They claimed that both individuals and organizations learn through constant inquiry which is a cyclical process of questioning, data collection, reflection and action. In his later works, Kolb continued to build his Experiential Learning Theory (ELT) on similar lines of reasoning (Kolb 1976).

Gustavsen (2004), while enhancing the idea of action research, claims that making knowledge actionable entails transforming abstract knowledge into practical knowledge as applicable to the field (Gustavsen 2004). Essentially, as per Kolb's ELT, as more and more knowledge becomes actionable, it should lead to a greater learning. As per 'Experiential Learning Theory' perspective, learning has been theorized to occur in a continuous circular loop that has four distinct stages – Concrete Experience (CE), Reflective Observation (RO), Abstract Conceptualization (AC), and Active Experimentation (AE) (Kolb 1976).

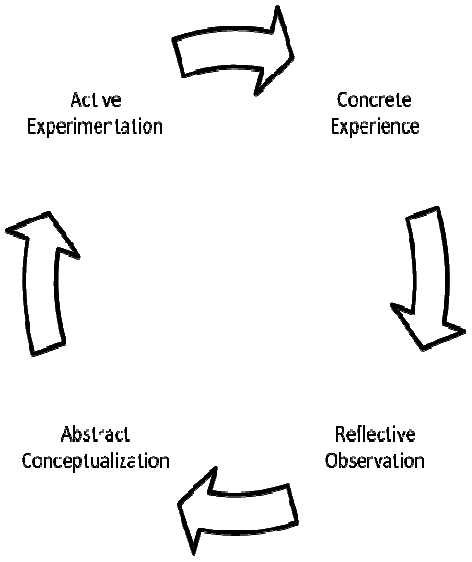


Figure 2.6 – The Experiential Learning Model
(adapted from Kolb, CMR, 1976)

As shown in Figure 2.6, the four stages yield twin sets of diametrically opposite parameters that define two dimensions. One dimension encompasses all events that range from concrete experiencing to abstract conceptualization. And the other dimension comprises of behavior ranging from active experimentation to reflective observation. Thus, learning is said to occur as the learner oscillates between the roles of an involved actor and a detached observer as he/she moves from specific instances to abstract generalizations. The learning cycle continues when these generalizations guide the decisions and actions to specific tasks.

2.2.2 Mindshift Learning Perspective

Another key argument in favor of TDD comes from the recently proposed theory named Mindshift Learning Theory (MLT) (Armstrong, Hardgrave 2007). The authors of MLT claim that when the perceived complexity and structure of the new knowledge are almost paradigmatically different from each other then it is more difficult for the learner to acquire new knowledge because there is a shift in the knowledge structure, as the learner has to spend more cognitive resources to build a new knowledge structure to assimilate new knowledge. The learning is impeded by 'proactive interference'.

In the traditional method of software development, software developers think in terms of variables, algorithms, and methods as they begin to write software. During the process of software development, they mostly analyze the task purely from technical perspective. Once the software is developed, it is being tested by testers. The testers test the software from a functional perspective since they try to identify flaws in the code from the usability viewpoint. In TDD, software developers are forced to adopt the functional perspective, in addition to the technical perspective, as they have to build unit testcases before writing the actual code. Software developers, while using TDD, therefore, apply both technical and functional perspectives and frequently shift between the two as they build code. The software developer still has to think in terms of data structures and algorithms but there is a need of constant switch in focus between technical and functionality perspectives.

As pointed out in the earlier section, the main difference between the traditional way of writing code and TDD is that in TDD, the developer gets feedback on the written code much earlier than in the traditional development. Nelson et. al (2009) claim that when learning a new methodology, the software developers learn incrementally in a series of cognitive shifts. The testcases are also written incrementally. The feedback cycle is short and quick feedback iterations increase the learning in the developer. In TDD, therefore, there is much greater learning as the action and reflection proceed simultaneously in testing and coding.

2.2.3 Flow Theory Perspective

Flow theory suggests that an individual experiences flow when the level of engagement is high and it leads to better performance outcomes. As per Flow theory, while performing certain tasks that people find interesting, they can be so deeply engaged that they become oblivious to the ambient conditions and may even lose track of time. They experience a heightened sense of enjoyment as a result of flow experience. People are motivated intrinsically and they feel that they are in total control. In such state of mind, they are not easily distracted. Flow has been expressed as ‘a peculiar dynamic state – the holistic sensation that people feel when they act in total involvement’ (Csikszentmihalyi 1990, p. 36).

The ‘flow experience’ has been characterized by four dimensions – focused concentration, perceived control, transformation of time, and autotelic experience.

Focused concentration has been explained as ‘a centering of attention on a limited stimulus field’ (Agarwal & Karahanna 2000, Csikszentmihalyi 1990, p. 40). Perceived control has been defined as ‘a sense that the outcomes of the activity are, in principle, under the person’s control’ (Csikszentmihalyi, 1990, p.33). Transformation of time is yet another dimension of Flow where the individual loses sense of track of time, principally because of heightened sense of enjoyment (Agarwal & Karahanna 2000). The motivation behind experience is entirely intrinsic, and it has therefore been described as an autotelic experience (Csikszentmihalyi 1990, p.67).

Flow experience has been theorized to be associated with some unique features relating to the activities themselves. Csikszentmihalyi (1975) argues that the activities that can cause flow should have three important characteristics – Goal clarity, feedback and a perceived balance of challenge and skill. Flow has been found to occur when people engage in sports (like playing chess, badminton, etc.) or gaming (like online games) (Delespaul, Reis & deVaries 2004, Madrigal 2006). In such activities, the participants have a clear goal to achieve, they get rapid feedback in unequivocal terms, and since people normally choose to engage in sports and games against opponents with similar level of competence (either human or a chosen level of difficulty in a computer game, for instance), there is a balance between challenge and skill.

Among software development methodologies, Test Driven Development stands out as a good candidate that can potentially lead to a flow experience. In TDD, the developer

has a clear unambiguous goal, before he or she begins to write code. In fact, the very first step of developing test suites forces the developer to first come up with a test that will fail. Finding a solution to the task thus becomes a clearly defined goal that the developer wants to achieve. The process of TDD is geared towards giving rapid feedback as the developer proceeds with the cycle of red, green and refactor. Red, Green and Refactor are the steps used in TDD, as shown in Table 2.2. Also, since the developer himself or herself decides the goal (test case), it is safe to assume that the challenge that he or she will set will be comparable to his or her skill level. Thus, it is posited that it is possible to achieve a balance between skill level and challenge by implementing the TDD process of software development. Hence, as a technique of developing code, TDD makes a good candidate for application of the flow theory. Application of Flow theory in the case of TDD should reveal interesting findings.

It is also claimed that Flow can promote learning outcomes as some people experience more joy as compared to others while learning (Asakawa & Csikszentmihalyi 1998). In the case of foreign language instruction, Flow theory has been found to provide a useful framework to evaluate learning, though the findings indicate that it is difficult to achieve Flow in classroom conditions (Egbert 2004). A recent study that examined impact of Flow on learning outcomes in case of a graduate level MIS course, Flow was found to directly affect perceived learning and student satisfaction (Guo et al. 2007).

Taking a cognitive perspective, learning is viewed as an active, goal oriented, and cumulative process. In a study, students were taught course material using the problem solving approach; the method was found to be superior to the traditional method of teaching as the students were found to be more engaged and they also learned better (Shuell 1990). Software development is an inherently task oriented activity that subsumes problem solving. In case of IT education, problem based learning has been claimed to be a suitable method of teaching computer application courses (Mykytyn 2007).

2.2.4 Self Determined Learning Theory Perspective

Argyris and Shon's (1978) theory of reflective learning is perhaps most suited to explain the software development teams using Agile methodologies (Nerur, Balijepally 2007). The developers get immediate feedback and it leads them to be able to continuously adjust so as to reach the solution. The Self Determined Learning Theory (SDLT) seems to provide a judicious explanation in case of pair programmers using TDD. SDLT can provide coherent explanations to the learning acquired by developers when they work on programming task using TDD. Briefly, opportunities for gain provoke engagement, engagement affects adjustments, and adjustments determine what learned (Mithaug et al. 2003) is. SDLT draws from both Cognitivism and Constructivism streams of literature. Since the developers write the test themselves, the opportunities for gain of knowledge are likely to be high. The feedback to a test when run is almost immediate, thus it is again likely to enhance the engagement. The

adjustment part is when the developers refactor the code. The parallels between the SDLT and TDD are as shown in Table 2.2. In the TDD process, red, green, and refactor are the standard steps used to denote the three stages. Eclipse IDE implements these steps and visually represents failing of unit tests as a red bar and passing of a unit test as a green bar. A different IDE could implement the same three steps in a different fashion, yet the steps of failing and passing of unit tests remains critical to implementation of unit tests.

Table 2.2 -Parallels between SDLT and TDD

Self-Determined Learning Theory (SDLT)	Test Driven Development (TDD)
Opportunities for gain of knowledge leads to engagement	Red – write unit test
Engagement leads to adjustment	Green – write enough code to pass test
Adjustment leads to learning	Refactor - cut redundancies, optimize

In the context of Software Development, in the traditional approach the feedback cycle is longer as compared to the TDD, as the developers are able to get a feedback only when they run the tests after writing the whole code. In case of TDD, the frequent tests provide quicker feedback with much shorter cycle time as the developers write codes (Erdogmus, Morisio & Torchiano 2005). There is more number of tests written in TDD as compared to the traditional approach thereby leading to a greater quantum of feedback in case of TDD.

Researchers have explored the relationship between feedback and learning. Butler and Winne (1995) came up with a synthesized model of Self Regulated Learning and Feedback cycle (Mithaug et al. 2003). Their model suggests that feedback leads to cognitive engagement with tasks and they examine the relation among forms of engagement and achievement.

It is posited that since in TDD more feedback is obtained in much shorter time, it is likely to lead to a greater cognitive engagement among the Software Developers in the task while using the TDD approach as compared to the traditional approach. This study focuses on the cognitive side of engagement rather than the psychological side of engagement for two reasons. First, software development is a cognition intensive task that demands more cognitive input from the software developer. Second, research in the field of social psychology suggests that there is no direct link between psychological engagement and learning. Hence, it is proposed to use Cognitive Engagement in the model. Since research indicates a direct relation between level of cognitive engagement and learning (Mithaug et al. 2003), it is posited that the TDD approach will lead to a higher learning among Software Developers as compared to the traditional approach. Theory driven model of learning is represented graphically in Figure 2.7.

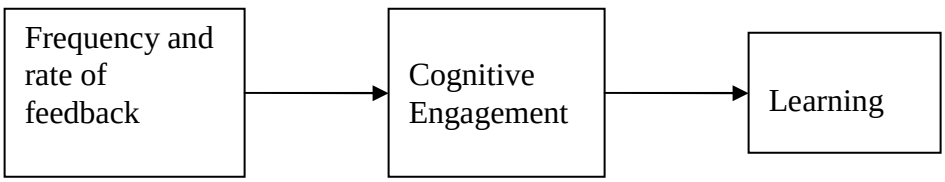


Figure 2.7 – Theory driven model of learning

2.3 Groups and Teams Literature

2.3.1 Level of analysis

Hackman (2003) suggests that conducting research across levels of analysis can increase our understanding of a phenomenon. Studying groups offers us the possibility to move to both more micro (individual) and macro (teams and groups) levels to explain process and outcomes as they unfold in groups. Reductionism is the philosophy that purports that to study the component parts of a phenomenon will lead us to understand the phenomenon. However, it may not always be possible to understand phenomena by observing them only through Reductionism. Bracketing either up or down the observational level has the advantage of revealing more insights into a phenomenon. It is claimed that bracketing reveals hidden variance, interaction across levels, and while enriching our understanding, it can give us better choice of concepts in developing actionable theory (Hackman 2003). By choosing the individual versus pairs as levels of analysis, this study strives to achieve a better understanding of the process of TDD as levels of analysis change.

2.3.2 Task characteristics based perspective

McGrath (1984) described different types of tasks in his study on task typology. Based upon psychological dimension (from cognitive to behavioral) and coordination dimension (from collaboration to conflict resolution), the eight types of tasks identified include Creativity tasks, Intellective tasks, Judgment tasks, Cognitive conflict tasks,

Mixed motive tasks, Contests / battles, Psychomotor tasks, and Planning tasks (McGrath 1984). For the field of software engineering, intellective and judgment tasks are of special interest. Intellective tasks require a high amount of information processing; they are normally governed by some specific set of rules or guidelines that enable the individual or group to follow some specified and accepted procedures to arrive at a definitive solution. The objectivity with which the task solution can be found makes these tasks stand out from other tasks. Judgmental tasks, on the other hand, require evaluation of the various options to arrive at an answer through group consensus. A definitive and universally constant answer may be difficult to find for such tasks. There is much scope for debate and arguments among group members before the group can reach a commonly agreeable solution (Laughlin & Ellis 1986).

When individuals work in groups, it becomes important to consider the task characteristics since putting individuals together as groups may not always result in increased productivity (Levi 2001). Some of the tasks could be more amenable to be performed by groups, while others could be better performed by individuals. Tasks that can be divided into subparts are termed as divisible tasks, while those that cannot be divided into subparts are termed as unitary tasks (Steiner 1972).

The tasks can also be classified as disjunctive and conjunctive tasks. Disjunctive tasks are divisible into subtasks and all group members can contribute by working on a subtask. The final solution would be an aggregation of the subtasks. Conjunctive tasks,

on the other hand, are not amenable to easy subdivision and the members have to debate and analyze until group consensus is reached (Balijepally et. al 2009).

A programming task is inherently similar to a problem solving task. Programming tasks are difficult to divide and the quality of output is emphasized over speed (Balijepally et. al 2009). When pairs work on a programming task, it becomes a challenge to deal with the indivisibility. The programmers have to communicate and discuss with each other frequently to come up with the solution. Finding solution to a programming task is an intellectually intensive process. Developers write code to provide useful functionality by applying logic and following structural rules governed by the programming language. Each developer can write the code that would provide the same functionality in multiple ways. Hence, there are unique characteristics of the tasks that influence the group performance.

2.3.3 Motivational Loss Perspective

Some researchers also found reduction in motivation when individuals work in teams, a phenomenon termed as Social Loafing. Karau & Williams (1993) came up with Collective Effort Model based (CEM) on Social Impact theory (The amount of social impact experienced in a situation is thought to be a function of the strength, immediacy, and number of sources and targets present). Collective Effort Model is a cognitive model based on the expectancy – value model of effort extended to groups. Researchers believe that social loafing occurs because there is usually a stronger perceived

contingency between individual effort and valued outcomes when working individually. When working collectively, factors other than the individual's effort frequently determine performance, and valued outcomes are often divided among all of the group members. Key factors contributing to the loss of motivation (social loafing) are arousal reduction, reduced evaluation potential, dispensability of effort, matching of effort, and self attention (Karau & Williams 1993). Arousal reduction has been explained from social impact theory perspective as drive reducing, thus leading to social loafing. For simple tasks, group members suffered loss in drive and reduced their effort, however, for novel difficult tasks, the group performance improved significantly (Jackson & Williams 1985). Evaluation potential refers to the loss in motivation when members of the group perceive that their individual performance evaluation cannot be easily and accurately ascertained. Group members would exhibit social loafing by hiding in the crowd (Harkins & Jackson 1985). Dispensability of effort is observed when the members of the group feel that their individual efforts are not going to lead to any significant contributions to the higher quality group output. In other words, social loafing is observed because of the dispensability of the group members (Kerr 1983, Kerr & Bruun 1983). Matching of effort is based upon the presumption that members of the group expect others to slack off, and to maintain equity, they reduce their own efforts (Jackson & Harkins 1985). Self attention refers to reduction is self awareness when people work collectively. From the self attention perspective, the self regulation and the attentiveness to the task details among members is claimed to be reduced

(Mullen 1983). However, studies on self attention have indicated mixed findings (Stevenson 1990).

Laughlin et. al (2003) argue that groups can perform better than individuals if strategies, operations, and procedures of problem solving are highly demonstrable. TDD provides a mechanism of building unit tests. In this study, it is argued that the process of building and running unit testcases enhances the demonstrability. In groups, demonstrability can reduce conflict and encourage participation. In the case of individual programmers, demonstrability can enhance engagement. But in the case of pair programmers, demonstrability can, additionally, reduce the loss of motivation. Therefore, it is argued that, because of two important reasons, TDD is a better method for software development when programmers work in teams. One, it provides an objective evaluative criterion to judge performance, thereby attenuating effect of the social loafing. And secondly, since the response (feedback) is immediate, the chances of perceived discrepancy in the individual effort and valued outcomes are minimized.

2.3.4 Motivational Gain Perspective

On the other hand, some researcher found motivational gains when the individuals were put in teams. Hertel and Messe (2000) have investigated the Kohler effect which propounds motivational gains when working in teams. Kohler found that team members working together did better at a persistently taxing task than what would be expected from their individual performances, particularly when there was a moderate

discrepancy in coworkers' capabilities. However, he group members can fail to identify the resources of the group and that can lead to deterioration in group performance (Kerr & Tindale, 2004).

Identifying with the group can also lead to better participation. The Social Identity has been perceived as glue that holds the group together and leads to group loyalty (Van Vugt & Hart 2004). Social Identity Theory has been used to extend the motivation literature to a collective perspective at group level. According to SIT, people tend to classify themselves and others into various social categories defined by prototypical characteristics abstracted from the members (Tajfel 1978).Traditionally, researchers consider work motivation as a separate entity and motivation to achieve a collective performance as derived from individual concerns and motives. Such approaches can help predict the group behavior and outcomes using Social Identity an intrinsic source of motivation (Ellemers, De Gilder & Haslam 2004). Ashforth and Mael (1989) show how the SIT can be applied in organizational socialization, role conflict and inter-group relations. Collective Identity has been articulated while exploring its various dimensions and a framework was developed which captures the multidimensional concept of collective/social identity (Ashmore, Deaux & McLaughlin-Volpe 2004). In real world, software development is accomplished when a number of software developers work in small groups (or as teams) closely coordinating with each other. Hence, Social Identity from a collective perspective leading to enhancement in intrinsic motivation in teams can be important for pair programmers.

In Hertel and Messe experiment, there seems to be competitive (survival) group cohesion on some activity matching to what the members were experts at (Hertel, Kerr & Messé 2000). Knowledge of the partners' abilities has also been investigated as a moderator in Kohler's effect (Messé et al. 2002). With specific case of computer related tasks, instrumentality of an individual in teams in bringing about success in the project has been identified as major factor of intrinsic motivational gains (Hertel, Deter & Konradt 2003). Instrumentality has been described as the individual's contribution was instrumental to the dyad's success.

TDD requires the developers to write unit tests before they begin to write code. The unit tests are run, and they are expected to fail the first time. However, when the pairs work on developing enough code so as to prevent the unit tests from failing, the team members can get a fairly good idea about the instrumentality of the team member in bringing about the success. When tests are re-run, and the tests pass, it provides a high degree of demonstrability to the team. It is, therefore, argued that TDD provides a better mechanism, through better demonstrability, to judge the instrumentality in bringing about a positive change in the software project; hence it is expected that pair programmers using TDD should experience motivational gains.

2.3.5 Cooperation Perspective

Another important factor that needs to be delved into deeper is Team cooperation. Levine and Thompson (1996) discussed intra-group conflict and elucidated mechanisms of Conflict avoidance, Conflict reduction, and Conflict creation that can lead to cooperation. Using the Social Identity lens, Cremer & Tyler (2005) point out that fair procedure can lead to better concept of self thereby motivating members and thus eliciting co-operation. In another paper, they talked about the process based model of leadership where they found that a leader can be successful in actuating an organizational change by taking steps that are judged procedurally fair (Tyler, De Cremer 2005). In a recent paper, the researchers point out that trust in leader moderates this relationship (De Cremer, Tyler 2007). In case of software development, since the programmers use a structured method of writing code, there are less chances of having any conflict on the grounds of not following the fair procedures. However, trust can be a key factor in team effectiveness. Demonstrability of the fairness and effectiveness of a procedure can be important for the mutual gain in trust among the pair programmers. All these findings potentially bear important implications for pairs developing code using TDD. Pair programming is an essential feature of agile methodologies of software development. Since, TDD is one of the agile methodologies; it is suited to be implemented using pair programming approach, though it can be implemented by an individual programmer as well. On the basis of the preceding discussions, it can be argued that the TDD technique, when used in software development, mitigates some of

the negative aspects pair programming and enhances some of the positive aspects of pairing. For instance, instrumentality helps in motivational gains while demonstrability can help in conflict resolution among pair programmers. Therefore, the teams comprising of two programmers have been claimed to enhance software development outcomes, as shown in Figure 2.8.

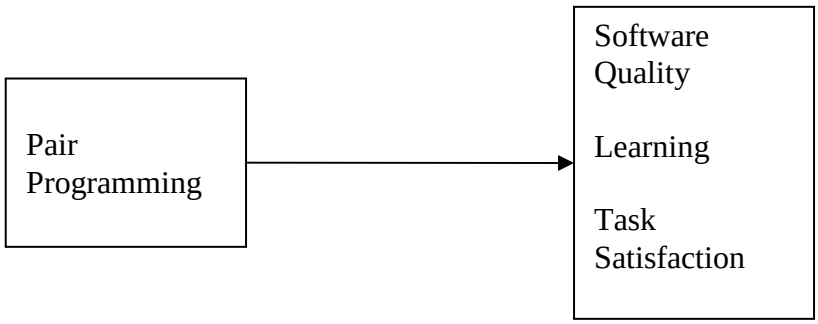


Figure 2.8 – Pair Programming and Outcomes of Software Development Process

CHAPTER 3

RESEARCH MODEL AND HYPOTHESES DEVELOPMENT

The main focus of this study is to investigate the effectiveness of TDD as compared to the traditional method of writing software. Also, this study also investigates, while using the TDD methodology, the change in the level of effectiveness when collaborating pairs write software as compared with individual programmers. This study also examines, using process variables drawn from theory, whether one software development methodology results in better learning and higher task satisfaction than the other in the scenarios of individual programmers and collaborating pairs.

On the basis of literature review discussed in the previous chapter, this study identifies problems as described in the following section. In subsequent sections, the research model is developed and different hypotheses are proposed. In order to examine the veracity of the conceptual model, hypotheses will be tested, as described in the following chapter.

3.1 Research Model

Based on an extensive literature review, this study derives a relationship between Feedback, Cognitive Engagement and Learning as discussed in the previous chapter. It was shown that the level of Cognitive Engagement will directly affects the frequency

and the rate of feedback. Cognitive Engagement will directly affect learning – higher cognitive engagement will lead to higher learning. One of the key differences between the Traditional approach of software development and TDD is that the feedback cycle is much shorter in TDD. Using the traditional approach, the developers first write the code and then after writing a large part of, or the complete code, they proceed with testing. Typically, a different individual writes tests in order to check the completed code for bugs. In case of TDD, the tests were written before the code was written, and more importantly, it was done by the same developer who eventually writes the code. The process of Red, Green and Refactor provides a mechanism of quick feedback. When pair programmers use TDD, their performance is likely to be enhanced as compared to the individual programmers using TDD. Instrumentality of a developer in bringing about success to the project can lead to the motivational gain in the other member of the group. TDD provides an effective means of demonstrability that helps the pairs in resolving any conflict. Demonstrability also reduces social loafing and improves the group effectiveness. Figure 3.1 depicts the model that develops from theory. The details of the Research Model for this study are shown in Figure 3.2 and Figure 3.3.

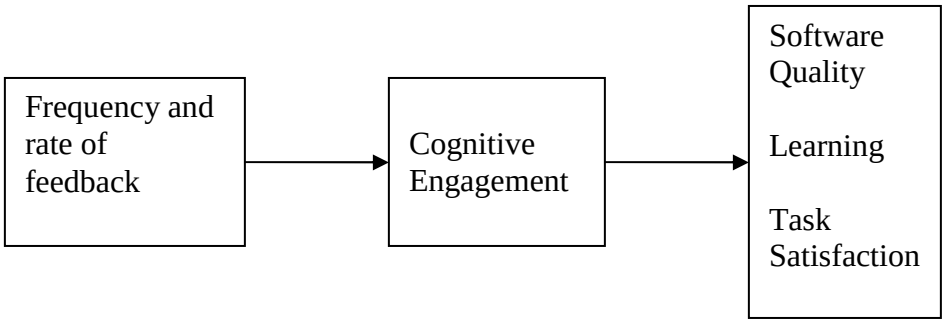


Figure 3.1 – Theory driven model

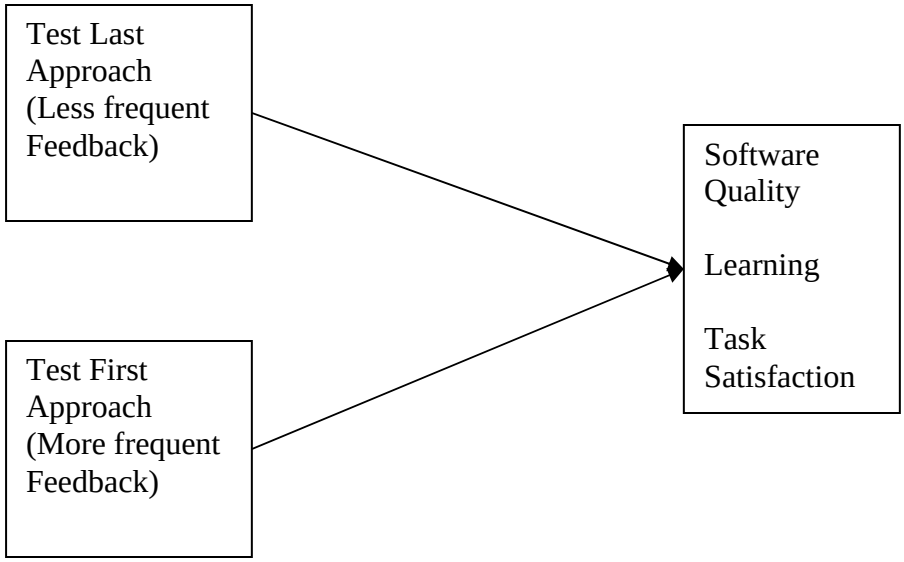


Figure 3.2 – Research Model (Traditional & TDD)

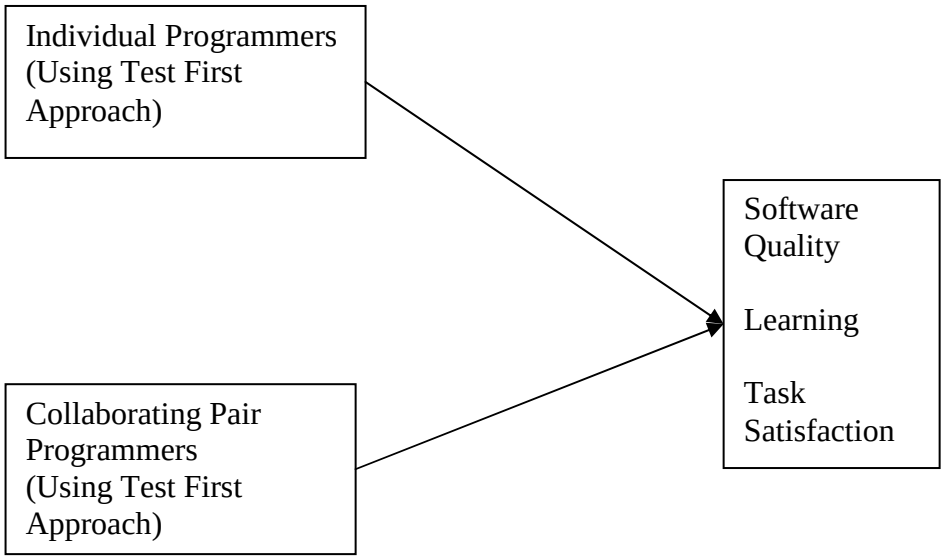


Figure 3.3 – Research Model (Individuals & Pairs)

The model was tested in two scenarios - one was that of individual developers and the other was that of pair programmers.

3.2 Impact of the Software Development Methodology on Software Outcomes

In the traditional software development method, as discussed earlier, coding precedes testing, while in the TDD technique, testing precedes coding. TDD forces the software developers to build testcases before they begin to write software. The software developer iteratively uses the cycle of Red, Green, and Refactor. From systems design perspective, using the TDD technique, the testcases are written before the phases of designing and coding. When the tests are run, the developer gets an immediate

feedback on the behavior of the code. This early feedback is the valuable information that can be quickly incorporated in the planning phase. Some IT organizations implement TDD through concurrent coding and testing. Summarily, the unit testing and the TDD process can be argued to be an improvement in the software development process. Among many other factors, researchers have claimed that the quality of software products is positively influenced by the improvement in the development process and usage of tools (Krishnan et. al 2000).

In the TDD technique, the process of designing and coding is directed towards writing code in small bits. Incremental code development enables the software developers to focus on one aspect of code at time. Unit testing helps the developers in quick identification of the errors in the written code. TDD can, therefore, be argued to prevent the software developers from committing errors while coding. Researchers have expressed the quality of software in terms of errors in the code. Errors can be considered per thousand lines of code, sometime reverse error ratio has also been used (Ramasubbu et. al 2008). One of the ways of ascertaining the software quality is by testing the code for errors. The code that is developed using TDD, in theory, is fully tested and is free of errors. Hence, it is argued that TDD is an improved process of software development that leads to a better quality of code. The specific hypothesis is given as under:

H1: While working on a programming task, individual programmers using TDD perform better in terms of Software Quality as compared to those using the traditional method of software development.

Feedback is one of the antecedents of flow experience (Csikszentmihalyi 1975). Flow has been found to be explained by Temporal Dissociation, Enjoyment and Focused Immersion (Agarwal & Karahanna 2000). Prior research suggests that feedback is the key factor that directly affects Cognitive Engagement. In their study on Computer Mediated Environments, Hoffman and Novak (1996) found that the feedback is the key antecedent to engaging online customers. Based on the literature review, it is argued that since feedback will be greater in the case of TDD, the developer would experience greater cognitive engagement. Nelson et. al (2009) claim that when learning a new method of software development, developers learn in small steps, through a series of cognitive shifts. It is argued that cognitive engagement enhances the effects of mind shifts thereby leading to greater learning.

This study focuses on three levels of learning – the ability to recall verbatim, the ability to comprehend, and the ability to solve problems. The ability of programmers to recall key words or key concepts, that they were expected to learn while working on the programming task, was considered as the first level of learning. The first level of learning was labeled as Verbatim Recall. The ability to understand the key attributes, and the inter-relationships between different classes, objects, and methods used in the

code solution, is termed as comprehension. Comprehension was considered as the second level of learning. The ability to apply the knowledge, gained while working on the programming task, in a new scenario, was considered as indicative of the problem solving ability of the programmer. Problem solving ability was considered as the third level of learning.

As shown earlier, while working on the programming task, the programmers using TDD would be more engaged than those using the traditional method of software development. It is, therefore, argued that they will have relatively better memory retention of the key words and key concepts as compared to the programmers who used the traditional method of software development. Hence, the participants using the TDD approach are hypothesized to have better verbatim recall than those who used the traditional approach of software development.

The participants who used the traditional method of software development did not get rapid feedback based upon the output from running the JUnit tests. Since, the software developers themselves write JUnit testcases, either concurrently or before coding, the TDD approach enables them to have a better comprehension of the problem. In other words, JUnit testcases enable the software developers to get immediate feedback when they are building functionalities. When testcases are run, the developers get a clear idea of the functionalities that their code will provide. It improves their comprehension about the programming task requirements. Building Junit testcases also goads them to

comprehend the problem better because they are constrained by the technique to look at the problem from multiple perspectives. The other perspectives that the software developer adopts include those of the tester and the end user. As per Experiential Learning theory, the software developer frequently moves between the two roles of an involved actor and a detached observer as he/she moves from specific instances to abstract generalizations (Kolb 1976). While looking at it from multiple perspectives, the software developer appreciates the nuances of the programming task problem that eventually enriches his or her comprehension of the task. Hence, it is hypothesized that the software developers using the TDD approach would comprehend the task problem better than those who use the traditional method of software development.

The third and final measure of learning for this study is the problem solving ability of the software developers. The problem solving ability was expressed as the ability to apply the knowledge, gained from working on the programming task, in a new context. While working on the programming task, the software developers were expected to learn about the structure and behavior of the code. They were expected to gain knowledge about the specific classes and the associated methods with those classes. While developing the solution to the programming task using TDD, the software developers get an opportunity look at the solution code from multiple perspectives, as discussed before, they can be expected to have a better cognitive model of the solution as compared to those developers who worked on the programming task using the traditional method of software development. It is argued that the developers, who used

TDD, are likely to apply the knowledge gained in a different context much better than those who used the traditional method of software development. Hence, the experiment participants who used the TDD approach, while developing the code for the main programming task, are argued to acquire better problem solving ability as compared to those who used the traditional method of software development. The specific hypotheses, that follow, are focused upon each of the three levels of learning.

H2a: While working on a programming task, individual programmers using TDD perform better in terms of verbatim recall as compared to those using the traditional method of software development.

H2b: While working on a programming task, individual programmers using TDD perform better in terms of comprehension as compared to those using the traditional method of software development.

H2c: While working on a programming task, individual programmers using TDD acquire higher problem solving ability as compared to those using the traditional method of software development.

Overall Task Satisfaction is explained by researchers in terms of fulfillment of expectations. Robbins (1998) expresses it as the difference between the anticipated and actual rewards. The anticipated rewards are based upon the individual's appraisal of the

task outcomes. It is important to keep in mind that this anticipation can change over a period of time. If there is delay in the actual reward, it might attenuate the feeling of fulfillment of expectations. Actual reward is closely associated with the feedback. Feedback has been considered by researchers as a task characteristic that leads to satisfaction in an individual (Hackman et al, 1975). The TDD technique requires the developers to run the unit testcases and write code incrementally. The developer run the testcases, they get immediate feedback. Apart from feedback, goal setting has also been claimed to influence satisfaction (Locke & Lantham 1990). Through constant feedback, the TDD technique also helps the developers in setting realistic goals. As the coding progresses, the developers can also adjust their goals. Testcases also provide a mechanism to measure progress in code development. By adopting the TDD approach, clear goals and rapid feedback would help the individual programmers achieve a higher overall task satisfaction as compared to those who used the traditional method of software development. Therefore it is hypothesized –

H3: While working on a programming task, overall task satisfaction of the individual programmers using TDD is higher than the overall task satisfaction of those using the traditional method of software development.

3.3 Individual versus Pairs in TDD

The effectiveness of the pair programmers is explored in the context of TDD in this section.

Several previous studies, performed in the academic settings, have shown that the software quality is enhanced when the software developers work as collaborating pairs (Edwards 2004, Janzen et. al 2006). The similar findings in the industry settings further bolster the claim that pair programmers, as compared to the individual programmers, develop the code of a higher quality (Bhat & Nagappan 2006, George et. al 2004). Laughlin et. al (2003) found groups to perform better than the individuals on problem solving. Hill (1982) claimed that performance in groups can be compared to the performance of the individual. She found support for Steiner's theory of process loss (Steiner & Rajaratnam 1961), but also found some evidence for process gain. Her findings indicated that the group performance is rarely better than the individual performance. She advocated a case for randomization to avoid confounding of the group conditions and the subject variables. In IS research, prior studies (Balijepally et. al 2009, Managalaraj 2007) did not find group performance better than the best individual in the nominal pair. Group performance was, however, found to be better than the second best individual of the nominal pair. Following the body of research in groups and recent findings in software development, it is not expected in this study that the pairs would outperform the best individual, but pairs are expected to outperform the

second best individual in the nominal pair. In this study, hypotheses are developed on the basis of conservative estimates. Hence, group performance would be compared with the second best individual in the nominal pair.

Performance effectiveness of pairs has been investigated by researchers using different theoretical perspectives. For this study, Motivational gain and Motivational loss perspectives are quite relevant. From motivational gain perspective, the research points towards gains in intrinsic motivation because of the instrumentality in bringing about success in problem solving (Hertel et al 2003). For persistently taxing tasks, Kohler effect is observed and groups exhibit motivational gains (Hertel & Messe 2003). As per the Social Facilitation perspective, participation of the software developer in pair condition will depend upon how much he or she identifies with the group. Research points out that Social Identity can be perceived of as the social glue that holds the group together (Van Vugt & De Cremer 1999).

Social Loafing literature suggests that when people are put in groups, it may lead to motivational losses and individuals would put in less than their best efforts in group condition (Harkins & Petty 1982). However, there are two important issues in case of software development scenario. First, the size of the group in case of pair programming is two, which is the smallest possible size of a physical group. Social Impact Theory (Latane 1981) predicts that social loafing varies directly with the group size. Since the smallest possible group size is used in this study, it is believed that the effect of social

loafing in case of pair programming would be minimal. Second, there is much less chance of social loafing when the task is more engaging (Brickner, Harkins & Ostrom 1986). In case of TDD, the developers get rapid feedback that leads to a higher cognitive engagement, as discussed in the previous section. Higher cognitive engagement thus reduces the chances of social loafing.

The software development is a unitary, intellective, and knowledge intensive task. The pair programmers, therefore, can not easily divide the task into subtasks. They have to proactively come up with scenarios of failure that will help them build unit testcases. For the programmers in pair condition, since a joint solution needs to be developed, the programmers have to discuss and debate each idea before they can develop code. It might lead to a conflict situation. Demonstrability can help resolve conflict in groups. The specific hypothesis is –

H4: While working on a programming task using TDD, performance of a collaborating pair in terms of software quality is higher than the performance of the second-best programmer in a nominal pair.

Learning in groups occurs through the activities in which individuals, acquire, share and combine knowledge by working mutually on a task (Argote et. al 2001). It has also been expressed as “an ongoing process of reflection and action, characterized by asking questions, seeking feedback, experimenting, reflecting on results and discussing errors”

(Edmondson 2002). The basic processes in group learning have been identified as sharing, storage, and retrieval (Wilson et. al 2007). In the pair condition, software developers get an opportunity to share knowledge and skills. Double loop learning can be argued to get enhanced since the pair condition provides a sounding board to the developers and they get an opportunity to review each other's ideas while learning from each other.

Similar to the last section, the learning has been measured in this study using three measures – verbatim recall, comprehension, and problem solving. Programmers in pair condition are likely to discuss the programming task and exchange their thoughts frequently using the terms and keywords given in the problem description. Programmers in the individual condition also look at the same terms and keywords but they do not have a need, or an opportunity, to vocalize these terms and keywords. Vocalization can aid in better memory retention. When the programmers would vocalize the keywords and terms, there might be a possibility that their partner might vocalize the same terms a bit differently, it is argued that it will enhance the cognitive engagement of the listener. Also, similar ideas might be expressed a bit differently by the programmers in the pair condition. When the developer listen to an idea (very similar to their own) expressed differently, it is again argued to enhance their cognitive engagement. As per Self Determined Learning theory (Mithaug et. al 2003), cognitive engagement leads to adjustments in the knowledge structures of the individual and that leads to learning. As the developers proceed with code development using TDD,

building and running testcases helps them in learning. Recent research suggests that learning takes place in small steps, through a series of shifts in knowledge structures (Nelson et. al 2009). Therefore, it is argued that they are likely to show higher memory retention as compared to those who do not have the advantage of discussing their thoughts with a colleague while working on a programming task.

A second measure for the dependent measure of Learning is comprehension. Comprehension denotes the overall understanding of the programmers about the assigned main task. When the programmers in group condition work on developing the solution, as per the task description given to them, they discuss and come up with the solution jointly. The programmers have the benefit of thinking aloud and get an immediate and additional feedback, which might be a differing view from their own. A developer adopts two different types of roles in case of pair programming. One is that of involved actor and the other is that of detached observer. For example, when a programmers builds a testcase, he or she is being an involved actor, while when the same programmer reviews the testcase built by the other programmer, he or she is being a detached observer. Kolb (1976), as per Experiential Learning theory, claims that constant oscillation of roles between being an involved actor and being a detached observer promotes learning. The pair programmers not only get to appreciate a different perspective, healthy debate and discussion is likely to enhance their overall comprehension of the programming task. Sharing of knowledge is argued to lead to a better understanding of the interrelationships between different objects and methods

used in developing the solution code. Since collaboration has been argued to enhance understanding, it is hypothesized that the collaborating pair will outperform the second best in the nominal pair in terms of comprehension.

The third measure for the dependent variable of learning is problem solving. For this study, problem solving has been considered as the ability to apply knowledge in a new context. As discussed before, when the participants work on a programming task, they are likely to learn about objects and attributes, and the relationships between different objects. When programmers work as collaborating pairs, they build solution through mutual discussion frequently exchanging ideas. The process of identifying solution mutually would require them to look at the problem from the other developer's perspective. It is argued that mutual collaboration would help them discover the relationships between objects thereby enhancing their problem solving ability.

The TDD method enables the developers to get rapid feedback. The pairs are able to discuss frequently since TDD method provides clearly defined objective goal by means of JUnit test cases. TDD enables quick feedback, experimentation, and reflection on results. However, individual programmers using TDD do not have the opportunity to discuss the problem with another developer. When using TDD in the pair condition, the developer can get an opportunity to discuss the problem with another developer. The discussion would help them in better memory retention, comprehension, and problem

solving. Therefore it is argued and hypothesized that the pairs would exhibit more learning as compared to individual programmers.

H5a: While working on a programming task using TDD, performance of the best programmer in the collaborating pair in terms of verbatim recall is higher than the performance of the second best programmer in a nominal pair.

H5b: While working on a programming task using TDD, performance of the best programmer in the collaborating pair in terms of comprehension is higher than the performance of the second best programmer in a nominal pair.

H5c: While working on a programming task using TDD, performance of the best programmer in the collaborating pair in terms of problem solving is higher than the performance of the second best programmer in a nominal pair.

Task satisfaction has been found to relate positively to the group condition. Individuals working in groups report higher satisfaction (Campion et.al 1993). When working in groups, people have been found to enjoy more than when they work individually (Garibaldi 1979). Programming tasks are similar to problem solving unitary tasks. Unitary tasks are difficult to divide into subtasks. When programmers work as pairs on the tasks, it requires them to work on the same aspect of the code and discuss among

themselves to find the solution. The constant discussion among programmers in pair condition can lead to conflict, if the developers have differing ideas.

To elicit intra-group cooperation, mechanisms of conflict reduction have been suggested by researchers (Levine & Thompson 1996). Task demonstrability can potentially reduce conflict. When developers use TDD in group condition, the unit testcases provide a handy mechanism for task demonstrability. Constant interaction over issues and errors not only provides them professional insights that lead to better performance outcomes, it also fulfills their emotional and social needs (Levine, Moreland 1998).

In the experiment, the pair programmers were aware of the condition that they had to come up with one joint solution and they will be judged individually for performance. It should have alleviated the feeling of competition and enhance the feeling of cooperation among the members of the team. Research studies show a moderate relationship between performance and task satisfaction (Judge et. al 2001). As per Social Identity theory, when programmers identify themselves with the group, it can lead to higher self esteem. When groups use TDD, it allows them to set and adjust their goals. From the goal setting perspective, realistic goal setting can lead to higher task satisfaction.

Prior research shows that the pair programmers were more satisfied the individual programmers on a complex programming task (Balijepally et. al 2009). Therefore, from a conservative estimate, it is hypothesized in this study that the best programmer in the collaborating pair would experience higher task satisfaction as compared to the second best individual programmer in nominal pair.

H6: While working on a programming task using TDD, overall task satisfaction of the best programmer in the collaborating pair is higher than overall satisfaction of the second best programmer in a nominal pair.

CHAPTER 4

RESEARCH METHODOLOGY

The details about methodology, laboratory setting, sample size considerations, manipulation checks and measurement considerations are discussed in this section.

4.1 Methodology

This study was carried out using the experimental design approach. A laboratory experiment was conducted with the university students as the subjects to test the hypotheses under consideration. The choice of laboratory set up was made to achieve more precision in study by controlling for extraneous factors that might confound the results. Experiments enable researchers to establish causality with a higher degree of confidence in the findings and inferences.

This study attempted to compare two software development methodologies. The participants developed solutions to the assigned programming task, using either of the two approaches:

1. Traditional Software Development.
2. TDD.

The participants using TDD could be in either of the two settings:

1. Individual Programmers

2. Collaborating Pair Programmers

Participants were given a warm up task before the main programming task. Three dependent variables were measured:

1. Software Quality

2. Learning

3. Task Satisfaction

Laptop computers with Eclipse IDE (Integrated Development Environment) were provided to all the participants. JUnit testcases were enabled only in those machines that were used by the participants who had to build code using TDD. The participants could access the JAVA API that Eclipse IDE provides. However, they were not provided access to the Internet to prevent anyone from searching for any probable solutions to the assigned tasks. Data pertaining to demographics, gender, years of software development experience, years of object-oriented programming and design experience were collected using questionnaire prior to the experimental procedures. Data regarding dependent measures of Learning and Task Satisfaction were also collected using questionnaire that was given to the students after they had completed the experimental programming task.

4.2 Subjects

The target participants for this study were undergraduate and graduate students who were enrolled in a course on object oriented programming or systems analysis and design. The participation in the experiment was voluntary, and to encourage participation, the students were given extra credit by instructors for participating in the experiment. The students also had a choice to opt for an alternative project, in case they chose not to participate in the experiment and yet work for additional grade points. Those who participated in the experiment were asked to read and sign informed consent forms before the beginning of the experiment. The informed consent form is as shown in the appendix A. Since the research involved human subjects, appropriate approvals from the Institutional Review Board (IRB) were obtained through the University's Office of Research Compliance. After the completion of the experiment session, following the protocol, debriefing was conducted and the participants read and signed the debriefing forms. The debriefing form is as shown in the Appendix B.

4.3 Experimental Setting

Experiments were conducted in the behavioral laboratory of a university. The laboratory research facility has eight closed cubicles. There was a central observation area from where the participants can be observed through partially blinded one way mirrored glass panes. The cubicles were isolated and so designed that there was minimal interference due to outside activity or noise. The glass panes were one way

mirrored with blinds and so the participants could not look at any other computer screen except their assigned laptops. Up to two people could be comfortably accommodated in each cubicle with enough table space. They were provided with laptop computers, scratch pad and pencils to enable them to work on given programming tasks. The participants were provided with a brief one page review sheet for reference, scratch paper, and pencil, if needed. The review sheet is given in the appendix C. All the participants were read out general instructions. Before they went into their respective cubicles, they were asked to read and sign informed consent forms. The participants were assigned randomly to any of the experimental conditions. A deck of playing cards was used to make the randomizations possible. Once the condition was ascertained they were directed towards their respective cubicles. The participants were not allowed to take their books, notes, or cell phones into the cubicles.

The participants were then given instructions in their cubicles specific to their condition. They were provided with the warm up task. The warm up task was to be completed within thirty minutes. If the participants did not finish within thirty minutes, they were asked to stop working on the warm up task. The warm up task descriptions for all the three conditions are given in appendix D. The participants were asked to work on the main task after they stopped working on the warm up task. At the beginning of the main task, the students were told that they should try to achieve the best solution to the problem described in the main task statement. They were allowed

up to two hours to work on the main task. The main task descriptions for all the three conditions are given in appendix E.

For the dependent measures of learning and task description, experiment participants were asked to fill in their responses in a paper based questionnaire immediately after they had finished the main programming task. The paper based questionnaire consisted of four sections. Three sections were designed to capture learning, while the fourth section consisted of questions on the perceptual measure. The participants were asked to log out and switch off the computer before they could get the questionnaire. The problem statements, review sheet, and scratch paper were also collected before the participants could attempt the questionnaire. Those who were in the collaborating pair condition were asked to sit separately in different rooms so that they could not interact with each other while responding to the items in the questionnaire. Hence, when the subjects filled out the questionnaire, they did not have access to any resource that could provide them any reference. They had solely to rely on their memories. The questionnaires are as shown in appendix F.

After they had submitted the questionnaires, the participants were debriefed. They were given debriefing forms to read and sign. Finally, they were asked to collect their books, notes, cell phones, or other belongings before leaving. The participants were thanked and reminded about not to discuss about the experiments with any of their colleagues, until all the experimental sessions were completed.

4.4 Sample Size

There were three groups for comparison. Using Power Analysis, and for large population effect size and level of significance at 0.05, prior research suggests a group size of twenty one per condition (Cohen 1992). Since collaborating pairs were used in this study for one condition, a group size of twenty one pairs was used. For comparison between the collaborating pairs with the individual programmers, nominal pairs were created following the process of randomization. In each of the other two experimental conditions, forty two participants were used, in order to get twenty one nominal pairs in each condition. Assignment of an individual developer to a nominal pair was completely random and each individual programmer had an equal chance of being considered for any of the nominal pairs. Nominal pairs were not actual pairs, as in the case of collaborating pairs, but they were created only to facilitate comparisons between individual programmers and programmers working as collaborating pairs. The total sample size used for this study was 126. The experiments were conducted over three semesters, namely spring, summer, and fall of 2008.

4.5 Research Design

The research design showing the three experimental conditions is represented in Figure 4.1. Each participant was randomly assigned to one of the three experimental conditions. All the participants completed a warm up task before attempting the main programming task. One group of participants developed the code solution using the

traditional method of software development. Participants in the other two groups used TDD to complete their programming tasks.

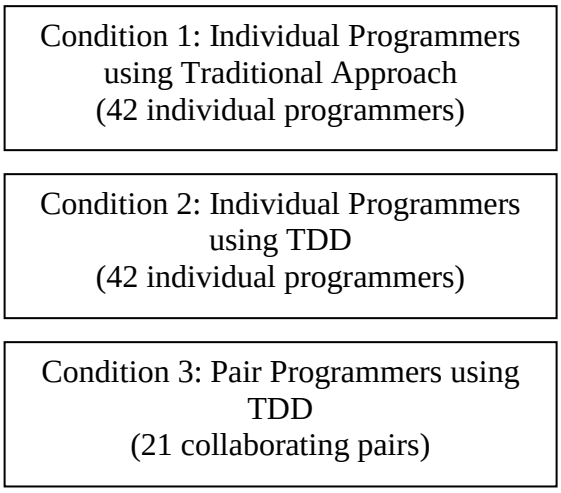


Figure 4.1 –Experimental Conditions

4.6 Measurement of Variables

4.6.1 Software Quality

Task performance of the programmers was measured by judging the quality of the code developed. A common rubric was used to reduce the amount of subjectivity in the assessment. Two types of errors have been identified in literature (Purao, 2003). Type I errors have been described as errors of omission, while Type II errors have been described as errors of commission. Complete solutions adequately providing required functionality would prevent against errors of omission and commission. The software quality assessment rubric is given in the appendix G. The programming task required programmers to use at least two classes. Each class was assessed based upon several steps. Points were awarded for creation of constructor, proper passing of input

variables, and correct initialization. The subjects had to write several methods for completing the code solution. Each method was checked for its syntax and functionality. The subjects lost points if their methods did not provide the required functionalities. The task required programmers to use some type of collection. The collection, namely Array or ArrayList, had to be implemented using composition, where objects of one class are stored in another class. This implementation also required creation of methods for populating the collection. The code solutions were also judged on the basis of syntax and functionality. The main method was also evaluated for proper execution. If the programmers provided more, and relevant, functionalities than asked, then they were awarded with bonus points. Again, specific grade points were assigned for different additional features like User Interface Design, additional methods etc. Software Quality was measured at the individual level for all participants who completed the programming task individually. It was measured at the group level for collaborating pairs.

4.6.2 Learning

Learning was measured through a questionnaire filled out after the completion of the main task. A more direct method of measuring learning could be to give the students another programming task after they had finished working on the main task. However, it could not be done because of feasibility considerations. The items of the questionnaires were developed based upon the prior literature (Mayer 1989, Gemino 1999). The questionnaires for all the experimental conditions are given in appendix F.

Three types of tests were used to measure learning – Cloze test, Comprehension test, and Problem Solving test.

The ability to recall verbatim was measured by using the Cloze test (Mayer, 1989). In a Cloze test, the subjects are provided with a statement from the problem description with several words missing. The subjects have to fill in the blanks based purely on memory. The Cloze test consisted of questions designed to assess verbatim recall of the description they read in the problem statement while attempting the main task. The Cloze test consisted of the same problem statement that the students had seen before while attempting the main programming task, with ten words removed. The words removed were mostly keywords that denoted important attributes of the objects when the software solution was developed.

The Comprehension test consisted of questions designed to test subject's understanding of the main programming task. Comprehension was measured by items in the questionnaire that required the experiment participants to identify objects, attributes of objects, and relationships between different objects based upon the programming task. Multiple questions were designed to assess understanding of the relationships between different objects that the subjects used to develop their code solutions. The questions were based exclusively on the scenario description given to the developers describing the main programming task.

After the section on Comprehension test, the questionnaire had a set of questions that went beyond the description provided to them in the main task. These types of questions have been described as problem solving questions (Mayer, 1989). The experiment participants were given a new task description. They were presented with a scenario that was different from the main programming task description. They were asked questions about how the classes and objects were similar to the classes and objects they had used in the main programming task. The subjects had to identify the class objects that would behave similarly to the class objects used in the main programming task. Specifically, the questions were designed to test whether the subjects were able to correctly apply the concepts they learned while solving the main programming task in a new context.

4.6.3 Overall Task Satisfaction

Overall Task Satisfaction was measured by adopting a pre-validated instrument (Balijepally 2006). A seven-point Likert Scale was used to develop items for measuring the level of overall task satisfaction. Participants were asked to report their overall experience based upon their working on the main programming task. On a scale of one to seven, their responses could range from very dissatisfied to very satisfied, very displeased to very pleased, very frustrated to very contented, and absolutely terrible to absolutely delighted.

4.7 Statistical Analysis

MANOVA and ANOVA were used for identifying important and significant differences between different groups. Two groups were considered at a time. First, the group that used the Traditional method of software development was compared with the group that used TDD. Second, the group that consisted of individual programmers was compared with the group that consisted of paired programmers, when both the groups used TDD. Nominal pairs were randomly created to facilitate the second comparison. The individual programmers that used TDD were randomly paired to create nominal pairs.

The first comparison between individual programmers using the Traditional method of software development and those using TDD was designed to reveal important and significant differences between the two methods of software development. The second comparison between individuals (as nominal pairs) and collaborating pairs both using TDD was designed to reveal important and significant differences between individual programmers versus pair programmers.

4.8 Pilot Testing

A pilot test was executed. Six students participated in the pilot test. Two programmers were asked to work as a collaborating pair and use TDD, two individual programmers were asked to use TDD, and two individual programmers worked on the problem using

the traditional method of software development. They were given clear instructions related to the task and the condition they were in. They were provided with laptops with Eclipse IDE to write a software solution to the programming task. Some minor changes were made in the script as a result of the pilot test.

CHAPTER 5

RESEARCH RESULTS

Results of Statistical analyses and hypotheses testing are given in this chapter.

5.1 Preliminary Analyses

5.1.1 Sample Characteristics

A total of 130 students participated in the experiment. Four students could not be included in the sample for various reasons. One student fell sick during the experiment and could not finish the main task. Three students were excluded because they did not respond completely to the questionnaire given out to them to measure dependent variables. Hence, responses from 126 students were used for data analysis. The mean age was found to be 26.64 years with 6.10 years of standard deviation. Of the 126 participants, ninety five (75.4%) were males and thirty one (24.6%) were females. In terms of education, seventy one (56.35%) students were undergraduate students while fifty five (43.65%) were graduate students. The experiments were conducted over three semesters. Forty students (31.75%) participated in the spring semester, Twenty seven students (21.43%) in the summer semester, and fifty nine students (46.82%) participated in the fall semester.

Two sets of analyses were done for this study. In first set of analysis one approach of software development (the traditional method) was compared with another approach of software Development (TDD). For the first set of analysis, the performance of the individual programmers using the traditional method of software development was compared with the performance of the individual programmers using TDD. Performance was measured through scores achieved by participants on three dependent measures - Solution Quality, Learning, and Overall Task Satisfaction. All the dependent measures were measured at the individual level.

The second set of analysis compared individual programmers with programmers working as collaborating pairs. Both the groups used TDD. The same performance measures, Solution Quality, Learning, and Overall Task Satisfaction were used for the second set of analysis. The subjects in pair condition worked collaboratively on the programming task and came up with one solution jointly. The software quality was measured at a group level in the case of pair condition. In a nominal pair, the best and the second best programmers were identified based upon their scores on the software quality measure. The other two measures of Learning and Overall Task Satisfaction were measured at individual level in both the conditions.

For comparison between the programmers using the traditional method of software development and those using TDD, MANOVA design was used for the analyses of dependent measures. The design is as shown in Figure 5.1.

Software Development approach		
	Traditional (1)	Condition (1) n = 42 individual programmers
	TDD (2)	Condition (2) n = 42 individual programmers

Figure 5.1 – MANOVA Design for comparison between Traditional and TDD

For comparison between the nominal pairs and the collaborating pairs, when both the groups used TDD, ANOVA and MANOVA designs were used. For ANOVA design three levels were used for the dependent measure of Software Quality as shown in Figure 5.2.

Condition	Solution Code developed using TDD
Best Programmer in Nominal Pair (1)	Condition (1) n = 21
Second BestProgrammer in Nominal Pair (2)	Condition (2) n = 21
Collaborating Pair (3)	Condition (3) n = 21 pairs

Figure 5.2 – ANOVA Design (I-P) (both using TDD)

For MANOVA design, four levels were used for the comparisons based on the dependent measures of Learning and Overall Task Satisfaction. The design is as shown in Figure 5.3.

Condition	Solution Code developed using TDD
Best Programmer in Nominal Pair (1)	Condition (1) n = 21
Second Best Programmer in Nominal Pair (2)	Condition (2) n = 21
Best Programmer in Collaborating Pair (3)	Condition (3) n = 21
Second Best Programmer in Collaborating Pair (4)	Condition (4) n = 21

Figure 5.3 – MANOVA Design (I-P) (both using TDD)

5.1.2 Factor Analysis

For this study, one perceptual measure was used. Overall Task Satisfaction was measured using a seven point Likert scale. Overall Task Satisfaction was measured using four items in the questionnaire as shown in Appendix F. Table 5.1 shows the correlation matrix for the items used for the perceptual measure. The mean values and standard deviations (in parentheses) of the scores on items are also displayed.

Table 5.1 – Correlation Matrix for the Perceptual Measure

	Item 1	Item 2	Item 3	Item 4
Item 1	5.50 (1.506)			
Item 2	0.885	5.48 (1.558)		
Item 3	0.873	0.779	5.36 (1.695)	
Item 4	0.853	0.833	0.889	5.29 (1.492)

Exploratory Factor Analysis was performed using Principal Component Analysis. All the four items were found to load into one factor. Table 5.2 shows the factor loadings found as a result of using Principal Component Analysis, along with Eigen Value and Variance Explained.

Table 5.2 – Factor Loadings for the Perceptual Measure

Questionnaire Item	Factor Overall Task Satisfaction	Communality Estimate
Item 1	0.958	0.917
Item 2	0.927	0.859
Item 3	0.939	0.882
Item 4	0.948	0.899
Eigen Value	3.557	
Variance Explained	88.91%	

Since high factor loadings were found, a composite score for Overall Task Satisfaction was used. The item scores were summated and then averaged to compute the composite score which was used in subsequent analysis.

5.1.3 Reliability of Dependent Measures

Dependent measures used for this study included Software Quality, Learning, and Overall Task Satisfaction. Software Quality was measured at two levels. For the participants using the traditional software development approach, Software Quality was measured at individual level. For those using TDD, software quality was measured at individual level in one case. And, in the other case, it was measured at the group level, where participants worked as collaborating pairs. The scores assigned by the two graders (doctoral GTA's) were then checked for internal consistency using Pearson correlation, value obtained was 0.791. In equivalent Cronbach's alpha terms, it would correspond to 0.889 (Carmines & Zeller 1979). Since all the questions for measuring learning were multiple choice objective types, the assessment was fairly straight forward. Two doctoral GTA's graded the responses on Learning and their reported scores matched one hundred percent with each other. The items on Overall Task Satisfaction were checked for internal consistency by Cronbach's alpha, which was found to be 0.956. Cronbach's alpha can be considered an indicator of internal consistency and homogeneity of the measured variable (Kerlinger, 1986). Generally, values over 0.7 are considered adequate for assuming reliability (Nunnally, 1978).

5.1.4 Assumptions check

Before proceeding with the statistical analysis, checks for assumptions violations were performed. In ANOVA, three assumptions are to be met to sustain statistical significance in substantiating claims hypothesized. The three assumptions are constancy of error variance, independence of error terms, and normality of the error terms (Kutner et al, 2004). The F test is considered fairly robust against violations of equal error variances in fixed ANOVA model if the factor level sample sizes are approximately equal or do not differ greatly (Kutner et al, 2004). In this study, samples sizes across the comparisons were equal; hence departures from equal variances cannot be considered any serious threat to generalizations. For checking violations of normality, minor violations were found in some of the cases. Transformations were applied as a remedy. Exponential transformation, with exponent value of 2.5 alleviated problem of normality violations. Upon examining the residual plots, no violations for independence of error terms were found. Power Analysis was also done as shown in Table 5.3.

Table 5.3 – Power Analysis

Comparison	Dependent Variable	Parameter	Partial Eta Squared	Power(at p = 0.05)
Traditional versus TDD	Software Quality	I - I	0.412	0.953
	Verbatim Recall	I - I	0.014	0.186
	Comprehension	I - I	0.053	0.565
	Problem Solving	I - I	0.003	0.077
	Overall Task Satisfaction	I - I	0.087	0.791
Individual versus Pairs	Software Quality	I - P	0.329	0.999
	Verbatim Recall	I - P	0.037	0.242
	Comprehension	I - P	0.004	0.066
	Problem Solving	I - P	0.038	0.249
	Overall Task Satisfaction	I - P	0.100	0.611
Legend: I – Individual Programmer, P – Collaborating Pair Programmer FB – First Best in Nominal Pair, SB – Second Best in the Nominal Pair				

5.1.5 Test of Significance – MANOVA and ANOVA

Two separate analyses were done for this study. In the first analysis, the participants using TDD were compared with those who used the traditional method of software development. All the dependent measures were used and MANOVA results of this analysis are shown in Table 5.4.

Table 5.4 – MANOVA results for comparison of Traditional with TDD

Statistical Test	Value	F value	Degrees of Freedom		Sig. p-Value
			Between Group	Within Group	
Pillai's Trace	0.197	3.829	5	78	0.004*
Wilk's Lambda	0.803	3.829	5	78	0.004*
Hotelling-Lawley Trace	0.245	3.829	5	78	0.004*
Roy's Largest Root	0.245	3.829	5	78	0.004*

**significant at p = 0.05*

With the MANOVA model significant, One-Way ANOVA was performed and the results are tabulated in Table 5.5.

Table 5.5 – One-Way ANOVA results for comparison of Traditional with TDD

Dependent Measure	Test Driven Development		Traditional Method		F Value	Sig. p-Value
	Mean	SD	Mean	SD		
Software Quality	93.73	17.40	76.06	25.78	13.55	0.000*
Verbatim Recall	7.71	1.70	7.26	1.87	1.342	0.125
Comprehension	6.60	1.49	6.02	1.44	3.175	0.038*
Problem Solving	8.33	2.02	8.52	1.53	0.237	0.314
Overall Task Satisfaction	5.64	1.31	4.72	1.68	7.850	0.003*

**significant at p = 0.05*

The second comparison was between the individual programmers and paired programmers. However, all the dependent measures could not be considered simultaneously since they were measured at different levels. The pair came up with a

joint code solution. Software Quality was measured at the group level for the pair condition. The score received by the collaborating pair on software quality was compared with the first and second best individual programmer in the nominal pair. A separate ANOVA was conducted; the details are as shown in Table 5.6.

Table 5.6 – One-Way ANOVA results on Software Quality (I-P)

Dependent Measure	Best Individual in Nominal Pair		Second Best Individual in Nominal Pair		Collaborating Pair		F Value	Sig. p-Value
	Mean	SD	Mean	SD	Mean	SD		
Software Quality	103.45	3.99	82.47	19.73	97.19	14.53	11.84	0.000*

**significant at p = 0.05*

The dependent measures for Learning and Overall Task Satisfaction were measured at the individual level for comparing Individual Programmers with the Programmers who worked as collaborating pairs. MANOVA was performed; results are shown in Table 5.7.

Table 5.7 – MANOVA results on Learning and Task Satisfaction (I-P)

Statistical Test	Value	F value	Degrees of Freedom		Sig. p-Value
			Between Group	Within Group	
Pillai's Trace	0.294	2.119	12	237	0.017*
Wilk's Lambda	0.734	2.107	12	204	0.018*
Hotelling-Lawley Trace	0.329	2.107	12	227	0.019*
Roy's Largest Root	0.181	3.569	4	79	0.010*

**significant at p = 0.05*

The MANOVA model was found to be significant, One-Way ANOVA's were performed, and the results are tabulated in Table 5.8.

Table 5.8 – One Way ANOVA results on Learning and Task Satisfaction (I-P)

F Value (pValue)	Best Individual in Nominal Pair		Second Best Individual in Nominal Pair		Best in Collaborating Pair		Second Best in Collaborating Pair	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
Verbatim Recall Score								
1.979 (0.062)	7.86	1.74	7.57	1.69	7.57	1.46	6.57	2.33
Comprehension Score								
2.468 (0.025*)	6.81	1.63	6.38	1.35	6.67	1.01	5.76	1.13
Problem Solving Score								
1.974 (0.062)	8.10	2.14	8.57	1.91	9.24	1.48	8.00	1.78
Overall Task Satisfaction Score								
2.489 (0.004*)	5.87	1.45	4.81	1.53	5.90	1.36	5.51	1.55

**significant at p = 0.05*

5.2 Hypotheses Testing

Hypotheses testing regarding comparison of the Traditional method of software development with TDD are discussed in this section.

5.2.1 Comparison of Traditional versus TDD

Hypothesis H1

H1: While working on a programming task, individual programmers using TDD perform better in terms of Software Quality as compared to the individual programmers using the traditional method of software development.

Table 5.9 – Mean Software Quality Scores for Traditional and TDD

Software Quality	Mean Scores
Test Driven Method of Software Development	93.73*
Traditional Method of Software Development	76.06

**significant at p = 0.05*

The performance of the programmers using TDD was found to be significantly higher than the performance of the programmers using the Traditional method of software development. On average, participants using TDD scored 93.73 while those using the traditional method of software development scored 76.06 on software quality measure

as shown in Table 5.9. The ANOVA test gave F value of 13.55 with p-Value 0.000 (significant at 0.05).

The following three hypotheses, H2a, H2b, and H2c, are for the dependent measure of Learning.

Hypothesis H2a

H2a: While working on a programming task, individual programmers using TDD perform better in terms of verbatim recall as compared to the individual programmers using the traditional method of software development.

Table 5.10 – Mean Verbatim Scores for Traditional and TDD

Verbatim Recall	Mean Scores
Test Driven Method of Software Development	7.71
Traditional Method of Software Development	7.26

**significant at p = 0.05*

In terms of verbatim recall, the performance of the subjects using TDD was found to be higher than the subjects using the traditional method. Participants using TDD scored 7.71 while those using the traditional method of software development scored 7.26, as shown in Table 5.10. However, this difference was not found to be statistically

significant. ANOVA test gave F value of 1.342 with p-Value 0.125 (not significant at 0.05).

Hypothesis H2b

H2b: While working on a programming task, individual programmers using TDD perform better in terms of comprehension as compared to the individual programmers using the traditional method of software development.

Table 5.11 – Mean Comprehension Scores for Traditional and TDD

Comprehension	Mean Scores
Test Driven Development	6.60*
Traditional Method	6.02

**significant at p = 0.05*

In terms of Comprehension, the performance of the subjects using the TDD approach was found to be higher than the subjects using the Traditional method of software development. On average, Participants using TDD scored 6.60, while those using the traditional method of software development scored 6.02, as shown in Figure 5.11. It was statistically significant. ANOVA test gave F value of 3.175 with p-Value 0.038 (significant at 0.05).

Hypothesis H2c

H2c: While working on a programming task, individual programmers using TDD perform better in terms of problem solving as compared to the individual programmers using the traditional method of software development.

Table 5.12 – Mean Problem Solving Scores for Traditional and TDD

Problem Solving	Mean Scores
Test Driven Development	8.33
Traditional Method	8.52

**significant at p = 0.05*

In the case of performance in terms of Problem Solving, contrary to the argument made in this study, it was found that the experiment subjects using the traditional method of software development scored higher than the subjects using TDD, as shown in Table 5.12. On average, programmers using the traditional method of software development scored 8.52 while those using TDD scored 8.33. The difference, however, was not statistically significant with F value 0.237 and p-Value 0.314.

Hypothesis H3

H3: While working on a programming task, overall task satisfaction of the individual programmers using TDD is higher than the overall task satisfaction of the individual programmers using the traditional method of software development.

Table 5.13 – Mean Overall Task Satisfaction Scores for Traditional and TDD

Overall Task Satisfaction	Mean Scores
Test Driven Development	5.64*
The Traditional Method	4.72

**significant at p = 0.05*

The experiment participants using TDD were found to score higher on Overall Task Satisfaction as compared to the programmer who used the Traditional method of software development. The mean scores achieved by the participants using the Traditional method of software development and TDD are given in Table 5.13. On average, the software developers using TDD scored 5.64 while those using the traditional method of software development scored 4.72. The difference between the two scores was found to be statistically significant. ANOVA test showed F value 7.85 with p-Value 0.003 (significant at 0.05).

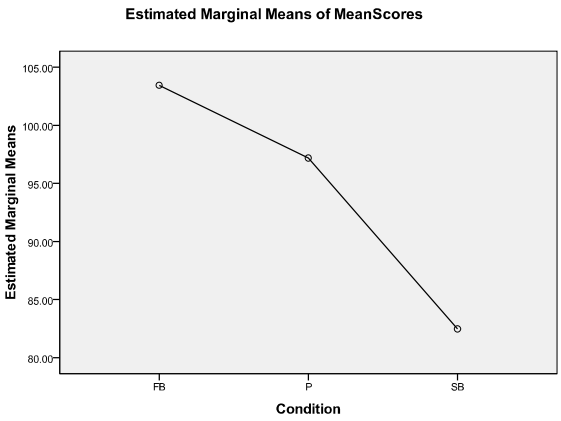
5.2.2 Comparison of Individual versus Pairs using TDD

The hypotheses regarding comparison of the collaborating pairs with the nominal pairs are discussed in this section. The two groups are compared on the basis of their performance on Software Quality, Learning, and Overall Task Satisfaction measures.

For comparison in terms of Software Quality, the ANOVA model was significant with F- Value 11.84 and p-Value 0.000 (significant at .05). Figure 5.4 shows the marginal means for scores on software quality for the three conditions.

Hypothesis H4

H4: While working on a programming task using TDD, performance of a collaborating pair in terms of software quality is higher than the performance of the second-best programmer in a nominal pair.



FB: First Best in Nominal Pair
SB: Second Best in Nominal Pair
P: Collaborating Pair

Figure 5.4 – Marginal Means of Software Quality (I-P)

Table 5.14 –Software Quality Scores (I-P)

Software Quality	Mean Scores
The best Programmer in Nominal Pair	103.45
Second best programmer in the nominal pair	84.47
Collaborating pair	97.19*

**significant at p = 0.05*

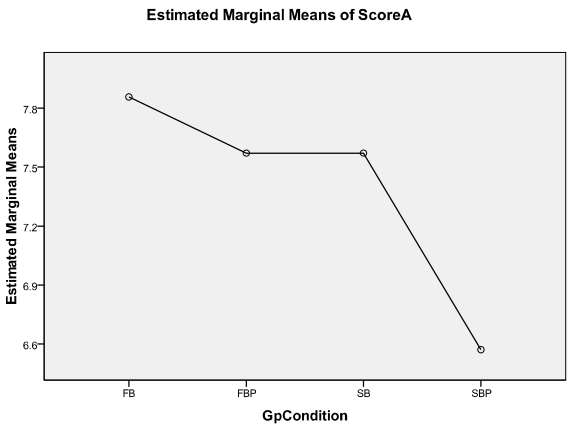
When the score on software quality of the collaborating pair was compared with that of the second best individual programmer in the nominal pair, the collaborating pair

scored higher. Table 5.14 shows the mean score attained by the collaborating pair, the best and the second best individual programmer. The best programmer in the nominal pair scored 103.45 points. The collaborating pair scored 97.19 while the second best programmer scored 84.47. Collaborating pair scored significantly higher score than the second best programmer in the nominal pair. To check statistical significance of this finding further, Bonferroni pairwise comparisons were examined. The Bonferroni confidence interval for the difference of 14.71 points between the collaborating pair and the second best programmer in the nominal pair was found to be between 3.82 and 25.61. This confidence interval did not contain zero, hence the difference was inferred as statistically significant. No significant difference was found between the best programmer in the nominal pair and the collaborating pair.

The hypotheses H5a, H5b, and H5c, are for the dependent measure of Learning.

Hypothesis H5a

H5a: While working on a programming task using TDD, performance of the best programmer in the collaborating pair in terms of verbatim recall is higher than the performance of the second best programmer in a nominal pair.



FB: First Best in Nominal Pair; SB: Second Best in Nominal Pair
FBP: First Best in Collaborating Pair; SBP: First Best in Collaborating Pair

Figure 5.5 – Marginal Means of Verbatim Recall (I-P)

Table 5.15 –Verbatim Recall Scores for Individuals and Pairs

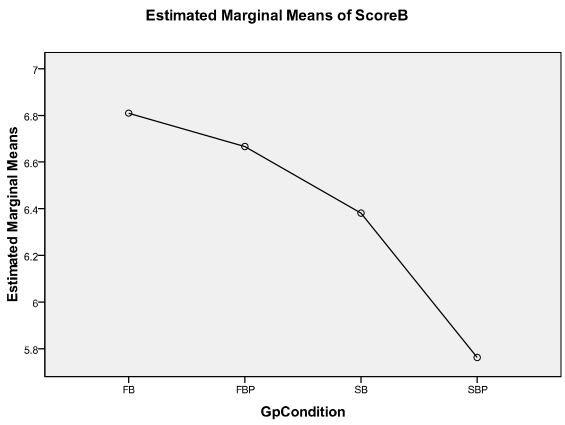
Verbatim Recall	Mean Scores
The best Programmer in Collaborating Pair	7.57
Second Best Programmer in Collaborating Pair	6.57
The best in programmer in the nominal pair	7.86
Second best programmer in the nominal pair	7.57

**significant at p = 0.05*

The second best programmer in the nominal pair scored same as the best programmer in the collaborating pair. Table 5.15 shows the mean scores on the Cloze test attained by the programmers. The means are plotted as in Figure 5.5. However, no significant differences were found between any of the four scores.

Hypothesis H5b

H5b: While working on a programming task using TDD, performance of the best programmer in the collaborating pair in terms of comprehension is higher than the performance of the second best programmer in a nominal pair.



FB: First Best in Nominal Pair; SB: Second Best in Nominal Pair
FBP: First Best in Collaborating Pair; SBP: First Best in Collaborating Pair

Figure 5.6 – Marginal Means of Comprehension scores (I-P)

Table 5.16 –Comprehension Scores (I-P)

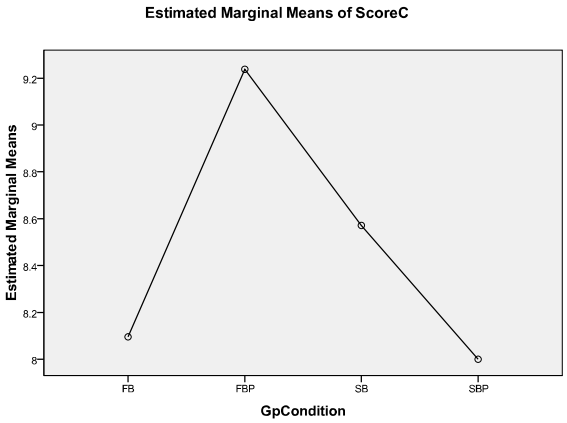
Comprehension	Mean Scores
The best Programmer in Collaborating Pair	6.67
Second Best Programmer in Collaborating Pair	5.76
The best in programmer in the nominal pair	6.81
Second best programmer in the nominal pair	6.38

**significant at p = 0.05*

The best programmer in the collaborating pair scored 6.67, which was higher than the second best programmer in the nominal pair whose score was 6.38. Table 5.16 shows the mean scores on Comprehension test attained by collaborating pair and the nominal pair. The means are plotted as in Figure 5.6. The difference between any two scores was not found to be statistically significant.

Hypothesis H5c

H5c: While working on a programming task using TDD, performance of the best programmer in the collaborating pair in terms of problem solving is higher than the performance of the second best programmer in a nominal pair.



FB: First Best in Nominal Pair; SB: Second Best in Nominal Pair
FBP: First Best in Collaborating Pair; SBP: First Best in Collaborating Pair

Figure 5.7 – Marginal Means of problem solving scores (I-P)

Table 5.17 – Problem solving scores (I-P)

Problem Solving	Mean Scores
The best Programmer in Collaborating Pair	9.24
Second Best Programmer in Collaborating Pair	8.00
The best in programmer in the nominal pair	8.10
Second best programmer in the nominal pair	8.57

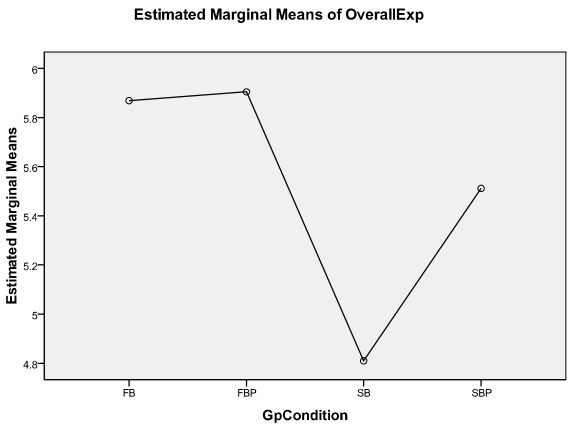
**significant at p = 0.05*

Table 5.17 shows the mean scores on Problem Solving test for the collaborating pairs and the nominal pairs. The difference was not found to be statistically significant at level 0.05. The means are plotted as shown in Figure 5.7. Thus, the collaborating pair scored higher than the second best in terms of problem solving, but the difference was not found to be statistically significant.

For the Task Satisfaction measure, ANOVA model was significant (F value 2.48, p value 0.004). The mean scores are shown in Figure 5.8.

Hypothesis H6

H6: While working on a programming task using TDD, overall task satisfaction of the best programmer in a collaborating pair is higher than overall satisfaction of the second best programmer in a nominal pair.



FB: First Best in Nominal Pair; SB: Second Best in Nominal Pair
FBP: First Best in Collaborating Pair; SBP: First Best in Collaborating Pair

Figure 5.8 – Marginal Means of overall task satisfaction (I-P)

Table 5.18 – Overall Task Satisfaction (I-P)

Problem Solving	Mean Scores
The best Programmer in Collaborating Pair	5.90
Second Best Programmer in Collaborating Pair	5.51
The best in programmer in the nominal pair	5.87
Second best programmer in the nominal pair	4.81

**significant at p = 0.05*

Mean scores of overall task satisfaction of the collaborating pairs and the individual programmers are as shown in Table 5.18. The second best programmer in the nominal pair scored 4.81 points while the best programmer in the collaborating pair scored 5.90 points. The difference however was not found to be statistically significant, when checked using Bonferroni multiple comparisons. The Bonferroni confidence interval include zero between the values -0.137 and 2.33.

The findings of this study are summarized in Table 5.19.

Table 5.19 – Summary of findings

Hypothesis	*Finding*
"While working on a programming task, individual programmers using TDD perform better in terms of……."	
H1: Software Quality as compared to the individual programmers using the traditional method of software development.	*Supported*
H2a: verbatim recall as compared to the individual programmers using the traditional method of software development.	*Not Supported*
H2b: comprehension as compared to the individual programmers using the traditional method of software development.	*Supported*
H2c: problem solving as compared to the individual using the traditional method of software development.	*Not supported*
H3: overall task satisfaction of the individual programmers using the traditional method of software development.	*Supported*
"While working on a programming task using TDD, performance of a collaborating pair in terms of……."	
H4: software quality is better than the performance of the second-best programmer in a nominal pair.	*Supported*
"While working on a programming task using TDD, performance of the best programmer in a collaborating pair in terms of……."	
H5a: verbatim recall is better than the performance of the second best programmer in a nominal pair.	*Not supported*
H5b: comprehension is better than the performance of the second best programmer in a nominal pair.	*Not supported*
H5c: problem solving is better than the performance of the second best programmer in a nominal pair.	*Not Supported*
H6: overall task satisfaction of the collaborating pair is better than overall satisfaction of the second best programmer in a nominal pair.	*Not supported*

CHAPTER 6

DISCUSSIONS AND CONCLUSIONS

This chapter includes discussions, implications of the findings for practitioners and academicians, and the limitations.

6.1 Discussion

Findings of the experiment are discussed in this section. Comparison of the traditional versus the TDD is followed by comparison of individual versus pair programmers.

6.1.1 Comparison of Traditional with TDD

Based upon the quality of software solutions produced, it was found in this study that on average, participants using TDD performed better than those using the Traditional method of software development. The group using TDD on average scored 93.73 points while the group using the Traditional method of software development scored 76.06 points. The difference between the two scores was found to be statistically significant. This finding is important since it helps resolve the debate on the effectiveness of TDD. Several studies were conducted to investigate the effectiveness of TDD using different methodologies across multiple settings. The mixed findings of those studies failed to

converge to provide guidance to academics and practitioners. This study followed a rigorous methodology and the findings clearly demonstrate the effectiveness of the TDD as software developers using TDD produced higher quality software solutions as compared to those using the Traditional method of software development.

This study also provides evidence that the use of TDD by developers leads to a better task comprehension of the task. The group using TDD scored 6.60 points while the group using the Traditional method of software development scored 6.02. The difference between the two scores was found to be statistically significant. It was argued in this study that TDD leads to better comprehension as compared to the traditional method of software development. The argument was based on the fact that TDD requires developers to build testcases before they write code. Doing so would require them to think ahead and more importantly look at the scenarios from multiple perspectives. The findings provide strength to the argument that TDD leads to better comprehension of the task among developers.

The process of using TDD requires developers to follow the cycle of red, green, and refactor. Testcases fail when they are run the first time. Developers then write just enough code to prevent testcases from failing. This cycle of testcases failing (red) and writing some code to prevent failure of testcases (green) is followed iteratively. After a few iterations, developers look at the written code for redundancy, and refactor the code

to enhance quality of code. The process of implementing TDD provides ample feedback to the developers at a rapid rate. Since the developers have higher confidence in the solution because of testing process, it leads to higher overall task satisfaction among them. The group using TDD on average scored 5.64 points while the group using the traditional method of software development scored 4.72. The difference was found to be statistically significant. Higher overall task satisfaction also provides further evidence of the effectiveness of TDD.

There were some hypotheses that were not found to be statistically significant. Verbatim recall, measure of first level of learning, was not found to be significantly different when developers used TDD in place of the Traditional method of software development. Programmers using TDD scored 7.71 which was a little higher than corresponding average of 7.26 for programmers that used the Traditional method of software development. Lack of significance can also be interpreted as a sign of the effectiveness of the tool used to implement TDD (i.e. Eclipse IDE with JUnits). One stream of researchers believes that if the tool is effective, it might reduce verbatim recall but increase comprehension (Gemino 1999, Mayer 1989). This line of thought finds further support because significantly higher score on comprehension measure were actually found in the study.

Both groups also did not exhibit any difference on the measure of problem solving, which was the third level of learning used in this study. Problem solving measure is more complex than other measures of learning. Other factors, such as stress, fatigue, motivation level, etc. that were not considered in the study might have influenced the measure of problem solving.

Summarizing, it was found in this study that as compared to the Traditional method of software development, TDD yields better quality software, leads to better comprehension, and helps developers achieve higher overall task satisfaction. However, no statistical evidence was found that would suggest that TDD promotes better verbatim recall or enhances problem solving ability as against the Traditional method of software development.

6.1.2 Comparison of Individuals with pairs using TDD

In case of the dependent measure of software quality, it was found that collaborative pairs on average performed better than the second best individual programmer in the nominal pair. For measures of learning and task satisfaction, no significant differences were found between nominal and collaborative pairs. Thus, this study found no evidence to suggest that a collaborating pair outperform the second best programmer in a nominal pair in terms of measures of learning or task satisfaction. There could be

several reasons for not finding significance in case of comparison of individuals with pairs. Some of those reasons are listed in the limitations section. No prior experience of collaborative work was required. Motivation of the students might have been an issue, as sometimes students do not wish to put in their best in an assignment that is not a direct part of their course grade. Sometimes, students are highly stressed out because of course load or any other personal issue. Having said that, randomization should have taken care for any such extraneous influences. However, the consideration of more variables, either as covariates or blocking variables, could have influenced the results as mentioned in the later section on future research directions.

6.2 Significance of the findings to practitioners

This study makes significant contributions to the practice of software development. First it demonstrates that TDD enables developers to produce code of a higher quality. As organizations strive to constantly add new features and functionalities to the existing IT systems, maintenance and updating becomes a major task. Developers are required to modify some components of the existing code to add new features and functionalities. Mostly a single change in code affects many different areas of the program because of coupling. Identifying areas affected by one change itself becomes a time consuming and cumbersome task. Higher quality would manifest in lack of errors, lack of coupling, and lack of redundancies. Therefore, code developed using TDD

would be easier to maintain and update. Second, TDD promotes developers to have better comprehension of the task. It is important for the organizations to use methodologies that help in better comprehension among employees. Learning is an important asset as software development task is a knowledge intensive process. Higher comprehension is also likely to build better relationships with client and enhance the quality of support in later years. Third, use of the TDD approach helps developers achieve higher overall task satisfaction. Higher task satisfaction is likely to enhance employee retention and improve productivity. Finally, this study provides evidence that when developers work together as collaborative pairs; they perform better than the second best in the nominal pair.

6.3 Significance of the findings to academicians

The findings of this study are significant for the researchers interested in the area of software development. The effectiveness of the TDD technique has been an issue of interest for the software development researchers. The findings of this study provide support to the claim that the TDD technique helps software developers write code of a higher quality. The research model used for this study was derived from theory. Particularly, the use of the Flow theory to explain the process of application of the TDD technique might be of interest to the software development researchers. The findings of this study are also significant for all those who are interested in the small group research. Researchers have compared individuals with groups, and have claimed that

groups perform better than individuals (Laughlin et. al 2003). However, the findings of this study provide further evidence to the claim that groups are rarely better than the best individual. In this study, it was hypothesized that, as compared to the traditional method of software development, the TDD technique would help software developers learn more about the task. Learning is a significant area for IS researchers since it has not been much investigated in the field of software development. The findings of this study on comprehension are significant for the academicians since they contribute to the literature on learning in the field of software development. Finally, the findings of this study are significant for the software development researchers interested in the topic of task satisfaction when developers use the TDD technique.

6.4 Limitations

Software developers working in the industry would have been ideal subjects for this study. Students were used as proxies for software developers. This is a limitation of this study. The use of the university students as subjects might raise issues of generalizability. It might be argued that students can not be considered as a good choice since they do not possess the experience or expertise to match with software developers who have been working in the industry. However, there are some factors that might actually be considered in favor of student subjects. Software professionals who have been working using one type of methodology might develop habitual routines specific

to that methodology and might try to build code from that mindset in the case they are randomly assigned to work following a different methodology. It might lead to bias in data collected. Hence, those professionals, who have been working using only the Traditional method of software development, might not be suitable to be considered as subjects for the TDD condition and vice versa. It is difficult to identify such professionals in the industry who can be considered equally competent in both the traditional and the TDD technique. Also, lack of experience of programming might actually be beneficial since the subjects are likely to be more amenable to learning a different method of coding quickly. Indirect measurement of learning is another limitation of this study. Mental exhaustion and fatigue accumulated while working on the main task imposed feasibility constraints on giving the participants another programming task. Due to feasibility constraints, reliability tests such as Kuder Richardson method could not be applied to the measures of learning. This was a limitation of the study. Another limitation of the study was that the collaborating pairs might not have got enough time to jell with each other. Pairs were put together in the experimental condition without any prior experience of working together as a pair. Although they were asked to work on the warm up task and that might have mitigated this issue to some extent. Pair programmers after collaborating on several projects can be expected to become more effective at software development since over a period of time there is likely to be a better jelling among the team members.

6.5 Future directions

There are several directions that the future studies can take. This study used participants from a university student population. In future, this study can be replicated using practitioners, the software professionals working in the industry; it might provide more generalizable results. Future studies could also consider the task complexity level. The complexity level of the programming tasks can be varied and that can throw more light on the effectiveness of the TDD technique. Learning was one of the measures used in this study. Future studies could measure the learning in a different way. An interesting, and perhaps more direct, way of measuring learning can be to ask the participants to work on a programming task, after they have finished working on the main task.

There can be several other measures that can be used in future studies. Several more process variables, like self efficacy, motivation, engagement, and task mental model, can be studied to reveal more insights into the effectiveness of TDD. For instance, investigation of the task mental models might reveal insights into the change in the thought processes of the developers when they use TDD. Motivation can be used as another interesting measure to investigate the phenomenon of TDD. Similarly, self efficacy and engagement can also be included in future studies on TDD.

APPENDIX A

INFORMED CONSENT

INFORMED CONSENT
Software Design Experiment

In this study you will be asked to work on a software design task using object oriented programming approach. You may be working individually or with another partner. You will work initially on a warm up task before proceeding to work on the main task. The total duration of the experimental session will be approximately three hours. Problem statements will be provided to you with a any additional materials required for solving the problems. You will also be asked to complete a questionnaire about your understanding of the task and also about your reactions to working on the task.

Since you may be working with other people you may experience some emotional discomfort, similar to what you could experience in the workplace when working on tasks of this nature. Those discomforts may include fatigue, boredom or frustration when you work with other people to solve problems.

The major benefit that you will get in participating in this study is to get an understanding of modern software development practices that may be of great value. You will be debriefed immediately after the experiment. The benefits to the investigator are increased understanding of pair working on software tasks and the usefulness of Test Driven Development (TDD) technique in software development.

Your participation in this experiment is voluntary and you can withdraw your informed consent at any time, if you find any procedures objectionable. Records of your participation and any data collected will be held in strict confidence.

This research study has been reviewed and approved by the University of Texas at Arlington Institutional Review Board. In the event you are injured in the course of this study, you may be covered under optional medical insurance that you carry. UTA does not offer any other compensation for injury.

This research is under the supervision of Dr. Radha Mahapatra. Dr. Mahapatra's office is room 521 of the Business Building and his phone number is (817) 272-3590. Please feel free to contact him if you have questions. If you have any questions about your rights as a subject or about a research related injury, you may contact the office of Research Compliance at 817-272-3723.

I had a chance to ask all questions regarding this study. I hereby consent to participate in the experiment and understand the above procedures.

Signature: _____

Print Name: _____

Date: _____

APPENDIX B

DEBRIEFING

DEBRIEFING

In this study we were interested in examining the effectiveness of working individually versus working in pairs. We are also interested in studying the effect of Test Driven Development (TDD) on the effectiveness of pair versus individuals in software development. Pairs are increasingly used in software development projects for various tasks and they are claimed to increase quality and productivity. Similarly, TDD is claimed to provide better software quality through more rigorous testing. Our specific interest in this study is to see whether there were differences in the quality of solution, when pairs and individuals employ the TDD technique. We are also interested in finding the effect of the process of code development on learning of developers. We are also interested in the task satisfaction achieved when individuals or pairs develop software using TDD.

In studying this question we randomly assigned members to individual or pair condition, employing the traditional versus TDD methodology of software development. Quality of your design, learning and task satisfaction will be used to judge the effect of various factors on effectiveness of pair / individuals in software development employing the traditional / TDD approach.

We will be conducting this study with students of the programming courses at UTA. It is vitally important to us and the success of this experiment that you keep the information that you have learned here in confidence. Please do not tell anyone about this experiment. We are confident that we can trust you. Thank you.

This research is under the supervision of Dr Radha Mahapatra and you can contact him at his office which is in Room 521 of the Business Building and his phone number is (817) 272-3590. If you are interested in knowing the results of the experiment, please feel free to contact him after five weeks.

Please sign below to indicate that you understand this debriefing and that you promise to keep what you have learned in confidence. Again, thank you very much for your participation.

Signature _____ Print Name_____

Date _____

APPENDIX C

REVIEW SHEET FOR PARTICIPANTS

Concepts Review

Composition: When one class has objects from another class, it is called as composition.

Collections: When you need to store variables / objects, you use collections. Arrays, Vectors, ArrayLists are all examples of collection. In Arrays, the size needs to be declared before you can add values, so they have the limitation of having a fixed (static) size. ArrayLists can be dynamically created when the objects are added.

ArrayLists: What is an ArrayList?
Java ArrayList is a resizable array which implements List interface. ArrayList provides all operation defined by List interface. Internally ArrayList uses array to store elements. ArrayList provides additional methods to manipulate the array that actually stores the elements. ArrayList is equivalent to Vector, but ArrayList is not synchronized. You don't have to declare any fixed size. It dynamically creates a collection of objects.

How to declare an ArrayList?
ArrayList < Object type > arrayList = **new** ArrayList < Object type > ( );

How to add to an ArrayList?

You can pass the Object type as parameter and call add method. Such as:

```
public void enroll (Student s){
        mList.add(s);      //mList is an ArrayList
}
```

How to read from an ArrayList?
Use a loop and iterate through the list to read the values one by one. For example, in the following code, c is the object of Course class and getAllStudent() is its method that returns Objects of Student type as an ArrayList. If we have to iterate thorough the list of Student objects, we will read from the ArrayList by using the following syntax:

```
ArrayList<Student> aList = c.getAllStudents(); //c is Course object
                                    // getAllStudents() returns ArrayList of Students
            for (int i = 0; I < aList.size(); i++){
                        Student s;
                        s = aList.get(i);
                        System.out.println(s.getName());}
```

APPENDIX D

WARM UP TASK FOR PARTICIPANTS

For individual participants using Traditional method of software development:

A movie rental business owner has hired you as a software consultant and wants you to develop an application for him. The application should allow a way to create a list of movies. It should also allow addition of movies to the list. The order of the movie list is not important. The application should display the total number of movies listed at a time. You should get the output displayed on the monitor (command prompt).

A sample output could be:

No. of Movies currently available: 5

\

For individual participants using TDD:

You have to use TDD and write relevant TestCase in developing the following application.

A movie rental business owner has hired you as a software consultant and wants you to develop an application for him. The application should allow a way to create a list of movies. It should also allow addition of movies to the list. The order of the movie list is not important. The application should display the total number of movies listed at a time. You should get the output displayed on the monitor (command prompt).

A sample output could be:

No. of Movies currently available: 5

For participants using TDD and working as collaborating pair:

You have to collaborate and work collectively as a pair on this task. You have to use TDD and write appropriate TestCase in developing the following application.

A movie rental business owner has hired you as a software consultant and wants you to develop an application for him. The application should allow a way to create a list of movies. It should also allow addition of movies to the list. The order of the movie list is not important. The application should display the total number of movies listed at a time. You should get the output displayed on the monitor (command prompt).

A sample output could be:

No. of Movies currently available: 5

APPENDIX E

MAIN TASK FOR PARTICIPANTS

For individual participants using Traditional method of software development:

The owner of a book store wants to keep records of the books in stock on the computer. The owner wants the application that would enable him to identify the books that are available in the store. You are required to develop an application that can be used to keep records of the books in the stock.

There can be many different ways of identifying a book. The most direct way to identify book is by its name. However, it might lead to a situation where two books may be having same names. Therefore, additionally a book should be described by a unique identifier number. The unique identifier number for the book should be an assigned integer.

The book store owner also wants the names of the author or authors to be available along with the name and unique identifier number of any book. A book could be written by one or more than one authors. The name of an author consists of the first name and the last name. Since that might lead to a situation where two authors may have same names, so in addition to his or her name, the author should also be identified by a unique identifier number. The application should be so developed that it contains details about the authors, it should have the functionality to add the author or authors to the existing book records.

In your application, you should have appropriate methods that will enable the user to get names and the unique identifier numbers of the books as well as the names and unique identifier numbers of the corresponding author or authors of the book. The book may have one or several authors. The application should accommodate any number of authors.

Your application should be able to display on console (at the command prompt) the information about the books and authors, a sample output of which is as shown below.

Book ID: 1234986
Book Name: Gravitational Relativity
Author1 ID: 653
Author1 Name: Issac Newton
Author2 ID: 474
Author2 Name: Albert Einstein

For individual participants using TDD:

You have to use TDD and write relevant TestCases in developing the following application.

The owner of a book store wants to keep records of the books in stock on the computer. The owner wants the application that would enable him to identify the books that are available in the store. You are required to develop an application that can be used to keep records of the books in the stock.

There can be many different ways of identifying a book. The most direct way to identify book is by its name. However, it might lead to a situation where two books may be having same names. Therefore, additionally a book should be described by a unique identifier number. The unique identifier number for the book should be an assigned integer.

The book store owner also wants the names of the author or authors to be available along with the name and unique identifier number of any book. A book could be written by one or more than one authors. The name of an author consists of the first name and the last name. Since that might lead to a situation where two authors may have same names, so in addition to his or her name, the author should also be identified by a unique identifier number. The application should be so developed that it contains details about the authors, it should have the functionality to add the author or authors to the existing book records.

In your application, you should have appropriate methods that will enable the user to get names and the unique identifier numbers of the books as well as the names and unique identifier numbers of the corresponding author or authors of the book. The book may have one or several authors. The application should accommodate any number of authors.

Your application should be able to display on console (at the command prompt) the information about the books and authors, a sample output of which is as shown below.

Book ID: 1234986
Book Name: Gravitational Relativity
Author1 ID: 653
Author1 Name: Issac Newton
Author2 ID: 474
Author2 Name: Albert Einstein

For participants using TDD and working as collaborating pair:

You have to collaborate and work collectively as a pair on this task. You have to use TDD and write appropriate TestCases in developing the following application.

The owner of a book store wants to keep records of the books in stock on the computer. The owner wants the application that would enable him to identify the books that are available in the store. You are required to develop an application that can be used to keep records of the books in the stock.

There can be many different ways of identifying a book. The most direct way to identify book is by its name. However, it might lead to a situation where two books may be having same names. Therefore, additionally a book should be described by a unique identifier number. The unique identifier number for the book should be an assigned integer.

The book store owner also wants the names of the author or authors to be available along with the name and unique identifier number of any book. A book could be written by one or more than one authors. The name of an author consists of the first name and the last name. Since that might lead to a situation where two authors may have same names, so in addition to his or her name, the author should also be identified by a unique identifier number. The application should be so developed that it contains details about the authors, it should have the functionality to add the author or authors to the existing book records.

In your application, you should have appropriate methods that will enable the user to get names and the unique identifier numbers of the books as well as the names and unique identifier numbers of the corresponding author or authors of the book. The book may have one or several authors. The application should accommodate any number of authors.

Your application should be able to display on console (at the command prompt) the information about the books and authors, a sample output of which is as shown below.

Book ID: 1234986
Book Name: Gravitational Relativity
Author1 ID: 653
Author1 Name: Issac Newton
Author2 ID: 474
Author2 Name: Albert Einstein

APPENDIX F

QUESTIONNAIRE

For individual participants using Traditional method of software development:

1. Please circle your gender:
Male
Female

2. Please indicate your age on your last birthday _____

3. Highest educational level (including currently pursuing degree):
a) High school b) Technical school or community college
c) Undergraduate degree d) Graduate degree
e) Doctoral Degree f) Others: _____

4. Indicate number of years of your programming experience in any programming language?
a) 0 – 1 b) 2 – 3 c) 3 – 4
c) 4 – 5 d) 5 – 6 e) more than 6

5. Indicate number of years of your programming experience in object-oriented languages?
a) 0 – 1 b) 2 – 3 c) 3 – 4
c) 4 – 5 d) 5 – 6 e) more than 6

6. What would you consider to be your level of experience in object-oriented programming?
a) No experience b) Novice
c) Intermediate d) Expert

7. What object-orient programming languages are you familiar with?
a) C++ b) C# c) Java
d) Small Talk e) Objective-C f) Eiffel
g) Python h) VB.NET i) Others _____

8. How comfortable are you with the IDE “Eclipse”?
a) Very comfortable b) comfortable
c) Not much comfortable d) Not at all comfortable

Section A
Please fill in the blanks based upon the description given in the Main task:

The owner of a book store wants to keep records of the books in stock. The owner wants an application that would enable him to _____ the books that are available in the store by their _____. Additionally, a book should be described by a / an _____that should be an assigned _____. A book could be written by one or more than one authors. The _____ of an author consists of the _____ and _____ names. But, that might lead to a situation where two authors may have _____ names. So the author should also be identified by a / an _____ as well. The application should be so developed that it contains details about the authors, it should have the functionality to _____ the author or authors to the existing book records.

Section B
Answer the following questions based on the description given in the Main Task.

1. Which fields (variables and references) are used in Book Class?
a. Book Name
b. Book ID
c. Publisher
d. Both a and b

2. A Book object is identified by -
a. Book Name
b. Unique Identifier Number
c. Both a and b
d. Either a or b.

3. How many authors can be added to a Book?
a. One
b. Two
c. As many as needed

4. Can Book object be added to an author?
a. Yes.
b. No.
c. Insufficient Information
d.

5. An author is identified by -
a. Author name
b. Author ID number
c. Both a and b
d. Either a or b

6. Can we list all the books written by an author without going through the entire collection of the books?
a. Yes
b. No
c. Insufficient Information

7. Two authors who have the same name may be identified by -
a. First, Middle, Last name together
b. Unique Identifier Number
c. A randomly generated numeric value

8. When checking for the availability specific book, it is best to search by -
a. Name
b. Unique Identifier Number
c. Publisher
d. All of the above

9. If you want to store the publisher information in your application, which is more appropriate place to store the information?
a. Book class
b. Author class
c. Publisher class

10. The application that you developed for the scenario is similar to which of the following -
a. Customers opening an account in Bank
b. Students registering for classes in Student Information System
c. Customer receiving invoices

Section C
Please read the following scenario and answer the questions that follow:

A major international conference is to be organized in six months. The organizers of this conference have announced a call for papers. Many researchers are expected to submit their papers for publication in the conference journal. You are required to

develop an application that can be used to keep records of the papers that are submitted to the conference. The organizers want to easily identify the submitted papers. The submitted paper can be identified by its title. Since there can be a problem for two papers having same title, the organizers would also want to identify the submitted paper by a unique identifier number. Your application should take integer value for the unique identifier number.

Since the organizers wish to maintain the standard of the papers that are published in their conference journal, quality of the submitted work needs to be judged. For this purpose, the organizers have requested researchers to serve as reviewers. However, those who choose to volunteer as reviewers will not be allowed to submit their own papers. The submitted papers will be reviewed by the reviewers before being accepted for publication in the conference journal.

For the review process, the organizers should be able to assign each paper to the reviewers. Hence, a paper should have details about the reviewers. The papers could be reviewed by one or more reviewers and the conference organizers should be able to add the name or names of the reviewer or reviewers to a submitted paper. The name of a reviewer consists of first and last names. There could be a scenario of two reviewers with the same name, so in addition to the name, the reviewer should also be identified by a unique identifier number. The application should have the functionality to add the reviewer or reviewers to the existing records of the submitted papers.

1. As compared to the Main task, Paper is analogous to -
a. Book
b. Author
c. Publisher

2. As compared to the Main task, Organizer is analogous to -
a. Author
b. Owner
c. Publisher

3. As compared to the Main task, Reviewer is analogous to -
a. Author
b. Publisher
c. Owner

4. You will resolve the issue of two reviewers of same name by -
a. first, middle, and last name together
b. a randomly generated numeric value

c. Unique Identifier Number

5. Will ArrayList of Authors will be similar to ArrayList of -
a. Papers
b. Reviewers
c. Organizers

Section D:
Please answer the following questions based upon your experiences:

1. How do you feel about your overall experience on working on the programming task today?

1. How do you feel about your overall experience on working on the programming task today?

Very Dissatisfied 1 2 3 4 5 6 7 Very Satisfied

Very Displeased 1 2 3 4 5 6 7 Very Pleased

Very Frustrated 1 2 3 4 5 6 7 Very Contented

Absolutely Terrible 1 2 3 4 5 6 7 Absolutely Delighted

2. How do you feel about your own performance on the programming task today?

Very Dissatisfied 1 2 3 4 5 6 7 Very Satisfied

Very Displeased 1 2 3 4 5 6 7 Very Pleased

Very Frustrated 1 2 3 4 5 6 7 Very Contented

Absolutely Terrible 1 2 3 4 5 6 7 Absolutely Delighted

For individual participants using TDD:

1. Please circle your gender:
Male
Female

2. Please indicate your age on your last birthday _____

3. Highest educational level (including currently pursuing degree):
a) High school b) Technical school or community college
c) Undergraduate degree d) Graduate degree
e) Doctoral Degree f) Others: _____

4. Indicate number of years of your programming experience in any programming language?
a) 0 – 1 b) 2 – 3 c) 3 – 4
c) 4 – 5 d) 5 – 6 e) more than 6

5. Indicate number of years of your programming experience in object-oriented languages?
a) 0 – 1 b) 2 – 3 c) 3 – 4
c) 4 – 5 d) 5 – 6 e) more than 6

6. What would you consider to be your level of experience in object-oriented programming?
a) No experience b) Novice
c) Intermediate d) Expert

7. What object-orient programming languages are you familiar with?
a) C++ b) C# c) Java
d) Small Talk e) Objective-C f) Eiffel
g) Python h) VB.NET i) Others _____

8. How comfortable are you with the IDE “Eclipse”?
a) Very comfortable b) comfortable
c) Not much comfortable d) Not at all comfortable

9. What would you consider to be your level of experience in TDD?
a) No experience b) Novice
c) Intermediate d) Expert

Section A
Please fill in the blanks based upon the description given in the Main task:

The owner of a book store wants to keep records of the books in stock. The owner wants an application that would enable him to _____ the books that are available in the store by their _____. Additionally, a book should be described by a / an _____that should be an assigned _____. A book could be written by one or more than one authors. The _____ of an author consists of the _____ and _____ names. But, that might lead to a situation where two authors may have _____ names. So the author should also be identified by a / an _____ as well. The application should be so developed that it contains details about the authors, it should have the functionality to _____ the author or authors to the existing book records.

Section B
Answer the following questions based on the description given in the Main Task.

11. Which fields (variables and references) are used in Book Class?
a. Book Name
b. Book ID
c. Publisher
d. Both a and b

12. A Book object is identified by -
a. Book Name
b. Unique Identifier Number
c. Both a and b
d. Either a or b.

13. How many authors can be added to a Book?
a. One
b. Two
c. As many as needed

14. Can Book object be added to an author?
a. Yes.
b. No.
c. Insufficient Information

15. An author is identified by -
a. Author name

b. Author ID number
c. Both a and b
d. Either a or b

16. Can we list all the books written by an author without going through the entire collection of the books?
a. Yes
b. No
c. Insufficient Information

17. Two authors who have the same name may be identified by -
a. First, Middle, Last name together
b. Unique Identifier Number
c. A randomly generated numeric value

18. When checking for the availability specific book, it is best to search by -
a. Name
b. Unique Identifier Number
c. Publisher
d. All of the above

19. If you want to store the publisher information in your application, which is more appropriate place to store the information?
a. Book class
b. Author class
c. Publisher class

20. The application that you developed for the scenario is similar to which of the following -
a. Customers opening an account in Bank
b. Students registering for classes in Student Information System
c. Customer receiving invoices

Section C
Please read the following scenario and answer the questions that follow:

A major international conference is to be organized in six months. The organizers of this conference have announced a call for papers. Many researchers are expected to submit their papers for publication in the conference journal. You are required to develop an application that can be used to keep records of the papers that are submitted to the conference. The organizers want to easily identify the submitted papers. The

submitted paper can be identified by its title. Since there can be a problem for two papers having same title, the organizers would also want to identify the submitted paper by a unique identifier number. Your application should take integer value for the unique identifier number.

Since the organizers wish to maintain the standard of the papers that are published in their conference journal, quality of the submitted work needs to be judged. For this purpose, the organizers have requested researchers to serve as reviewers. However, those who choose to volunteer as reviewers will not be allowed to submit their own papers. The submitted papers will be reviewed by the reviewers before being accepted for publication in the conference journal.

For the review process, the organizers should be able to assign each paper to the reviewers. Hence, a paper should have details about the reviewers. The papers could be reviewed by one or more reviewers and the conference organizers should be able to add the name or names of the reviewer or reviewers to a submitted paper. The name of a reviewer consists of first and last names. There could be a scenario of two reviewers with the same name, so in addition to the name, the reviewer should also be identified by a unique identifier number. The application should have the functionality to add the reviewer or reviewers to the existing records of the submitted papers.

6.As compared to the Main task, Paper is analogous to -
   a. Book
   b. Author
   c. Publisher

7.As compared to the Main task, Organizer is analogous to -
   a. Author
   b. Owner
   c. Publisher

8.As compared to the Main task, Reviewer is analogous to -
   a. Author
   b. Publisher
   c. Owner

9.You will resolve the issue of two reviewers of same name by -
   a. first, middle, and last name together
   b. a randomly generated numeric value
   c. Unique Identifier Number

10. Will ArrayList of Authors will be similar to ArrayList of -
    a. Papers
    b. Reviewers
    c. Organizers

Section D:
Please answer the following questions based upon your experiences:

2. How do you feel about your overall experience on working on the programming task today?

3. How do you feel about your overall experience on working on the programming task today?

Very Dissatisfied 1 2 3 4 5 6 7 Very Satisfied

Very Displeased 1 2 3 4 5 6 7 Very Pleased

Very Frustrated 1 2 3 4 5 6 7 Very Contented

Absolutely Terrible 1 2 3 4 5 6 7 Absolutely Delighted

4. How do you feel about your own performance on the programming task today?

Very Dissatisfied 1 2 3 4 5 6 7 Very Satisfied

Very Displeased 1 2 3 4 5 6 7 Very Pleased

Very Frustrated 1 2 3 4 5 6 7 Very Contented

Absolutely Terrible 1 2 3 4 5 6 7 Absolutely Delighted

For participants using TDD and working as collaborating pair:

1. Please circle your gender:
Male
Female

2. Please indicate your age on your last birthday _____

3. Highest educational level (including currently pursuing degree):
a) High school b) Technical school or community college
c) Undergraduate degree d) Graduate degree
e) Doctoral Degree f) Others: _____

4. Indicate number of years of your programming experience in any programming language?
a) 0 – 1 b) 2 – 3 c) 3 – 4
c) 4 – 5 d) 5 – 6 e) more than 6

5. Indicate number of years of your programming experience in object-oriented languages?
a) 0 – 1 b) 2 – 3 c) 3 – 4
c) 4 – 5 d) 5 – 6 e) more than 6

6. What would you consider to be your level of experience in object-oriented programming?
a) No experience b) Novice
c) Intermediate d) Expert

7. What object-orient programming languages are you familiar with?
a) C++ b) C# c) Java
d) Small Talk e) Objective-C f) Eiffel
g) Python h) VB.NET i) Others _____

8. How comfortable are you with the IDE “Eclipse”?
a) Very comfortable b) comfortable
c) Not much comfortable d) Not at all comfortable

9. What would you consider to be your level of experience in TDD?
a) No experience b) Novice
c) Intermediate d) Expert

10. Before today’s assigned task, have you ever worked with your partner before?

Yes
No

Section A
Please fill in the blanks based upon the description given in the Main task:

The owner of a book store wants to keep records of the books in stock. The owner wants an application that would enable him to _____ the books that are available in the store by their _____. Additionally, a book should be described by a / an _____that should be an assigned _____. A book could be written by one or more than one authors. The _____ of an author consists of the _____ and _____ names. But, that might lead to a situation where two authors may have _____ names. So the author should also be identified by a / an _____ as well. The application should be so developed that it contains details about the authors, it should have the functionality to _____ the author or authors to the existing book records.

Section B
Answer the following questions based on the description given in the Main Task.

21. Which fields (variables and references) are used in Book Class?
a. Book Name
b. Book ID
c. Publisher
d. Both a and b

22. A Book object is identified by -
a. Book Name
b. Unique Identifier Number
c. Both a and b
d. Either a or b.

23. How many authors can be added to a Book?
a. One
b. Two
c. As many as needed

24. Can Book object be added to an author?
a. Yes.
b. No.
c. Insufficient Information

25. An author is identified by -
a. Author name
b. Author ID number
c. Both a and b
d. Either a or b

26. Can we list all the books written by an author without going through the entire collection of the books?
a. Yes
b. No
c. Insufficient Information

27. Two authors who have the same name may be identified by -
a. First, Middle, Last name together
b. Unique Identifier Number
c. A randomly generated numeric value

28. When checking for the availability specific book, it is best to search by -
a. Name
b. Unique Identifier Number
c. Publisher
d. All of the above

29. If you want to store the publisher information in your application, which is more appropriate place to store the information?
a. Book class
b. Author class
c. Publisher class

30. The application that you developed for the scenario is similar to which of the following -
a. Customers opening an account in Bank
b. Students registering for classes in Student Information System
c. Customer receiving invoices

Section C
Please read the following scenario and answer the questions that follow:

A major international conference is to be organized in six months. The organizers of this conference have announced a call for papers. Many researchers are expected to submit their papers for publication in the conference journal. You are required to develop an application that can be used to keep records of the papers that are submitted to the conference. The organizers want to easily identify the submitted papers. The submitted paper can be identified by its title. Since there can be a problem for two papers having same title, the organizers would also want to identify the submitted paper by a unique identifier number. Your application should take integer value for the unique identifier number.

Since the organizers wish to maintain the standard of the papers that are published in their conference journal, quality of the submitted work needs to be judged. For this purpose, the organizers have requested researchers to serve as reviewers. However, those who choose to volunteer as reviewers will not be allowed to submit their own papers. The submitted papers will be reviewed by the reviewers before being accepted for publication in the conference journal.

For the review process, the organizers should be able to assign each paper to the reviewers. Hence, a paper should have details about the reviewers. The papers could be reviewed by one or more reviewers and the conference organizers should be able to add the name or names of the reviewer or reviewers to a submitted paper. The name of a reviewer consists of first and last names. There could be a scenario of two reviewers with the same name, so in addition to the name, the reviewer should also be identified by a unique identifier number. The application should have the functionality to add the reviewer or reviewers to the existing records of the submitted papers.

11. As compared to the Main task, Paper is analogous to -
a. Book
b. Author
c. Publisher

12. As compared to the Main task, Organizer is analogous to -
a. Author
b. Owner
c. Publisher

13. As compared to the Main task, Reviewer is analogous to -
a. Author

b. Publisher
c. Owner

14. You will resolve the issue of two reviewers of same name by -
a. first, middle, and last name together
b. a randomly generated numeric value
c. Unique Identifier Number

15. Will ArrayList of Authors will be similar to ArrayList of -
a. Papers
b. Reviewers
c. Organizers

Section D:
Please answer the following questions based upon your experiences:

3. How do you feel about your overall experience on working on the programming task today?

5. How do you feel about your overall experience on working on the programming task today?

Very Dissatisfied 1 2 3 4 5 6 7 Very Satisfied

Very Displeased 1 2 3 4 5 6 7 Very Pleased

Very Frustrated 1 2 3 4 5 6 7 Very Contented

Absolutely Terrible 1 2 3 4 5 6 7 Absolutely Delighted

6. How do you feel about your own performance on the programming task today?

Very Dissatisfied 1 2 3 4 5 6 7 Very Satisfied

Very Displeased 1 2 3 4 5 6 7 Very Pleased

Very Frustrated 1 2 3 4 5 6 7 Very Contented

Absolutely Terrible 1 2 3 4 5 6 7 Absolutely Delighted

7. How do you feel about the performance of your group on the programming task today?

Very Dissatisfied 1 2 3 4 5 6 7 Very Satisfied

Very Displeased 1 2 3 4 5 6 7 Very Pleased

Very Frustrated 1 2 3 4 5 6 7 Very Contented

Absolutely Terrible 1 2 3 4 5 6 7 Absolutely Delighted

APPENDIX G

SOFTWARE QUALITY ASSESSMENT RUBRIC

RUBRIC FOR EXPERIMENT TASK EVALUATION Session - Room No. -				
Sno	**Description**	**Max Points**	**Points Scored**	**Remarks**
I	*Book Class Evaluation*			
A	Variables Declaration Name should be String ID Number – either String or int ArrayList (should be ArrayList<Author> authors = new ArrayList<Author>()) Deduct points for the following: 1 point for each wrong variable type 1 point if ArrayList syntax is wrong 2 points if Arrays are used instead of ArrayList	2 4 -1 -1 -2		

B	Constructor:
	public Book(String name, int or String number){
	this.name = name;
	this.number = number;
	}
	Deduct points for the following:
	3 points if it has a return type,
	2 points for incorrect parameters, and
	1 points if assignments are wrong
		8
		-3
		-2
		-1		

C	Gettor Methods Gettor for book name (eg. getBookName()) Gettor for book number (eg. getBookNumber()) Deduct points for the following in each case: 1 point for invalid return type, 1 point for incorrect parameters, and 1 point if assignments are wrong	3 3 -1 -1 -1		
D	addAuthor method (Author object should be passed as a parameter) Deduct points for the following: Wrong parameter (If Author object is not passed as parameter) Invalid return type(should be void in this case) Lack of Functionality (i.e., adding the author to the array list or array) Syntax errors (missing semi colon, braces etc.)	10 -3 -2 -3 -2		
E	getAuthor method (ArrayList of Authors should be iterated through and Author object read one by one) Deduct points for the following: Wrong implementation of Loop Invalid return type Lack of Functionality (i.e., not getting author object from ArrayList)	15 -4 -3 -5		

Sno	Description	Max Points	Points Scored	Remarks
II	*Author Class Evaluation*			
A	Variables Declaration First Name should be a String Last Name should be a String Author ID Number – either String or int Deduct points for the following: 1 point for **each** wrong variable type	 2 2 2 -1		

B	Constructor:
	public Author(String fname, String lname, int number){
	this.fname = fname;
	this.lname = lname;
	this.number = number;
	}
	Deduct points for the following:
	3 points if it has a return type,
	2 points for incorrect parameters, and
	1 points if assignments are wrong (in each case)
		10
		
		-3
		-2
		-1		

C	Gettor Methods Gettor for author name (eg.getAuthorName()) Gettor for author number (eg.getAuthorNumber()) Deduct points for the following in **each** case: 1 point for invalid return type, 1 point for incorrect parameters, and 1 point if assignments are wrong	 3 3 -1 -1 -1		
III	*Display Class with Main method:*			
A	Creating Book Objects passing correct parameter (at least one book object)	5		
B	Creating Author Objectspassing correct parameters (at least two author objects)	5		
C	Adding at least two Author objects to Book objects by Calling addAuthor Method	6		
D	Getting information from ArrayList of Authors using getAuthorList method	6		
E	Creating display at Command prompt	5		
F	If program compiles correctly displaying required information without any errors	5		
IV	*Going beyond requirements*			
A	Maintainability considerations: appropriate indentation, comments etc.	5		
B	Using Settor Methods	5		
C	Creation of User Interface using JOption Pane	6		
D	Creating additional class and/or methods to provide enhanced functionality	8		
	Total	**125**		

REFERENCES

Agarwal, R. & Karahanna, E. 2000, "Time Flies when You're having Fun: Cognitive Absorption and Beliefs about Information Technology Usage", *MIS Quarterly*, vol. 24, no. 4, pp. 665-694.

Agrawal, M. & Chari, K. 2007, "Software Effort, Quality, and Cycle Time: A Study of CMM Level 5 Projects", *IEEE Transactions on Software Engineering*, vol. 33, no. 3, pp. 145-156.

Ahuja, M.K., Chudoba, K.M., Kacmar, C.J., McKnight, D.H. & George, J.F. 2007, "It Road Warriors: Balancing Work--Family Conflict, Job Autonomy, and Work Overload to Mitigate Turnover Intentions", *MIS Quarterly*, vol. 31, pp. 1-17.

Alavi, M., Marakas, G.M. & Youngjin Yoo 2002, "A Comparative Study of Distributed Learning Environments on Learning Outcomes", *Information Systems Research*, vol. 13, no. 4, pp. 404-415.

Ambler, S.W. 2006, "Survey Says: Agile Works in Practice", *Dr.Dobb's Journal: The World of Software Development*, vol. 31, pp. 62-64.

Argote, L., Gruenfeld, D.H. & Naquin, C. 2001, "Group learning in organizations" in *Groups at work: Advances in theory and research*, ed. M.E. Turner, Lawrence Erlbaum Associates, Mahwah, NJ, pp. 369-411.

Armstrong, D.J. & Hardgrave, B.C. 2007, "Understanding Mindshift Learning: the Transition to Object-Oriented Development", *MIS Quarterly*, vol. 31, no. 3, pp. 453-474.

Asakawa, & Csikszentmihalyi, M. 1998, "The Quality of Experience of Asian American Adolescents in Academic Activities: An Exploration of Educational Achievement", *Journal of Research on Adolescence*, vol. 8, no. 2, pp. 241-262.

Ashforth, B.E. & Mael, F. 1989, "Social Identity Theory and the Organization", *Academy of Management Review*, vol. 14, no. 1, pp. 20.

Ashmore, R.D., Deaux, K. & McLaughlin-Volpe, T. 2004, "An Organizing Framework for Collective Identity: Articulation and Significance of Multidimensionality", *Psychological bulletin*, vol. 130, no. 1, pp. 80-114.

Balijepally V., Mahapatra R., Nerur S. & Price, K. 2009, “Are two heads better than one for software development? The productivity paradox of pair programming”, MIS Quarterly, vol. 33, no.1, pp. 91 – 118

Beck, K. 1999, Extreme Programming explained: embrace change, Addison-Wesley, Harlow.

Bhat, T. & Nagappan, N. 2006, "Evaluating the efficacy of Test-Driven Development: Industrial Case Studies", Proceedings of International Symposium on Empirical Software Engineering, (ISESE), ACM Press, September 21–22, 2006, Rio de Janeiro, Brazil

Blum, B.I. 1994, "A Taxonomy of Software Development Methods", *Communications of the ACM*, vol. 37, no. 11, pp. 82-94.

Brickner, M.A., Harkins, S.G. & Ostrom, T.M. 1986, "Effects of Personal Involvement: Thought-Provoking Implications for Social Loafing", *Journal of Personality and Social Psychology*, vol. 51, pp. 763-770.

Campion, M.A., Medsker, G.J. & Higgs, C.A. 1993, "Relations Between Work Group Characteristics and Effectiveness: Implications for Designing Effective Work Groups", *Personnel Psychology*, vol. 46, no. 4, pp. 823-850.

Canfora, A. 2006, "Evaluating Advantages of TDD: A Controlled Experiment with Professionals", Proceedings of International Symposium on Empirical Software Engineering, (ISESE), ACM Press, pp. 364-371

Carmines, E.G. & Zeller, R.A. 1979, "Reliability and Validity Assessment" Series: Quantitative Applications in Social Sciences. Ed. John Sullivan, Sage Publications.

Cohen, J. 1992, "A Power Primer", *Psychological bulletin*, vol. 112, no. 1, pp. 155-159.

Crispin, L. 2006, "Driving Software Quality: How Test-Driven Development Impacts Software Quality", *Software, IEEE*, vol. 23, no. 6, pp. 70-71.

Csikszentmihalyi, M. 1990, *Flow : the psychology of optimal experience*, Harper & Row, New York.

Csikszentmihalyi, M. 1975, *Beyond Boredom and Anxiety: Experiencing Flow in Work and Play*, Jossey-Bass Publishers, San Francisco.

Darcy, D.P., Kemerer, C.F., Slaughter, S.A. & Tomayko, J.E. 2005, "The Structural Complexity of Software: An Experimental Test", *IEEE Transactions on Software Engineering*, vol. 31, no. 11, pp. 982-995.

De Cremer, D. & Tyler, T.R. 2007, "The Effects of Trust in Authority and Procedural Fairness on Cooperation", *Journal of Applied Psychology*, vol. 92, no. 3, pp. 639-649.

Delespaul, P., Reis, H. & deVaries, M. 2004, "Ecological and Motivational Determinants of Activation: Studying compared to sports and watching TV", *Social Indicators Research*, vol. 67, pp. 129-143.

Edmondson, A. 2002, "The local and variegated nature of learning in organizations: A group level perspective", *Organization Science*, vol. 13, pp. 128-146.

Edwards, S. 2004, "Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action" ACM SIGSCE Bulletin, pp. 26–30.

Egbert, J. 2004, "A Study of Flow Theory in the Foreign Language Classroom", *Canadian Modern Language Review/ La Revue canadienne des langues vivantes*, vol. 60, no. 5, pp. 549-587.

Ellemers, N., De Gilder, D. & Haslam, S.A. 2004, "Motivating Individuals and Groups at Work: a Social Identity Perspective on Leadership and Group Performance", *Academy of Management Review*, vol. 29, no. 3, pp. 459-478.

Erdogmus, H., Morisio, M. & Torchiano, M. 2005, "On the Effectiveness of the Test-First Approach to Programming", *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 226-237.

Garibaldi, A.M. 1979, Affective Contributions of Cooperative and Group Goad Structures, *Journal of Educational Psychology*, 71, 788-794

Gemino, A.C. 1999, Empirical comparison of Systems Analysis Modeling Techniques, The University of British Columbia, Canada

George, B. & Williams, L. 2004, "A structured experiment of test-driven development", *Information & Software Technology*, vol. 46, pp. 337-342.

Gill, G.K. & Kemerer, C.F. 1991, "Cyclomatic Complexity Density and Software Maintenance Productivity", *IEEE Transactions on Software Engineering*, vol. 17, no. 12, pp. 1284-1288.

Guindon, R. 1990, "Designing the Design Process: Exploiting Opportunistic Thoughts", *Human-Computer Interaction*, vol. 5, no. 2, pp. 305-317.

Guo, Y., Klein, B., Ro, Y. & Rossin, D. 2007, "The impact of Flow on Learning Outcomes in a Graduate-Level Information Management Course", *Journal of Global Business Issues*, vol. 1, no. 2, pp. 31-39.

Gustavsen, B. 2004, "Making knowledge actionable", *Concepts & Transformation,* vol. 9, no. 2, pp. 147-180.

Hackman, J.R., Oldham, G., Janson, R. & Purdy, K. 1975, "A New Strategy for Job Enrichment", *California Management Review,* vol. 17, no. 4, pp. 57-64.

Hackman, J.R. 2003, "Learning more by crossing levels: evidence from airplanes, hospitals, and orchestras", *Journal of Organizational Behavior,* vol. 24, no. 8, pp. 905-922.

Hair, J.H., Black, W.C., Babin, B.J., Anderson, R.E., Tatham, R.L., 2006, Multivariate Data Analysis, Prentice Hall – Pearson Education

Hammell, T. 2005, *Test-Driven Development - A J2EE Example,* Apress, Springer-Verlag, New York.

Harkins, S.G. & Jackson, J.M. 1985, The Role of Evaluation in Eliminating Social Loafing. *Personality and Social Psychology,* 11, 575-584

Harkins, S.G. & Petty, R., E. 1982, "Effects of Task Difficulty and Task Uniqueness on Social Loafing", *Journal of Personality and Social Psychology,* vol. 43, no. 6, pp. 1229-1241.

Hertel, G., Deter, C. & Konradt, U. 2003, "Motivation Gains in Computer-Supported Groups", *Journal of Applied Social Psychology,* vol. 33, no. 10, pp. 2080-2105.

Hertel, G., Kerr, N.L. & Messé, L.A. 2000, "Motivation Gains in Performance Groups: Paradigmatic and Theoretical Developments on the Kohler Effect", *Journal of Personality & Social Psychology,* vol. 79, no. 4, pp. 580-601.

Hiltz, S.R. 1995, *The Virtual Classroom - Learning without limits via computer networks,* First edn, Ablex Publishing Corporation, Norwood, NJ.

Hinsz, V.B. & Nickell, G.S. 2004, "Positive Reactions to Working in Groups in a Study of Group and Individual Goal Decision Making.", *Group Dynamics,* vol. 8, no. 4, pp. 253-264.

Hoffman, D.L. & Novak, T.P. 1996, "Marketing in hypermedia computer-mediated environments: Conceptual foundations", *Journal of Marketing,* vol. 60, no. 3, pp. 50-55.

Iaffaldano, M.T. & Muchinsky, P.M. 1985, "Job Satisfaction and Job Performance: A Meta-Analysis", *Psychological Bulletin,* vol. 97, no. 2, pp. 251-273.

Jackson, J.M. & Harkins, S.G. 1985 Equity in Effort: An Explanation of the Social Loafing Effect. *Journal of Personality and Social Psychology*, 49(5), 1199-1206

Jackson, J. M. & Williams, K.D. 1985, Social Loafing on Difficult Tasks: Working collectively can improve performance. *Journal of Personality and Social Psychology*, 49(4), 937-942

Janzen, D. & Saiedian, H. 2005, "Test-driven development concepts, taxonomy, and future direction", *Computer,* vol. 38, no. 9, pp. 43-50.

Janzen, D.S. & Saiedian, H. 2006, "On the Influence of Test-Driven Development on Software Design", Proceedings of 19th Conference on Software Engineering Education and Training, pp. 141-148

Jeffries, R. & Melnik, G. 2007, "TDD: The Art of Fearless Programming", *IEEE Software,* vol. 24, no. 3, pp. 24-30.

Jetten, J., Spears, R. & Postmes, T. 2004, "Intergroup Distinctiveness and Differentiation: A Meta-Analytic Integration", *Journal of Personality & Social Psychology,* vol. 86, no. 6, pp. 862-879.

Judge, T.A., Thoresen, C.J., Bono, J.E. & Patton, G.K. 2001, "The Job Satisfaction-Job Performance Relationship: A Qualitative and Quantitative Review", *Psychological Bulletin,* vol. 127, no. 3, pp. 376-407.

Kerr, N.L. 1983 Motivation Losses in Groups: A Social Dilemma Analysis. *Journal of Personality & Social Psychology,* 45, 819-828.

Kerr, N.L. & Bruun, S. 1983 The Dispensability of Member Effort and Group Motivation Losses: Free Rider Effects. *Journal of Personality & Social Psychology,* 44, 78-94.

Karau, S.J. & Williams, K.D. 1993, "Social Loafing: A Meta-Analytic Review and Theoretical Integration", *Journal of Personality & Social Psychology,* vol. 65, no. 4, pp. 681-706.

Kerlinger, F. N. 1986, Foundations of Behavioral Research, Fort Worth, TX: Holt, Rinehart, & Winston, Inc.

Kerr, N.L. & Tindale, R.S. 2004, "Group Performance and Decision Making", Annual Review of Psychology, 55(1), 623-655.

Kolb, D.A. 1976, "Management and the Learning Process", *California management review,* vol. 18, no. 3, pp. 21-31.

Krishnan, M.S., Kriebel, C.H., Kekre, S. & Mukhopadhyay, T. 2000, "An Empirical Analysis of Productivity and Quality in Software Products", *Management Science,* vol. 46, no. 6, pp. 745-756.

Kuhn, D.R., Wallace, D.R. & Gallo Jr., A.M. 2004, "Software Fault Interactions and Implications for Software Testing", *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418-421.

Kutner, M.H., Nachtsheim, C.J., Neter, J. & Li, W. 2005, Applied Linear Statistical Models, Fifth Edition, Mc-GrawHill

Latane, B. 1981, "The Psychology of Social Impact", *American Psychologist*, vol. 36, pp. 343-356.

Laughlin, P.R., & Ellis, A.L. 1986, "Demonstrability and Social Combination Processes on Mathematical Intellective Tasks", *Journal of Experimental Social Psychology*, vol 22, pp. 177-189

Laughlin, P.R., Zander, M.L., Knievel, E.M., & Tan, T.K. 2003, "Groups perform better than the Best Individuals on Letters-to-Numbers Problems: Informative Equations and Effective Strategies", *Journal of Personality and Social Psychology*, vol 85, issue 4, pp. 684-694

Levi, D. 2001, *Group Dynamics for Teams*, Thousand Oaks, CA, Sage Publishers

Levine, J.M. & Moreland, R.L. 1998, "Small Groups" in *Handbook of Social Psychology*, eds. D.T. Gilbert & S.T. Fiske, Fourth ed, McGraw Hill, New York, NY, pp. 415-469.

Levine, J.M. & Thompson, L. 1996, "Conflict in groups" in *Social Psychology: Handbook of basic principles*, eds. E.T. Higgins & A.W. Kruglanski, First ed, Guilford Press, New York, pp. 745-776.

Locke, E.A. 1976, "The Nature and Causes of Job Satisfaction" in *Handbook of Industrial and Organizational Psychology*, ed. M.D. Dunnette, Rand McNally, Chicago, pp. 1297-1349.

Locke, E.A. & Latham, G.P. 1990, "Work Motivation and Satisfaction: Light at the End of the Tunnel", *Psychological Science*, vol. 1, no. 4, pp. 240-246.

Madeyski, L. 2005, "Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality" in *Software Engineering: Evolution and Emerging Technologies*, eds. K. Zelinski & T. Szmuc, First edn, IOS Press, UK, pp. 113-123.

Madrigal, R. 2006, "Measuring the Multidimensional nature of Sporting Event Performance Consumption", *Journal of Leisure Research*, vol. 38, no. 3, pp. 267-292.

Managalaraj, G. 2006, Influence of codified knowledge on software design task performance: a comparison of pairs with individuals, PhD dissertation, The University of Texas, Arlington.

Marquardt, M. 1999, *Action Learning in action,* Davies-Black, Palo Alto, CA.

Mayer, R. E. 1989, “Models for Understanding”, Review of Educational Research, Vol. 59, No. 1, pp 43-64

McCain, K.W. & Salvucci, L.J. 2006, "How influential is Brooks' Law? A longitudinal citation context analysis of Frederick Brooks' The Mythical Man-Month", *Journal of Information Science,* vol. 32, no. 3, pp. 277-295.

McDermid, J. & Rook, P. 1993, "Software Development Process Models" in *Software Engineer's Reference Book* CRC Press, pp. 26-28.

McGrath, J. E. 1984, *Groups: Interaction and Performance,* Englewood Cliffs, N.J., Prentice- Hall Inc.

Messé, L.A., Hertel, G., Kerr, N.L., Lount Jr., R.B. & Park, E.S. 2002, "Knowledge of Partner's Ability as a Moderator of Group Motivation Gains: An Exploration of the Kohler Discrepancy Effect", *Journal of Personality & Social Psychology,* vol. 82, no. 6, pp. 935-946.

Mithaug, D.E., Mithaug, D.K., Aagran, Martin, J. E. & Wehmeyer, M.L. 2003, *Self Determined Learning Theory: Construction, Verification, and Evaluation,* Lawrence Erlbaum Associates, Mahwah, New Jersey.

Mullen, B. 1983. Operationalizing the Effect of the Group on the Individual: A Self Attention Perspective, *Journal of Experimental Social Psychology,*19, 292-322

Mykytyn, P.P. 2007, "Educating Our Students in Computer Application Concepts: A Case for Problem-Based Learning", *Journal of Organizational & End User Computing,* vol. 19, no. 1, pp. 51-61.

Nelson, H. J., Armstrong, D.J. & Nelson, K.M. 2009, " Patterns of Transition: The Shift from Traditional to Object-Oriented Development ", *Journal of Management Information Systems*, 25, 4, 271–297

Nerur, S. & Balijepally, V. 2007, "Theoretical Reflections on Agile Development Methodologies", *Communications of the ACM,* vol. 50, no. 3, pp. 79-83.

Nurminen, J.K. 2003, "Using software complexity measures to analyze algorithms—an experiment with the shortest-paths algorithms", *Computers & Operations Research,* vol. 30, no. 8, pp. 1121-1130.

Nunnally, J.C. 1978, Psychometric Theory, New York, NY: McGraw-Hill

Olague, H.M., Etzkorn, L.H., Gholston, S. & Quattlebaum, S. 2007, "Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes", *IEEE Transactions on Software Engineering,* vol. 33, no. 6, pp. 402-419.

Pedler, M. 1991, *Action Learning in practice,* Grower, London.

Petty, M.M., McGee, G.W. & Cavender, J.W. 1984, "
A Meta-Analysis of the Relationships Between Individual Job Satisfaction and Individual Performance", *Academy of Management Review,* vol. 9, pp. 712-730.

PMI Report 2007, "Failure Is Not an Option", *PM Network,* vol. 21, no. 6, pp. 8-10.

Prahalad, C.K. & Krishnan, M.S. 1999 "The New Meaning of Quality in the Information Age", *Harvard business review,* vol. 77, no. 5, pp. 109-118.

Pressman, R.S. 2005, *Software Engineering - A Practitioner's Approach,* Sixth ed., McGraw-Hill International Edition, India.

Purao, S., Storey, V.C., and Han, T.D. 2003 " Improving analysis pattern reuse in conceptual designs: Augmenting automated processes with supervised learning," Information Systems Research (14:3), pp. 269-290

Ramasubbu, N., Mithas, S., Krishnan, M.S. & Kemerer, C.F. 2008, Work Dispersion, Process-Based Learning, and Offshore Software Development Performance, MIS Quarterly 32(2), 437-458

Robbins, S.P. 1998, *Organizational Behavior: Concepts, Controversies, Applications,* Fourth ed., Simon and Schuster, Upper Sadler River, NJ.

Shuell, T.J. 1990, "Teaching and Learning as Problem Solving", *Theory Into Practice,* vol. 29, no. 2, pp. 102-110.

Steele-Johnson, D., Beauregard, R.S., Hoover, P.B. & Schmidt, A.M. 2000, "Goal Orientation and Task Demand Effects on Motivation, Affect, and Performance", *Journal of Applied Psychology,* vol. 85, no. 5, pp. 724-738.

Steiner, I.D. & Rajaratnam, N.A. 1961, "A Model for the comparison of individual and group performance scores", *Behavioral Science,* vol. 6, pp. 142-147.

Subramanyam, R. & Krishnan, M.S. 2003, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects", *IEEE Transactions on Software Engineering,* vol. 29, no. 4, pp. 297-310.

Tajfel, H. 1978, *Differentiation between Social Groups*, First ed., Academic Press, London.

Tyler, T.R. & De Cremer, D. 2005, "Process-based leadership: Fair procedures and reactions to organizational change", *Leadership Quarterly*, vol. 16, no. 4, pp. 529-545.

Van Vugt, M. & De Cremer, D. 1999, "Leadership in Social Dilemmas: The Effects of Group Identification on Collective Actions to Provide Public Goods", *Journal of Personality & Social Psychology*, vol. 76, no. 4, pp. 587-599.

van Vugt, M. & Hart, C.M. 2004, "Social Identity as Social Glue: The Origins of Group Loyalty", *Journal of Personality & Social Psychology*, vol. 86, no. 4, pp. 585-598.

Wilson, J.M., Goodman, P.S. & Cronin, M.A. 2007, "Group Learning", *Academy of Management Review*, vol. 32, no. 4, pp. 1041-1059.

Yorks, L., O'Neil, J. & Marsick, V. 1999, *Action Learning: Theoretical bases and varieties of practices.*, Academy of Human Resources Development on Action Learning.

Zander, A.F. 1974, Productivity and Group Success: Team Spirit vs. the Individual Achiever, *Psychology Today*, 8(6), 64-68

BIOGRAPHICAL INFORMATION

Vikram Singh Bhadauria received his doctorate in Business Administration, with a major in Information Systems, from the University of Texas at Arlington. He holds a Masters in e-Commerce, an MBA, and a Bachelor in Civil Engineering from India. He has over eight years of experience in management, consultancy, and teaching. His current research interests include software development, open source software, knowledge management, and philosophical inquiry.