ABSTRACT

GEORGE, BOBY, Analysis and Quantification of Test Driven Development Approach. (Under the direction of Dr. Laurie Ann Williams.)

Software industry is increasingly becoming more demanding on development schedules and resources. Often, software production deals with ever-changing requirements and with development cycles measured in weeks or months. To respond to these demands and still produce high quality software, over years, software practitioners have developed a number of strategies. One of the more recent one is Test Driven Development (TDD). This is an emerging object-oriented development practice that purports to aid in producing high quality software quickly. TDD has been popularized through the Extreme Programming (XP) methodology. TDD proponents profess that, for small to mid-size software, the technique leads to quicker development of higher quality code. Anecdotal evidence supports this. However, until now there has been little quantitative empirical support for this TDD claim.

The work presented in this thesis is concerned with a set of structured TDD experiments on very small programs with pair programmers. Programmers were both students and professionals. In each programmer category (students and professionals), one group used TDD and the other (control group) a waterfall-like software development approach. The experiments provide some interesting observations regarding TDD.

When TDD was used, both student and professional TDD developers appear to achieve higher code quality, as measured using functional black box testing. The TDD student pairs passed 16% more test cases while TDD

professional pair passed 18% more test cases than the their corresponding control groups.

However, professional TDD developer pairs did spent about 16% more time on development. It was not established whether the increase in the quality was due to extra development time, or due to the TDD development process itself. On the other hand, the student experiments were time-limited. Both the TDD and the non-TDD student programmers had to complete the assignment in 75 minutes. Professional programmers took about 285 minutes on the average, to complete the same assignment. Consequently, the development cycle of the student-developed software was severely constrained and the resulting code was underdeveloped and of much poorer quality than the professional code. Still, it is interesting to note that the code developed using the TDD approach under these severe restrictions appears to be less faulty than the one developed using the more classical waterfall-like approach. It is conjectured that this may be due to the granularity of the TDD process, one to two test cases per feedback loop, which may encourage more frequent and tighter verification and validation episodes. These tighter cycles may result in a code that is better when compared to that developed by a coarser granularity waterfall-like model.

As part of the study, a survey was conducted of the participating programmers. The majority of the programmers thought that TDD was an effective approach, which improved their productivity.

Analysis and Quantification of Test Driven Development Approach

By
Boby George

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER SCIENCE

Raleigh

2002

APPROVED BY:

_____
Chair of Advisory Committee

PERSONAL BIOGRAPHY

Boby George is a graduate student in the Department of Computer Science, North Carolina State University. In 1999, he got his Bachelor of Technology in Computer Science and Engineering from University of Kerala, India. After his graduation, he worked as a system executive at eFunds International India Private Limited. His responsibilities included the automation of the testing process and preparation of test cases. In his graduate study, from August 2001 at NC State, Boby George focused on agile software development methodologies, in particular the Test Driven Development approach.

ACKNOWLEDGEMENTS

It is the hard work and contribution of many and not one that made this work possible. First and foremost, thanks to my research committee members, in particular the chair, Dr. Laurie Williams for proposing the topic and for all your persistent effort and guidance. A special thanks to Dr. Mladen Vouk for all your detailed review and suggestions that made this thesis work more accurate and to Dr. Aldo Dagnino, thank you for your keen interest in my research work.

To the Fall 2001 undergraduate software engineering students of North Carolina State University, the John Deere, RoleModel Software, and Ericsson developers, thank you for participating in the long and strenuous experiments. Also, I express my gratitude to Ken Auer for arranging the RoleModel Software experiment and giving valuable insight, to Doug Taylor for arranging John Deere experiment, and to Lisa Woodring for arranging the Ericsson experiment. Lastly, I express my appreciation to AT&T who provided the research funding for this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Research motivation

Software development is the evolution of a concept from design to implementation. But unlike other fields, the actual implementation need not and most often does not, correspond to the initial design. Unlike building a bridge from its design on paper, software development is more of a dynamic process. It is dynamic in that changes in requirements, development technologies and market conditions necessitate continuous modifications to the design of the software being developed. Michael Feathers, a leading consultant, asserts "There is little correlation between that [design] formality and quality of the resulting design. At times, there even appears to be an inverse correlation" [1].

Test Driven Development (TDD) [2], is a practice that Kent Beck, an originator of Extreme Programming (XP) [3-6], has been advocating. TDD is also known by other names such as, Test First Design (TFD), Test First Programming (TFP) and Test Driven Design (TDD) [1, 7]. The TDD approach evolves the design of a system starting with the unit test cases of an object. Writing test cases and implementing that object or object methods then triggers the need for other objects/methods.

An object is the basic building block of Object Oriented Programming (OOP). Unless objects are designed judiciously, dependency problems, such as tight coupling of objects or fragile super classes (inadequate encapsulation) can creep in. These problems could result in a large and complex code base that compiles and runs slowly. Rather than building objects that we think we need

(due to improper understanding of requirements or incomplete specifications), the TDD approach guides us to build objects we know we need (based on test cases). TDD proponents anecdotally contend that, for small to mid-size systems, this approach leads to the implementation of a simpler design than that would result from a detailed upfront design. Moreover, the continuous regression testing done with TDD appears to improve the code quality.

Although intriguing, the TDD concept needs exhaustive analysis. Software practitioners are sometimes concerned about the lack of upfront design coupled with the ensuing need to make design decisions at every stage of development. Also questions such as what to test and how to test exist. However, many software developers anecdotally contend that TDD is more effective than traditional methodologies [7, 8]. This necessitates the need to empirically analyze and quantify the effectiveness of this approach to validate or invalidate these anecdotal claims.

1.2 Historical perspective

In TDD the design, coding and testing phases are integrated into a single phase. Such processes are named as single-phase software development processes. The earliest single phase software development process is the ad-hoc model [9] (which was also the first software development model), where the focus was to deliver the code quickly. The lack of formal documentation in ad-hoc model led to repeatability problems, very poor quality control and code improvement troubles, as only the original developers were knowledgeable about the implementation details of the software developed [9].

A structured system development approach, or waterfall model [9], was introduced to reduce the problems associated with the ad-hoc model. However, the strict adherence to top-down sequential and independent phases (analysis, design, code and test) followed in waterfall model necessitated the need for through understanding of requirements. The problem faced by the structured development model (complete understanding of requirements) led to the development of evolutionary software development models (where software evolves in iterative cycles) which required less up-front information and offered greater flexibility [9]. Incremental model is an evolutionary software development approach which comprise of short increments of the waterfall model in an overlapping fashion [10].

Developing new systems, where requirements are unclear, involves considerable risks. Such risks can be reduced by techniques like prototyping [10]. Prototyping, in which a simplified replica of the system is built, helps to understand the feasibility of developing the proposed system and the development approach used. After the development approach is decided, the prototypes are discarded. Alternately, models in which the prototypes are grown and refined into the final product are named evolutionary prototyping models. A common evolutionary prototyping model is the spiral model [9, 10]. The spiral model provides potential for rapid development of incremental versions of software through the use of non-overlapping iterations [9].

Along with the improvements in development process, software testing strategies also underwent considerable changes. The most popular testing model

is the V model [11] that stresses close integration of testing with the software analysis and development phases. The model specifies that unit testing verifies code implementation, integration testing verifies low-level design, system testing verifies system design, and acceptance testing validates business requirements [11]. Hence, the model (in which testing is done parallel with development) increases the prominence of testing throughout the development process.

In order to achieve a higher success rate in meeting the system objectives, the software models need to include Verification and Validation (V&V). Verification focuses on building the software right, while validation focuses on building the right software [12]. The IEEE Standard for software verification and validation plans [13] specifies that V&V be performed in parallel with software development. Iterative in nature, the V&V standard guides early defect detection and correction, reduced project risk and cost, and enhanced software quality and reliability.

The development of the TDD approach appears to be inspired from existing software development and testing strategies. TDD is based on bottom-up implementation strategy, where the implementation starts with lowest modules or lowest level of code. The various other development/testing strategies on which TDD appears to be based include incremental (as the approach proceeds in increments) and the V-model (as the approach also specifies for a tight test-code cycle). However, TDD seems to differ from other models such as the V-model in the level of granularity (the number of test cases written before implementation code per feedback loop). TDD specifies a higher granularity i.e.

one or two unit test be written first then the implementation code instead of thinking about several unit tests before code as specified in the V-model. The incremental, V-model bottom-up strategy that TDD seems to follow necessitates writing test drivers and mock objects to simulate the modules that are not implemented yet. A comparison of TDD with other software models is included in section 2.5.

1.3 Research approach

This research empirically examines TDD approach to see if the following three hypotheses hold:

- The TDD approach will yield code with superior external code quality when compared with code developed with a more traditional waterfall-like model practice. External code quality will be assessed based on the number of functional (black box test cases) test cases passed.
- The TDD approach will yield code with superior internal code quality when compared with code developed with a more traditional waterfall-like model practice. Internal code quality will be assessed based on established Object-Oriented Design (OOD) metrics. The definition of superior value or desirable value is specified in table 2.
- Programmers who practice TDD approach will develop code quicker, than developers who develop code with a more traditional waterfall-like model practice. Programmer speed will be measured by the time to complete (hours) a specified program.

To investigate these hypotheses, research data was collected from structured experiments conducted with student and professional developers, who developed a very small application using OOP methodology. The experiments were conducted four times – first with 138 advanced undergraduate (juniors and seniors) students and then three times with professional developers (eight developers each from three companies).

All developers were randomly assigned to two groups, TDD and non-TDD pairs. Since all developers were familiar with pair programming concept (a practice where two programmers develop software side by side in one computer) [14] and two professional organizations development groups used pair programming for their day-to-day programming, all experiments were conducted with pair programming practice. Each pair was asked to develop a bowling game application. The typical size of the code developed was less than 200 lines of code (LOC). The specifications of this game were adapted from a XP episode [15] and are attached in Appendices F and G. The non-TDD pairs developed the application using the conventional waterfall-like approach while the TDD pairs used test-then-code approach. The controlled experiment environment ensured that the developer pairs used their assigned approaches.

The code developed was analyzed for quality based on functional test cases and on object-oriented quality metrics. The functional test cases used for the evaluation (of both TDD and non-TDD codes) were developed by the researcher and were different from the test cases the pairs developed. Additionally, the amount of time taken by each professional pair was recorded.

Code coverage analysis was done to understand the efficiency of the test cases written by the developers. Lastly, the qualitative impressions of the developers on the effectiveness of the technique were also recorded.

1.4 Thesis layout

The remainder of this thesis starts with a presentation on related work, including a brief introduction to XP. The second chapter explains TDD and its related topics, unit testing, existing software models and refactoring. Also a comparison of the test-then-code model and code-then-test model with TDD is presented in chapter 2. Chapter 3 focuses on testing, in particular unit testing, and the related concepts such as coverage and boundary analysis. Chapter 4 delineates traditional the OOP approach and explains in detail the working principle of TDD with an example. Chapter 5 lists the details of the experiment design and the talks about the various statistical tests used in this research work. Chapter 6 presents the external validity issues of the experiments conducted and analyses the experimental data to draw inferences and conclusions on the effectiveness of TDD. Chapter 7 summarizes the major conclusions of this research and suggests future related research. The Appendices expound each of the experiments conducted.

2. RELATED WORK

This chapter provides a survey of concepts relevant to TDD starting with the XP methodology and TDD as practice of XP. The chapter succinctly explains the TDD model, other test-then-code and code-then-test models followed by a comparison of the models. The chapter concludes with a discussion on the software metrics (and their limitations).

2.1 Extreme Programming (XP)

The current software industry faces an uncertain environment as stated by the Uncertainty Principle - “Uncertainty is inherent and inevitable in software development process and products” [16, 17]. However, the need for producing high quality software has become paramount due to increased competition. Although processes like Personal Software Process (PSP) [17] and Rational Unified Process (RUP) [18] provide guidelines on delivering quality software, the industry is concerned with delivering quality software faster while handling requirement changes efficiently. These concerns have led to the development of agile software development models [19]. Among the principles of agile models, the following are noteworthy: use light-but-sufficient rules of project behavior and use human- and communication-oriented rules. Agile models view process secondary to the primary goal of delivering software, i.e. the development process should adapt to project requirements.

In the late 1980s, Kent Beck, Ward Cunningham, and Ron Jeffries formulated XP, the most popular agile methodology [20]. XP (which is based on four values: simplicity, communication, feedback and courage,) is a compilation

of twelve practices of software development. TDD is one of those twelve practices. Major accomplishments of XP include the development of the Chrysler Comprehensive Compensation (C3) system, an application consisting of 2,000 classes and 30,000 methods, and Fords' VCAPS system [21, 22]. However, like TDD, the success of XP is anecdotal. The popularity of XP is often attributed to the sound approach it takes in dealing with complexity and constant changes occurring in today's software projects [23].

Fowler [24] attributes the success of the XP methodology to its strong emphasis on testing. XP relies on TDD for code development. It might be claimed that XP achieves its values of simplicity, courage and feedback by practicing TDD. The simplicity advocated in XP's principle - "Do the simplest thing that could possibly work" is achieved by TDD's approach of implementing only the minimal set of objects required. With TDD, test cases are written for each and every function. The existence of this strong testing framework aids developers in being confident in making required modifications without "breaking" existing code. Finally, the higher granularity of test-code cycle gives the developer constant feedback on defects in the code.

2.2 Unit Testing

Unit testing, performed by developers [25], is the practice of testing a unit of code (like a class in OOP) in isolation to verify whether all code in the individual unit performs as expected (that is, the unit gives proper results). The proper results are derived from system requirements and design specification. The aim of unit testing is to find defects in the software and not to demonstrate

the absence of the same [26]. Unit testing is very effective when done frequently and independently (i.e. running a unit test does not affect other unit tests). Unit testing independently reduces testing complexity and eliminates any side effects that might occur when all tests run together.

Proponents of Object Oriented (OO) development and agile methodologies advocate writing unit test code in parallel with code [27]. While many developers recommend to, not only think but also write unit test cases before writing production code [25]. Conversely, some developers strongly argue that, ideally, system testing (integrated testing of the application being developed) is the only required testing and unit testing is done to increase the speed of development [28].

Testing all the permutations and combinations of inputs is not practical. Hence, the set of tests to be performed should be decided prudently. Testing is often not practiced by developers. As a result, when pressure to meet deadlines increase, thorough testing might not take place [29]. As a result, unit testing demands a change in the practices of many developers.

JUnit[1], developed by Kent Beck and Erich Gamma, is a popular testing framework tool for unit testing applications written in Java. The tool provides a framework for creating test cases, stubs, and harnesses with which unit testing can be automated for regression testing. Such testing stabilizes (decrease the defect rate) the system [25]. Similar to JUnit, there exists a series of xUnits

[1] http://www.junit.org/index.htm

(HttpUnit[2], CppUnit[3], DelphiUnit[4] and others) for different languages. Currently, the xUnit frameworks lack capabilities for testing Graphical User Interface (GUI).

2.3 Software Models

The waterfall model, also called as the linear sequential model, is a systematic, sequential approach to software development which was originally suggested by Winston Royce in 1970 [30]. The model, which originally had feedback loops, consisted of four main phase: analysis (requirements are gathered and documented), design (translation of the requirements into a representation of software), code (translation of design into machine-readable form) and test (verifies the correctness of the implementation) [12]. Although waterfall is the oldest model in use, it has many problems such as difficulty in stating all the requirements explicitly early in the process and long gestation period and blocking states due to the sequential nature of work flow [31, 32] . The non-TDD pairs (control group) in our experiment used the waterfall-like model for code development (i.e. they had a design, code and test phases followed by a debugging phase).

As stated before, the V-model emphasizes tight integration of testing in all phases of software development. The V-model, defined by Paul Rook in 1980, aims to improve the efficiency and effectiveness of software development [11]. The model depicts the entire process as “V” with development activities on left side (going downwards) and the corresponding test activities needed on the right

[2] http://www.httpunit.org
[3] http://cppunit.sourceforge.net/
[4] http://dunit.sourceforge.net/

side (going upwards) [11]. The various stages of lifecycle and the test cases that need to be developed in parallel include business requirements (develop acceptance test cases), high-level design (develop system test cases), low-level design (develop integration tests) and code (develop unit tests). As with TDD, the V-model specifies that the design of test cases must occur first (before the coding phase) and that various levels of testing be done in parallel with development phases [33]. Detractors of the V-model contend that the model fails to define in which sequence the units need to be build and tested, and that the model is suited only when the requirements are clearly defined [33].

2.4 Test Driven Development (TDD)

TDD is an emerging practice that is professed to produce high quality software quickly, while dealing with inevitable design and requirement changes. The TDD approach, used sporadically by developers for decades [34], was recently popularized by XP methodology. With TDD, automated unit test code is written before implementation code is written. New functionality is not considered to be properly implemented unless unit test cases for the new functionality and every other unit test cases ever written for the code base succeed. An important rule in TDD is: “If you can’t write test for what you are about to code, then you shouldn’t even be thinking about coding” [7]. Also when a defect is found later in the development cycle or by the customer, it is a TDD practice to write unit test cases that exposes the defect, before fixing it.

Some believe that strict adherence to TDD can help to minimize, if not eliminate, the need for upfront design. In addition, the practice is highly flexible

and can be adapted to any process methodology, including those that specify low level (detailed) upfront design phases. In any process, there exists a gap between the decision (design) and feedback (performance obtained due to the implementation of that design). The success of TDD is attributed to the lowering, if not elimination, of that gap, as the test-then-code cycle gives constant feedback on the design performance to the programmer [2]. Hence it is this higher granularity of test-then-code cycle in TDD that differentiates the approach from other testing and development models.

Proponents of the TDD approach anecdotally list the following benefits:

- TDD induces developers to write code that is automatically testable [35]. If programs are written without consideration towards being automatically testable, writing such test cases afterwards can be very difficult, if not impossible. Benefits of automated testing include: (1) production of a reliable system; (2) improvement in quality of the test effort; and (3) reduction of the test effort and minimization of the schedule [36].
- The TDD approach encourages simpler design [8]. The design stays simple, as only required and sufficient code is implemented. Complex designs are more prone to be defective as explained by the Complexity Barrier [33] law - “Software complexity and (therefore that of defects) grow to the limits of our ability to manage that complexity”.
- The TDD approach results in faster code development as more time is spent developing code rather than testing/debugging [7].

- Historically, “debugging is a bottleneck in programming” [37]. Due to the test-code cycle of TDD, the developer gets an instant feedback on the correctness of the code [35].
- Incorporating late design changes to the code base is safer, as regression testing is a norm in TDD [7]. By running automated regression tests, one can easily check whether a change broke any existing logic. This builds confidence in making changes without worrying about side affects.
- Proper unit test cases can be written only if the developer understands the system requirements properly [8]. So, TDD motivates developers to understanding what is expected of the system being developed.

2.5 Differences between TDD and other models

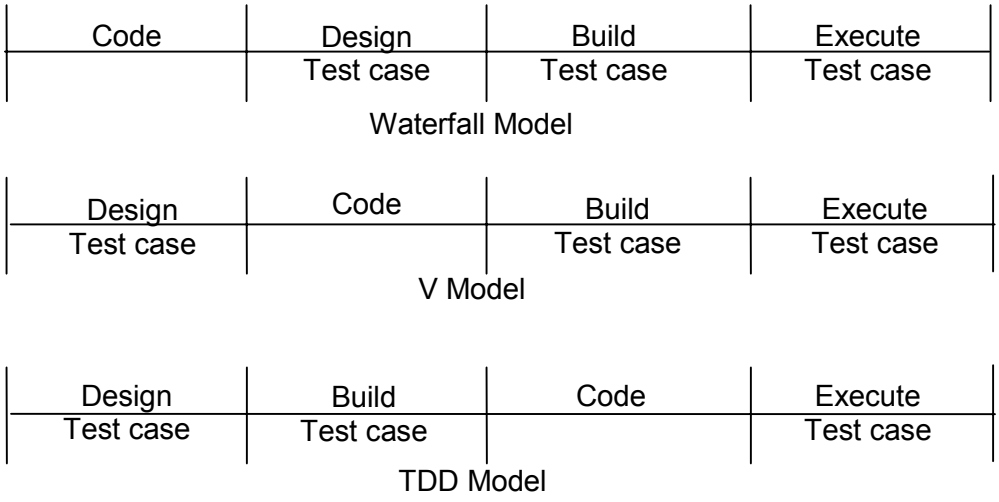
TDD approach differs from other testing approaches in two key areas. First, the approach discourages the development of high- and low-level design prior to commencing code implementation (referred to as Big Design Up Front (BDUF) [4] in XP). As the development progress, the initial design changes (with the discovery of new objects and conditions) more often than in other processes. This is due to the fact that initial design is not well thought prior to coding.

The second difference is the degree to which testing is performed in TDD. As discussed, several development models such as v Model specify that testing be done in parallel with development. The possible test case scenarios are thought about and the required conditions to pass those test cases are included in the implementation. Later, the test cases are run to verify that the implementation is correct. In TDD, this process is taken one step further; the test

cases are actually implemented in a very incremental fashion. First, a few (i.e. between one and three) test cases are written and run to make sure that they fail (if it passes then either the test case is wrong or the implementation already exist). Then, the code implementation is written to pass those few test cases. Hence in TDD, the test cases are thought out and written before the production code is written and the execution of test cases is more frequent when compared with other testing approaches. The degree of granularity of the test-code cycle is higher in TDD approach.

Figure 1 diagrammatically compares the waterfall model (used by non-TDD pairs), the V-model, and the TDD approach based on the occurrence of the various sub-phases of unit testing. The unit testing process can be subdivided into the three sub-phases namely identifying what and how to test (designing test cases), building the tests and executing them [11]. In waterfall model, all the sub-phases of the unit testing occurs after code is written, while in V-model, the design of the test cases occurs before the coding and incase of TDD, the design and building of test cases occurs before the coding.

Figure 1: Comparison of various Software Models



As TDD is closely integrated with XP, the stress on documentation is considerably lower. The proponents of the approach claim that the source code and the test cases will suffice the need for the required documentation [38]. Section 4.1.1 delineates the limitations in requirements gathering and reusability issues that exist in the present object-oriented development practices.

2.6 Refactoring

Often poorly designed programs, created by less-experienced developers, are reused. This results in code that is difficult to modify or maintain [39]. “Refactoring is the process of changing a software system in such a way that that it does not alter the external behavior of the code, yet improves its internal structure” [39]. The first step in refactoring is to build a solid set of automated unit tests for the code being refactored. As stated by Martin Fowler “If you want to refactor, the essential precondition is having solid tests” [39]. Refactoring should always be done in small steps so that if mistakes are made, it is easy to spot the incorrectly refactored code. Usually done before new functionality is

incorporated, refactoring is said to improve the design and understandability of code, and is professed to allow the faster development of new code [39]. However, such claims are anecdotal.

Refactoring has its share of limitations, of which the prominent one is performance loss. Hence, for large or life critical systems (real time), refactoring might not be appropriate [39, 40]. Additionally, refactoring public and published interfaces is challenging and should be done only when the side effects are well bounded.

In TDD, code refactoring is recommended after implementation of any functionality. The TDD approach relies on the professed advantages of refactoring to obtain simplicity and elimination of duplicate code; thus refactoring compensates for the lack of upfront design.

2.7 Pair programming

Pair programming is a technique in which two developers continuously collaborate by working side-by-side at one computer on the same design, algorithm, code and test [14]. One developer, the driver, has control of the keyboard and is responsible for the development. The other, the navigator, observes the work of the driver, intervening whenever the driver makes syntax and logical errors. Additionally, the driver and navigator are continuous brainstorming partners. The driver and navigator irrespective of their software expertise switch roles at regular interval. Like TDD, pair programming has been practiced sporadically for many years and has been recently popularized by XP [14]. The professed benefits of pair programming include tacit knowledge

management and improved code quality. Experiments conducted in 1999 at the University of Utah with undergraduate students showed that pair programmers produced better quality code with minimal increase in programmer hours [14].

2.8 Object Oriented Metrics

Object Oriented Metrics (OOM) were developed to effectively assess the quality of the code structure so that classes that may contain faults or are costly to maintain can be spotted easily [41]. Empirical results widely support theoretical conclusions on the use of metrics to evaluate and predict software quality [42-45]. Some established OOM suites include Chidamber Kemerer (CK) [46] and Metrics for Object Oriented Design (MOOD) [47].

Limitations of metrics

ISO/IEC international standard (14598) on software product quality states, “Internal metrics are of little value unless there is evidence that they are related to external quality”. Churcher [48] and El Emam [49] express reservations on the validity of metrics. They contend that metrics tend to exclude external factors such as programmers’ physical and mental stress. The validation of these metrics requires demonstrating, convincingly that (1) the metric measures what it purports to measure (for example, a coupling metric really measures coupling) and (2) the metric is a good early indication of an important external attribute, such as reliability, maintainability and fault-proneness [49]. Some argue that the validity of OOD metrics needs further refinement to take into account the recent developments in ODD [41, 50]. Further, some metrics (for example, LOC metric) are language and programming style dependent and hence, cannot be used

across languages and sometimes, across developers [51]. Lastly, since metric threshold values (benchmark values) are determined from previous project experiences, they cannot be used to measure quality in projects based on a new language or design.

We addressed both the capabilities and the limitations of OOM. To gain any possible valuable information, we evaluated our internal code quality based on the following metrics:

Weighted Methods per Class (WMC)

WMC [46], a traditional metric indicating the level of complexity, measures the number of methods in a class excluding methods from ancestor (parent) and other classes, such as friend classes. Historically, a lower WMC value indicates lower complexity, which in turn means lesser effort and time. However, recent trends support a larger number of smaller methods per class. For example, refactoring guides developers to break their code into larger number of smaller methods, in accordance with the OOP principle – “a method should do only one function” [39].

Response For Class (RFC)

RFC [46], is the cardinality of all methods that are invoked (executed) in response to a message a class receives (either from outside or from inside the class). This metrics evaluates the complexity of code in terms of the number of methods in a class and the amount of communication with other classes. Usually, low value of RFC is desired as the complexity increases with increase in number

of methods and amount of communication, leading to reduced maintainability and reuse difficulty.

Coupling

Coupling indicates the interdependencies between objects. Coupling is measured using metrics such as Coupling Between Objects (CBO) and Coupling Factor (CF). CBO indicates the dependency of an object to other objects. CF is the ratio of the number of non-heritance couplings to the total number of couplings in a system. Empirical evidence supports lower values for CBO and CF to ensure (1) better maintainability (2) lower enhancement cost and defect propagation.

Lines of Code (LOC)

LOC, one of the oldest metrics, can be used to assess design quality based on the number of lines in a piece of code. The effectiveness of the metrics is still debated, since the metric is affected by the programming style used and language syntax. In general, for projects with identical functionality, a lower value of LOC indicates superior design [52] and lower projected maintenance costs.

Encapsulation

Encapsulation or information hiding, an OOP principle, hides the data associated with an object. Interaction with other objects is achieved through public interfaces - a set of well defined methods. Encapsulation is a proven theoretical technique that increases maintainability (ease of modification) by a factor of four [53]. For this research, encapsulation is measured using two metrics - Attribute Hiding Factor (AHF) and Method Hiding Factor (MHF). AHF, a

fraction, measures the visibility (accessibility from other classes) of attributes in classes. Hiding attributes, achieved by declaring as private, is a desirable quality and hence AHF value should be large. MHF, a fraction, indicates the visibility of methods in classes. The code reusability is improved by hiding methods, except the ones acting as public interfaces, and hence a large value for MHF is desirable.

3. TESTING METHODOLOGIES

TDD, as the name states, is a test-driven approach. Because of its strong focus on testing, a brief discussion of testing in OO development and related concepts such as unit testing and code coverage are included in this chapter. The chapter concludes with a short note of the limitations of the testing process.

3.1 Testing methodology in OO Development

Software testing is done to improve quality and to ascertain the proper functioning of software. However, testing is not a substitute for weak software development process. Primarily, the quality of software is often a function of the quality of the software process/methodology used [54]. In OO development, testing is a recommended ongoing activity. Grady Booch, Ed Berard and many others recommend the “Recursive/Parallel Model” as the most appropriate for object oriented life cycle in which testing is continuously interwoven with the development process [55]. This model can be paraphrased as “Analyze a little, design a little, code a little, and test a little, repeat”.

OO development testing strategies are different from procedural testing strategies [27] due to characteristics of OOP, such as encapsulation and complex inheritance relationships. As in traditional testing methodologies, discrete testing phases such as system, integration, user acceptance, stress and performance tests are done. However, depending upon test requirements, the frequency of testing is intended to be higher in OO development. After each development increment, the code is to be unit and functionally tested [27]. This incremental approach to testing increases the detection of problems earlier in the

development phase thereby, lowering cost. The cost of defect correction rises exponentially with advancement of software life cycle and is minimized if defects are detected earlier in the cycle [27]. Hence, it could be stated that testing has a very high prominence in OO development.

Gelperin and Hetzel [56] summarize the past four decades of software testing into four testing models: Demonstration, Destruction, Evaluation and Prevention, of which the latest, the prevention model, started in 1988. The primary goal of the prevention model is defect prevention. Although defect prevention is better than detection and subsequent correction, complete defect prevention is unachievable. Hence, the model has a secondary goal, defect detection [33]. Based on the prevention model and IEEE standards for testing [57], a comprehensive methodology called Systematic Test and Evaluation Process (STEP) [58] was formulated. The usage of STEP, in which testing is done in parallel with development, lead to significant improvement in software quality [56]. In fact, from as early as 1974 improvement in code quality was noted when testing was done in parallel with development.

> *The test planning must begin early in the development cycle…testers look at the specifications from an entirely different point of view than do the developers. As such, they can frequently spot flaws in the logic of the design, or the incompleteness of the specification. In doing this, they help the developers to produce a superior product, and accelerate the program development. [59]*

The act of “designing test” rather than the act of testing (i.e. knowing what to test and how to test) is one of the best known defect preventers [33]. Although highly speculative, the concept of “test then code” could have developed from this realization:

> *The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded-indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding, and the release. [33]*

Gelperin and Hetzel originally advocated the concept of “test then code” back in 1987 [34]. TDD is actually the revival of this concept.

3.2 Unit testing

Typically, three distinct types of testing are done on software systems: unit/component testing, integration testing and system testing [33]. Each testing type takes a different approach to detect the presence of defects. The ANSI/IEEE standard for unit testing [57] defines unit testing as a process that has three activities:

1. *Perform test planning phase.* Planning the general approach to unit testing, followed by determining the features and elements to be tested, and refining the overall plan are done in this phase.
2. *Acquire test set phase*. In this phase, sets of tests are designed and the refined plan and designed tests are implemented.
3. *Measure test unit phase.* During this phase, the unit test are executed and evaluated for their effectiveness.

When more than one unit is to be tested, acquire and measure activities are performed at least once (for each unit). However, the planning phase is done only once (to address the total set of test units). The objective of unit testing is stated as follows:

Unit testing is the testing we do to show that the unit does not satisfy its functional specification and/or that its implemented structure does not match the intended design structure, [33]

A unit is defined as the smallest testable piece of software or the smallest code that can be compiled or assembled, linked, loaded and put in the control of a test harness or driver [33]. A component is an integrated aggregate of one or more units, and component testing is done to verify the functional specification of the component [33]. The developers who know the internal working of the unit perform unit testing. Experiments ranging from no testing to entire testing performed by developers have validated the conjecture that unit/component testing and software quality responsibility should be handled by developers [33]. Ideally, the unit tests should test each line of code, invoke all paths of branches directly or indirectly, verify correctness of operations at normal and limiting parameter values, verify normal and abnormal termination of loops and recursions, verify proper handling of files and data structures, and evaluate timing/synchronization issues and hardware dependencies, if any [60]. In short, unit testing strives to achieve defect free unit code to a considerable extent [17].

Automating the execution and verification of unit tests can minimize the time needed for executing unit tests. An automated testing framework can provide a mechanism to combine and run all the test cases and to produce a compiled report on the results, without the intervention of the user. When automated sets of unit tests are accumulated in a test suite and run more often, unit testing can also be considered regression testing. Ideally, after every defect fix, regression testing is done to ensure that nothing else has broken due to the

code fix. Automated unit testing can help to attain the frequency of testing necessary for TDD.

Unit testing (in fact software testing itself) is one kind of verification to ensure that software is build correctly. Through unit testing, it is verified that the detailed design of the unit has been correctly implemented [61]. The IEEE standards on software V&V advocates for the following tasks during the implementation phase [13]:

(1) Source Code Traceability Analysis. Trace source code to design specifications and vice versa to ensure that the design and code are accurate, complete and consistent.

(2) Source Code Evaluation. Evaluate source code for correctness, consistency and testability to verify that the code is compliant with established standards and practices.

(3) Source Code Interface Analysis. Evaluate source code with hardware, operator and software interface design documentation for correctness, consistency and completeness.

(4) Source Code Documentation Evaluation. Evaluate draft code-related documents with source code to ensure completeness, correctness and consistency.

(5) Test Case Generation. Develop test cases for unit testing, integration testing, system testing and acceptance testing.

(6) Test Procedure Generation. Develop test procedures for the test cases generated.

(7) Component Test Execution. Perform unit testing as required by unit test procedures. Analyze results to determine the correctness of the implementation and trace/document results as required by the test plan.

3.3 Various Types of Testing

Apart from unit testing, other types of testing include integration, system and acceptance testing [62]. Integration testing, done by developers, verifies that the unit tested components function according to the design requirements after integrated with other unit tested components. The integration usually occurs incrementally either in a top-down or bottom-up manner. Regression testing is vital during integration testing, because the code base changes when new modules are integrated. Further, whenever a defect is corrected, the entire integration test cases should be re-executed to ensure that the correction did not break any existing logic. The TDD approach supports the incremental building of such an automated regression test suite.

System testing is a series of different tests aimed at fully evaluating the system developed that has software as component (other components might include hardware, people, etc). The objective of system testing is to verify that system elements have been properly integrated and perform their intended functions. The system testing, a black box testing methodology normally performed by QA engineers, validates the overall requirements in terms of quality, security and performance aspects of the system.

Acceptance testing, done by clients usually at their own site, validates that the software build was according to requirements. The objective of acceptance

testing to enable customer understand the capabilities of the system developed and thereby determine whether to accept the system or reject it [62]. Normally, acceptance testing (validation) is done through a series of black box tests that demonstrate conformity with requirements [12].

3.3 Testability

Software testability measures the ease with which software can be tested. Testability is important because it increases the failure rate of software during development and testing phases thereby increasing defect detection rate before the software is released the quality of the software [63]. The IEEE Standard Glossary of Software Engineering Terminology [64] defines software testability as "(1) the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met, and (2) the degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met." A less formal definition for software testability was given Miller and Voas [65] and can be paraphrased as the probability that a piece of software (that has a fault) will fail on its next execution (with a particular assumed input distribution) during testing. Hence testability, according to latter definition, helps to identify the faults that are hidden during particular testing schemes/test cases, thereby reducing the likelihood of residual faults. Accurate mathematical predictions of such likelihood are vital while developing life- and safety-critical applications; such methods are also used in non-critical applications.

Pressman [12] provides a set of characteristics that will lead to testable software.

1. *Operability.* More efficient testing can be done on a system that works better.
2. *Observability*. Internals such as system states and variable are more visible of the internals during execution
3. *Controllability*. Better control over software leads to more tests that can be automated and optimized.
4. *Decomposability*. Tests should be independent, reducing the scope of each test.
5. *Simplicity*. Tests should be simple, which leads to more frequent testing
6. *Stability*. Fewer changes to code and test cases leads to fewer disruptions in testing.
7. *Understandability*. More understanding of the software leads to smarter tests.

These characteristics help to produce better sampling of test inputs with high failure rates, thereby exposing the faults present in the software during development cycle. Software design-for-testability focuses on designing code in a way that if faults exist in a code then upon execution the code will “fail big” exposing the defect [65]. Software process such as TDD’s constant emphasis on writing test cases along with code development creates a testable system [2].

3.4 Mean Time Between Failure

Software failures are normally caused due to design or implementation faults that remain undetected during development phase and materialize as failure in customer environment [66]. Although one can assess quality as the number of defects/LOC metrics, which gives a good indication of the number of faults in released software, a typical software user is more concerned about failures (downtime) that occurs due to those faults [12]. Estimating and/or measuring the failure rate provides an indication of the operational capabilities or reliability of the software i.e. the probability of failure-free operation of a computer program [66].

Mean Time Between Failure (MTBF) measures software reliability as a measure of downtime. MTBF gives the statistical mean time a component is expected to work before it fails. MTBF is found out by summing the Mean Time To Failure (MTTF) and Mean Time To Repair (MTTR). MTTF gives the average amount of time the system is up, while MTTR gives the average amount of time needed to repair the system. Since many software failures are transient and recoverable, some are critical of the use of MTBF as a measure of software reliability [67].

3.5 Code coverage

The quality of a test suite or the test adequacy criteria is often assessed based on code coverage, a measure of test completeness. Code coverage, also called test coverage, is a measure of the percentage of code executed by a test

or a set of tests. Code coverage analysis is important because the probability of finding new defects is considered to be inversely proportional to the amount of code not yet covered by the test cases [54]. The code coverage analyses measured in this research include: statement coverage (the ratio of executed statements to the total number of statements); branch coverage (the ratio of branch paths covered to the total number of branch paths that exist); and method coverage (ratio of methods executed during test runs to total number of methods in the code). For any project it is recommended to choose a minimum coverage objective that must be attained by the testing process (based on available testing resources and importance of defect prevention post release) before the software is released. The normal recommended coverage percentage is 80% - 90% in statement and branch coverage [68]. Since complete testing is not possible, it is advisable to find a subset that gives the highest code coverage, thereby, increasing the error detection rate.

Boundary analysis is used to test the behavior of a program for inputs that lie at the input domain's border. It not only tests inputs that are valid and lie on the boundary, but also inputs that are just off the boundary and those that are invalid. There exists four types of boundary analysis (based on test case adequacy criteria): the N* 1 domain adequacy (has at least N test cases on the boundary and at least one test case just off the boundary), N * N domain adequacy (has at least N test cases on the boundary and at least N linearly independent test cases off the boundary), V * V domain adequacy (a test case for each vertex v or point of intersection of several boundaries) and N + 2 domain

adequacy (has at least N +2 test cases for each boundary) [69]. While the first three types of boundary analysis techniques are effective for domains with linear functions at the border, N + 2 domain adequacy is effective for the detection of errors for nonlinear domains [69].

3.6 Limitations of testing process

Software testing has a number of limitations. It has been proven that complete testing is impossible to achieve. Although every program operates on a finite number of inputs, testing of each possible input and path through the code is theoretically and practically impossible [33]. Moreover, the pesticide paradox and complexity barrier increases the defect rate. The pesticide paradox states, “Every method you use to prevent or find defects leaves a residue of subtler defects against which those methods are ineffectual” [33]. The complexity barrier states that “software complexity (and therefore that of defects) grows to the limits of our ability to manage that complexity” [33]. Since complete testing is not possible, testing is stopped when the probability of defect decreases to inconsequential level [33]. The inconsequential level is determined by various factors, such as business specifications and/or technical requirements.

4. TEST DRIVEN DEVELOPMENT

This chapter starts with a survey on the current OO development practices, followed by detailed explanation of the TDD approach. This section also explains the strategic approach TDD takes to achieve reusable code.

4.1 Traditional OOP approaches

The OOP development approach centers on modeling the real world in terms of objects, in contrast to older approaches, which focuses on a function-oriented view (separate data from procedures that operate on data) [46]. The most common OOP approaches are: the Booch method, the Rumbaugh method, the Jacobson method, the Coad and Yourdon method, and the Wirfs-Brock method [12]. Recently, Booch, Rumbaugh and Jacobson methods were combined to form the Unified Modeling Language (UML) [70]. UML is based on object modeling (the object model is regarded as a physical model, simulating the behavior of either a real or imaginary part of the world) and is widely used [12]. UML represents the system using five different views, but the identification of objects is left to developers' discretion [12].

Most OO approaches are similar in concept, but differ in methodologies used for identifying and implementing objects. The OO development methodology, as outlined by Booch [71], consists four steps that can be categorized into Object Oriented Analysis (OOA) followed by Object Relationship Model (ORM), Object Behavior Model (OBM) and finally OOD. OOA identifies the objects (along with their attributes, behavior and relationship to other objects) that are needed. The objects are identified either by grammatical parsing or

identifying the physical devices looking for persistent data. Coad and Yourdon [72] identifies six selection characteristics to identify potential objects. The decision to include objects is some what subjective and needs considerable experience on the part of developers [12]. The attributes and operations (behavior) are defined for the objects. All these are done by parsing/examining the problem statement [12]. The decomposition of the system into objects is the most important and difficult part [12, 73].

OOA establishes the first iteration of classes and their relationship. ORM is used to properly define the relationship between the objects and can be derived in three steps [12]:

- From the classes obtained using OOA, a network of collaborator objects is drawn.
- The list of objects is reviewed for responsibilities and collaborators and each unlabeled connection between objects is labeled.
- The cardinality of the relationship is evaluated.

The third step, OBM, establishes the behavior of the objects. Analyzing the use cases and identifying all the events in the system establishes the behavior of objects. The events are then mapped to objects and a State Transition Diagram (STD) is built. Finally the STD is verified for accuracy and consistency. The final step is to implement the classes after OOD (detailed low level design of classes) is done.

Mathematically, OOD can be represented as an ordered tuple consisting of a set of elements, a set of relations and a set of binary operations [71],

$D \equiv (A, R_1 \ldots R_n, O_1 \ldots O_m)$

Where,

D is OOD represented as a relational system.

A is a set of object-elements.

$R_1 \ldots R_n$ are empirical relations on objects-elements of A.

$O_1 \ldots O_m$ are binary operations on elements of A.

Again, using mathematical notions, an object can be represented as substantial individuals (token or identity by which object is represented in a system) that possess properties [46],

$X = \langle x, p(x) \rangle$

Where

X is the Object.

x is the substantial individual (token or identity).

p(x) is the finite collection of x's properties.

The proposed benefits of OOP include encapsulation (leading to data protection and tight coupling between data and its operations), reusability and understandability [12, 74]. The above mentioned advantages are achieved using concepts such as inheritance, polymorphism and to some extent overriding. Design patterns form the basis for developing reusable designs [73]. However designing reusable object-oriented systems is hard, as ideally the system should have the right balance between reusability (generalization), simplicity (specialization), flexibility (for ease of update), and granularity.

The traditional object oriented methodologies place significant emphasis on upfront design. The general assumption is that this emphasis will result in proper reuse of code, there by lowering development time. The professed strategic payoff in using an OO development approach (instead of a functional approach) is that requirement changes can be incorporated easily (that is, maintainability or ease of modification of the code increases). For initial development of a system, OO development might require more development time. However, this investment pays off with subsequent iterations (for maintenance) in which the time needed will be reduced when compared to a non-OOP system. Hence, the overall lifecycle cost will be less. OO development demands an evolutionary approach to software development [12].

4.1.1 Limitations of traditional OO development approaches

> *The overly simplistic object technology hype is that you take a set of requirements; build an object model that identifies classes relevant to the problem and the relationships between the classes; add design classes relevant to the solution; code the classes; and that's about it. Unfortunately, this hype is believed by many people. [27]*

The flaws in the above hype are many. The initial set of requirements is rarely correct. As pointed out by Alistair Cockburn, a leading agile software methodology proponent, perfection in requirements gathering can never be achieved [19]. In *Agile Software Development* [19], Cockburn shows via an example how each person is influenced by his/her own interpretations when parsing a pattern or developing software. He argues that it is not possible to produce a perfectly written requirements specification. Since it is not possible to

achieve perfection, is it not better to learn how to manage this imperfection? Moreover, design involves much more than adding classes [27].

In spite of the wide acceptance of OO development, the software industry is still facing the “software crisis,” indicating that the OO development approach might not be the silver bullet. Reusability is often the most often claimed goal in OO development. In traditional OOP, reusability is achieved by predicting (through experience and client interactions) which code portion might get reused and then incorporating the appropriate design early in the process. Over the years, however, there has been a realization that reusable components are evolved as the project progresses, rather than being designed early on in the project [75]. Hence accurate prediction might not be possible.

4.2 TDD Explained

TDD is not just a testing technique; it is also an analysis and design technique. While thinking of and writing test cases, essentially an analysis of what the system will do and will not do occurs i.e. the scope of the code gets explicitly stated. With TDD, software development starts with the identification of an object followed by continuous iterations through the steps of (1) writing unit tests (preferably automated), and then (2) writing implementation code to pass these and all other tests previously written for the code base. This cycle of test-then-code, is in accord with the IEEE V&V standard recommendations. It is repeated until all the requirements are implemented. Finally, the modules of the code are refactored, i.e., restructured as necessary, to improve design while still passing the same unit regression test cases. TDD enhancements with respect to

traditional test and code methods lie in explicit insistence on incrementally writing test case before code implementation and continuous refactoring.

The following example, adapted from Kent Beck's article [8], explains the TDD approach in detail. An application is to be developed that will return a person's mortality table, given the person's age. When using TDD, the developer starts by finding a single important object; the problem statement is analyzed to find that one object. For this example, an ideal object would be `MortalityTable`, the resultant table expected from the application. Thinking about what can be tested in the `MortalityTable` object results in writing the unit test to check whether a table actually exists:

```
MortalityTable actualTable = new MortalityTable (bob);
AssertEquals (expectedTable, actualTable);
```

So, to create a `MortalityTable` object to test, a Person (another class), "bob" can be created and passed as parameter to the `MortalityTable` object's constructor. The Person object, bob, might be created using the following constructor:

```
Person bob = new Person ("Bob", "Newhart", 48, NON-SMOKER,
```

At this point, the programmer might realize that this constructor takes a lot of arguments (and that tests have to be written for each). This immediate feedback of "the need to write tests" could force the programmer to consider a simpler way. Actually, to get a mortality table, all that is required is a person's age and whether they smoke. Hence there is no need to create an object named Person. This results in a change in test case to:

```
MortalityTable actualTable = new MortalityTable (48, NON-
SMOKER);
```

As seen, a design change occurred, resulting in a simpler design (the reduction of an object). Considerable expertise in OO development might be required to create simpler design, but TDD seems to reduce the level of expertise required because it provides constant feedback on the design solution chosen through test cases. As seen in the example, the subtle, yet important, event that occurs is the process of elimination and creation of objects while thinking about what to test. The general rule of thumb is that if no useful test can be thought of for an object then that object can be eliminated.

After writing test cases that generally will not even compile, implementation code is written to pass these test cases. Incrementally, every test case that can be thought of (to handle new functionality, to handle error conditions, etc) is written. First test cases are written, then code is implemented, a few more test cases are written, more code is implemented, and so on.

Before additional code is written, the unit being implemented is refactored to eliminate duplicate code and to keep the unit simple and manageable. Refactoring is a good practice in any methodology, though its criticality is lessened when architecture and design have been completed and inspected [39]. Code developed with TDD is integrated frequently with the code base. After a unit is fully developed, integration test cases are written and the unit is integrated with other units. Regression testing is done during integration phase also. In most methodologies, including TDD, the customer/client runs acceptance tests on the product. Testable code is developed throughout the process. This testable

code enables developers to develop white box test cases to test the logic of the (black box) acceptance test cases; these extensive test cases are included in the automated test suite and are executed often. Using this technique, the development team can feel more confident of success when the client runs the acceptance test cases themselves.

4.2.1 TDD without High/Low Level Design

> *Instead of adding a "proof" step to programmers' chores, why not simply adopt programming practices that "attract" correct code as a limit function, not as an absolute value. If you write unit tests for every feature, and if you refactor to simplify code between each step, and if you add features one at a time and only after all the unit tests pass, you will create what mathematicians call an "iterative dynamic attractor". This is a point in a phase space that all flows converge on. Code is more likely to change for the better over time instead of for the worse; the attractor approaches correctness as a limit function.*
>
> – TDD Programmer on google groups

With XP, developers do not formally prepare high or low-level designs. They work in pairs and informally apply any design techniques they feel are appropriate for the problem at hand. For example, they might brainstorm the class structure by performing a CRC card session [76] or draft out a UML class diagram [70, 77] that is not inspected, archived or shared. Alternatively, they start implementation by writing automated unit test cases without any design. They understand that they might not arrive at the “best” design the first time they implement the code. For developers who work in this way, refactoring, as proposed by Martin Fowler [18], in concert with TDD is essential. After successfully writing test cases and implementing the code for new functionality, the developers look back through the code to see if any structural improvements

can be made. The automated test cases give the developer the confidence that the structural improvements they make do not alter the behavior of the system. It is this iterative refinement through test cases and refactoring that yields the *iterative dynamic attractor* behavior referenced in the above quote.

4.2.2 Evolutionary software process models and TDD

Evolutionary software process models are iterative in nature and are often used when it is desirable for the product to evolve over time [12]. Since TDD is also an iterative approach that drives the evolution of the product over a period of time, it shares considerable characteristics of evolutionary software process models. However, there exist some differences between the approaches. Evolutionary software process models, such as spiral model, have upfront design and requirement analysis phases in their iteration cycles at project level. Also for some models, such as incremental model, each iteration-cycle time is very high when compared with TDD approach. The purpose of iterations in evolutionary models is to provide a feedback mechanism to the customer and developers about on technical risks and performance of the product (customer evaluation). In TDD, these feedback cycles are tighter due to higher granularity of the test-code cycle.

One of the mistakes that can be made while designing software is confusing various project stages, such as requirement gathering and analysis, with development processes [78]. Since process models order the project stages either linearly or cyclically, developers assume that they should perform software development tasks in stages or iteration cycles. Thus the sequence used to

guide development at project level gets mapped onto behavior occurring at individual level. However, in research performed at Microelectronics and Computer Technology Corporation (MCC), a R&D consortium of 11 technology companies, it was shown that the behavior of individual designers while designing software (especially when evolutionary models are used) is better characterized as “opportunistic process” [78]. That is, in making design decisions the designers move back and forth between levels of abstraction ranging from application domain issues to detailed design to coding issues. This movement is neither sequential nor cyclic and is initiated by the recognition of issues that may occur at any level of abstraction. In TDD such movements are more explicitly as the design changes according to the recognition or issues and needs.

In evolutionary prototyping, (the most common model being the spiral model), the prototype developed is continuously improved through out the iterations. The TDD approach is also similar to the spiral model, in the sense that both models can be adapted to apply throughout the life of computer software in comparison to other process that stop at product release. The evolutionary models, in particular the spiral model, have its own set of limitations. It might be difficult to convince the customer that evolutionary approaches are controllable (as iteration cycles can go on forever). Further, spiral model requires considerable risk assessment expertise (unless a risk is not uncovered and managed, problems will occur). In comparison, TDD’s iteration cycles are smaller and well defined. Also, risk management is not a focus area in TDD as the

process focus more on implementation level. Hence risk management has to be done external to the approach.

4.2.3 Reusability, Design Patterns and TDD

TDD takes a tactical approach to reuse. Instead of trying to predict the future, TDD waits until the required reusability becomes clear and is abstracted into the design at that time. Advocates of TDD, state that this approach, although it results in late design changes, is more efficient since only the required abstractions are incorporated[5]. To achieve reusability and maintainability, the code must be tested thoroughly for defects and changes must be localized to a particular section of the code. TDD's test-then-code approach hence seems capable of achieving reusability when it is needed.

Questions are raised on how well or poorly TDD can be composed with the use of Design Patterns (DP) [79]. DPs are a compilation of well-known practical approaches taken by experienced developers. Although DP promotes proper reuse and offer "good" design solutions, developers (especially less experienced) tend to face difficulties in selecting and implementing the right DP [79, 80]. Hence, choosing the right DP for a particular solution might become easier if there exist a feedback on the cost and benefits of using that pattern for that problem. Critics of DP argue that incorrect DP usage tend to promote complex design as DP encourages the use of abstraction and a "design by contract" approach [81]. When patterns are used inappropriately, unwanted objects may be created leading to the need for increased testing effort. Although

[5] Quote from Robert Martin in Extreme Programming newsgroup

seemingly contradictory concepts, TDD (code-for-the-current-needs approach) and DP (code-for-future-needs approach) might work hand in hand [79] by making use of TDD's ability to bring simplicity into design by providing continuous feedback to the developers on design decisions.

Critics of TDD argue that implementing the simplest design with constant refactoring is a concern (proper reuse might not occur) [79]. However, when DP is combined with TDD, the developer is provided a continuous feedback on the chosen pattern. The increased testing effort (thinking about what and how to test) an inappropriate DP might prompt the developer to self-justify the DP choice (based on current or assured, but not predicted, requirement needs) or choose another DP. The continuous feedback provided by TDD might enable the developer to iterate through the patterns, till the right pattern is found, thereby making the solution simpler (due to TDD) while “properly” implementing reuse (since DP is used).

5. EXPERIMENT DESIGN

This chapter discusses the various procedures and methods used in conducting software engineering experimentations. The chapter also explains the various statistical variables, tests and primary hypothesis of the tests used for this research.

5.1 Basics of Software Engineering Experimentation

Experimentation in software engineering is the process of matching facts with the suppositions, assumptions, speculations and beliefs in software construction [82]. There exist two different approaches to run empirical experimentations: quantitative experiments, which aim to obtain numerical relationship between several variables, and qualitative experiments, which aim to interpret or examine objects in terms of explanations that people offer. Of the two approaches, quantitative investigation is professed to generate more justifiable and formal results than qualitative [82]. However, qualitative investigation is helpful in providing an overall view or to support quantitative findings. Hence, these two types of analysis are complementary and not competitive in nature. This research work uses both approaches.

Normally, experiments are conducted in two stages (levels) – laboratory and real world. First-level laboratory experimentation is done in a strictly controlled environment by researchers/the inventor of the idea, where the proposed idea is initially checked for any promising results. The environment is less controlled and made more realistic when positive results are obtained. For software engineering experiments, a laboratory experiment is a development

project that is not subjected to any pressures, such as market and budget, while other factors such as process and techniques used can be controlled. The second levels of experiments are conducted on real projects. This research is done at the first level, where the environment is strictly controlled.

There exist three reasoning methods for arriving at a hypothesis: deduction (proves something must be), induction (shows something is really operational) and abduction (suggests that something could be) [82]. By means of deduction, a preliminary hypothesis can be compared against data (collected through various means such as experimentation and observation) for its validity. If the data obtained does not coincide with the consequence of the hypothesis, then the discrepancy can lead, by means of induction and abduction, to the hypothesis being modified. The modified hypothesis is again compared with data to check for coincidence. Since this research is not based on any previous TDD findings, the hypotheses are preliminary in nature and tentatively examine the effectiveness of the approach.

5.2 Basics of Statistics

Statistics plays an extremely important role in the quantitative experimentation. An experimenter often faces difficulty in discovering and understanding complex effects as well as relationships that exists between the variables present in an experiment [82]. This problem is aggravated when data gets contaminated with errors due to reporting and measuring. Statistical methods such as data analysis are used to help solve these problems. Of the two types of statistics, descriptive and inferential, the basic features of data collected

can be described using descriptive statistics while inferential statistics is used to derive implications from the data. Descriptive statistics, used to present quantitative descriptions of data, has two common numerical indices: (1) Measures of Central Tendency and (2) Measures of Variability [83]. Both measures of central tendency and measures of variability are used to describe the distribution of results in this research.

5.2.1 Measures of Central Tendency

Used to represent the typical score in a distribution, measures of central tendency can be described in terms of mode, median and mean. Mean is defined as the probability weighted numerical average of all scores in the distribution [83]; often it is assume that the weights are the same. Median is the score found at the exact middle of the distribution or 50^{th} percentile. Mode gives the most frequent score. Different measures of central tendency exist as no one measure can describe every possible data set. Unlike mean, median is considered to represents the “typical” value in a data set and is resistant to variations caused by outlier data points [84]. However mean (which is unique) takes into account the total of all observations and always exist.

5.2.2 Measures of Variability

Measures of variability indicate the spread of the scores or how the scores in a distribution differ from one another. The two common measures of variability are Inter-Quartile Range (IQR) and Standard Deviation (SD). One of the methods for handling outlier scores is to exclude portions of data that are at the extremes

of the distribution, as done with IQR. IQR, a measure of the spread, is the difference between the 75^{th} percentile and the 25^{th} percentile [84]. As with median, IQR is also resistant to wild swings from a single observation. SD indicates the average deviation of the sample values from the mean (average value). In other words, it represents the measure of error in an experimental quantity.

For this research work, mean and median are used to measure the central tendency, while SD and IQR are used to measure the variability. The median and IQR were included as common summaries because mean and SD can change due to a single or a small number of irregular measurements [46].

5.2.3 Box Plots

A box plot is a common way to display the distribution of values in an experiment. It demonstrates graphically the quartiles of a distribution. The edges of the box mark the 25^{th} and 75^{th} percentiles, while the horizontal line at the center of box marks the median of distribution. The range of the distribution on each side is shown by “whiskers” or vertical lines that extend from the box. Box plots give a good idea of the center and spread of the distribution.

5.3 Statistical significant analysis

Due to the limitations in experiments and data recording, statistical significant analysis was not done with numerical results. The survey results obtained from experiments are analyzed for statistical significance. As a part of qualitative investigations, these inferential studies are conducted by running

statistical tests on data collected. Essentially these test checks whether the differences in results are large enough not to occur by chance, i.e. the probability of the results to occur due to chance are low. Normally p (probability) values of 0.05 or less are considered as statistically significant. In this research work, the qualitative survey results were examined using normal approximation and the Spearman's Rho test.

5.3.1 Normal approximation

Normal approximation is used check for the statistical significance of a large data set consisting of single variable results that is categorical and has only two values (for example yes and no) [83]. The comments of the student developers of this research work were evaluated using normal approximation. The comments were first grouped into three categories – positive, negative and neutral. Then, neutral and negative were grouped into one; this group was compared with the positive comment group to evaluate the statistical significance. The null hypothesis tested stated that the proportions of students who thought that TDD was effective (reflected by positive comments) and the proportion of students who thought TDD was not effective (reflected by negative and neutral comments) are equal. The alternative hypothesis stated that proportions of students who thought that TDD was effective were greater than the proportion of students who thought TDD was not effective.

5.3.2 Spearman's Rho Test

The Spearman's rho test is used to evaluate the statistical significance for ordinal data sets. Spearman's rho is a measure of the linear relationship between two variables that are measured using ranking scheme. In this research work the survey results that were ordinal in nature were evaluated for their statistical significance using Spearman's rho test. Spearman's rho test was chosen since the responses were first converted into ranks.

6. EXPERIMENTAL RESULTS

This chapter quantifies and analyzes the effectiveness of TDD approach based on the data obtained from four experiments involving students of North Carolina State University (NCSU) and professional developers. The chapter discusses in detail the validity of the experiments and the quantitative and qualitative analysis was conducted on the results obtained from the experiments.

6.1 Experiment Details

The experiments will now be briefly described; specifics of the research design can be found in Appendix A. The experiments were conducted in two phases, first with students and then with professional programmers. All the participants were randomly assigned to one of two groups, TDD and non-TDD pairs. Both the groups were asked to develop a bowling game with TDD pairs using the TDD approach (see Appendix F), while non-TDD pairs used a waterfall-like software development approach (see Appendix G). All groups coded using the pair programming technique.

The first set of experiments, conducted in the fall of 2001, was run with 138 computer science students (30 TDD and 37 non-TDD pairs). Previous demonstrations with the bowling game (with somewhat similar specifications) lead us to believe the students could complete the program in the allotted time. However, the students did not, leading us to use this experiment to evaluate the effectiveness of the TDD approach in a time-constrained environment. Prior to the start of the experiment, students studied the TDD technique, got familiarized with the JUnit tool, and then completed two homework assignments utilizing the

technique and the tool. They then participated in the experiment during one class laboratory period. After the experiment, the TDD pairs were asked to comment on the TDD approach.

The next sets of experiments were conducted on small, eight-person groups of developers from three companies (John Deere, RoleModel Software and Ericsson) to evaluate the effectiveness of TDD in a professional, non-time constrained environment. The professional developers had varying levels (novice to expert) of experience with the TDD approach. In order to balance the time TDD pairs spent in writing test cases, the non-TDD developers (excluding John Deere non-TDD pairs) were asked to write test cases after development (in a code then test manner). Prior to the experiment, a short survey (found in Appendix E) was distributed to collect developers' impression of the TDD approach.

The time taken for development and the quality of the code (in terms of number of test cases passed and OOD metrics) were measured. Lastly, the effectiveness of the test cases written by TDD developers was analyzed using code coverage analysis. The details of the experiments conducted can be found in Appendix B (for the students) and C (for the professional programmers).

6.2 External Validity

There are four important problems for the external validity (generalization) of the experiment. First, the experiments and the subsequent data analysis were hampered by certain limitations. The experiments with the student programmers were conducted in a time constraint environment. The time required for

development was estimated (to be 75 minutes) based on the time taken to develop a program that had somewhat similar specifications. Subsequently, the professional programmer took an average of 285 minutes to complete this same program. Hence, only one pair completed the assignment, indicating partial or incomplete results in almost all cases. Further, the analysis of student data did not take into account the grade distribution (course grades are considered to be reflection on the quality of student skills).

In all cases, a multivariate analysis needs to be done to explore the causal relationships that might exist between the various other factors that affected the effectiveness of the developers. The experiments and subsequent data analysis of professional developers code explicitly did not take into account the low number of data points and the uneven time comparison (due to failure of non-TDD developers to develop test code). The experiments with professional developers were further confounded by the requirements modification after the first set of experiment. These limitations might be confounding the data recorded and hence the statistical analysis done on recorded data is restricted to reporting the mean and other statistical values. Therefore the results are observations and are not statistically validated.

Second, the results of the study must be considered in the context of developers who use the pair programming practice. The pair programming practice is not required in TDD. However, this variable was used with both the experimental and control group and is uniform throughout the experiments conducted. Two of the three professional developer organizations used pair

programming in their day-to-day development. The other two groups (the students and other professional group) were familiar with pair programming practices. Hence all the experiments were conducted with pair programming practice so that the objective of experiment (which is to evaluate the effectiveness of TDD in the day-to-day development work environment) is not violated.

Third, the application used in the evaluation process was very small; the typical size of the code developed was less than 200 LOC. Hence the specific findings are in the context of very small programs and cannot be generalized to mid-size to large application development. Prior anecdotal reports of the success of TDD are generally in the context of XP, which has been stated to be effective for 10-12 co-located object-oriented programmers [5].

Fourth, the subjects of the experiments had varying experience with TDD (from novice to expert) and software development (undergraduate students to professional developers having up to 20 years of experience). Further, the students and the third set of professional developers had only two to three weeks of experience with TDD before the experiment was conducted. Hence it is conceivable that the test-first approach on these subjects is not stabilized and mid-term benefits from these subjects should be higher than observed in the experiment.

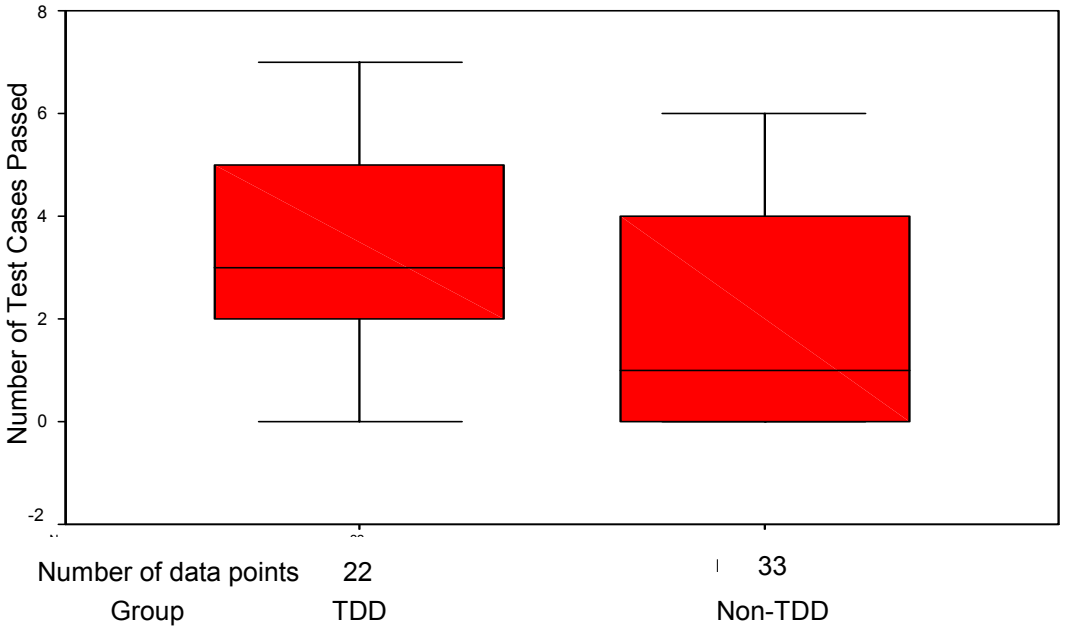
6.3 Quantitative Analysis

The differences of external and internal code quality and of productivity between the TDD and the non-TDD groups were examined quantitatively. The results of this analysis are presented in this section

6.3.1 External code quality

Product quality is a very important metric. We will first discuss the quality of the student programs, followed by that of the professional programmers. The external code quality of both TDD and non-TDD student pairs were evaluated using seven black box test cases, like entering throws for a frame and verifying the score displayed (see Table 3 in Appendix B for the specific test cases). These test cases were used to determine the completeness and the thoroughness of the application. Due to various reasons (like failure to develop any code, see Appendix B for details), of the 30 TDD and 37 non-TDD student data points, only 22 TDD and 33 non-TDD data points were used for evaluation. As indicated in Figure 2, on average, it was observed that the student TDD pairs' code passed approximately 16% more test cases than non-TDD pairs.

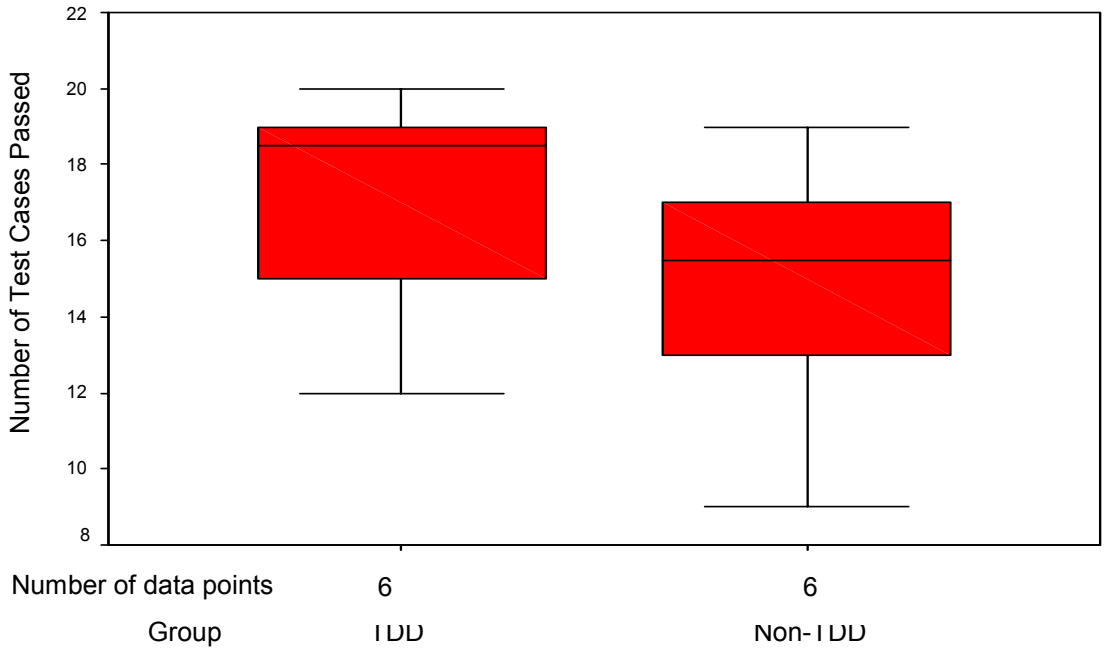
Figure 2: Box plot for Test Cases Passed by Students' Code



The box plot of Figure 2 highlights several important points about the TDD pairs' code quality. First, the median value is clearly much higher than the median for non-TDD. Also, the 25th percentile (lower edge of TDD's box) is above non-TDD's median.

The professional developers' code was evaluated using a set of twenty black box test cases (refer to Appendix C for details). As shown in Figure 3, on an average, professional TDD pairs' code passed approximately 18% more test cases than non-TDD pairs. As in the case of students' code, the external code quality of professional TDD pairs was much higher than that of the non-TDD pairs. The box plot (Figure 3) demonstrates this graphically.

Figure 3: Box plot for Test Cases Passed by Professional Developers' Code



The median value is again much higher than that for non-TDD. Also, the total range of values for TDD pairs is significantly smaller, indicating greater consistency in high quality.

A hypothesis of this research was that the TDD approach will yield code with superior external code quality as confirmed by the success of more number of functional test cases by code developed through TDD method when compared with code developed with a more traditional waterfall-like model practice. Based on the data analysis conducted, the experimental findings are supportive that the TDD approach yields code with superior external code quality. However, the study limitations must be considered.

6.3.2 Internal code quality

The internal quality of the code was assessed using OOM. Again, the student results are discussed prior to the results of the professional programmers. For this analysis, we eliminated the student pairs whose code failed to pass any test cases. As a result, the data points were reduced to 20 TDD and 19 non-TDD pairs. Table 1 summarizes the results of metrics evaluation of student code. Table 5 in Appendix B provides detailed results of the metrics evaluation, which includes Standard Deviation (SD), median and Inter-Quartile Ranges (IQR) of the measurements apart from the mean.

Table 1: Summary of metrics evaluation of student code

Metrics	Desirable Value	Superior Performance By (Pair type)	Mean Value of TDD / non TDD pair
AHF	High	TDD	48.20 / 34
CC	Low	TDD	11.45 / 13.11
LOC	Low	TDD	70.20 / 94.74
MHF	High	Non-TDD	2.55 / 10.53
NOC	Low	TDD	1.65 / 1.68
NOO	Low	Non-TDD	4.90 / 4.89
RFC	Low	TDD	10.20 / 11.42
WMPC 1	High	Non-TDD	11.45 / 13.11
WMPC 2	High	TDD	9.10 / 8.42

As evident from the table, of the nine metrics used, the TDD pairs fared better than the non-TDD pairs in six metrics. The non-TDD pares fared better in the remaining three. The TDD pairs' code had fewer lines of code, operations and a lower value for RFC.

Table 2 summarizes the metrics evaluation of professional developers' code. In addition to the metrics used in student code evaluation, the professional developers' code was evaluated using additional metrics such as LOCOM to measure the cohesion and coupling measures in the code. See Table 9 in Appendix C for detailed results of metrics evaluation (which includes SD, median and IQR of the measurements apart from mean). The results indicate that, of the 14 metrics used, the TDD groups performed better than non-TDD in five and non-TDD surpassed TDD in eight metrics. One metric, MHF, was inconclusive.

In addition to metrics evaluation, code inspection was performed on the code developed by professional developers. It was observed that three out of the six non-TDD code data points had orphan variables or methods (variables or methods that exist in the program but are not called or used in the program).

Table 2: Summary of metrics evaluation of professional developers

Metrics	Desirable Value	Superior Performance By (Pair type)	Mean Value of TDD / non-TDD pair
AHF	High	TDD	69.83/52.50
CBO	Low	Non-TDD	6.17/5.50
CC	Low	Non-TDD	25/24.33
CF	Low	Non-TDD	172/160.17
LOC	Low	TDD	226/235.17
LOCOM 1	High	Non-TDD	11.83/12.17
LOCOM2	High	Non-TDD	89.33/98.67
LOCOM3	High	Non-TDD	59.17/65.83
MHF	High	Inconclusive	2/2
NOC	Low	TDD	3.33/4.67
NOO	Low	Non-TDD	10/9.67
RFC	Low	Non-TDD	32.83/20.5
WMPC1	High	TDD	25/24.33
WMPC2	High	TDD	15.5/14.33

Both student and professional TDD pairs consistently fared well in metrics like LOC, AHF and NOC. Non-TDD student and professional pairs consistently fared well only in one metric NOO. It might be subjectively interpreted that TDD leads to lower lines of code. Programs that have fewer lines of code while performing the same functionality are considered to have better design characteristics and require lower maintenance costs [52]. Due to reasons, such as low number of data points and small LOC (details of which are listed in section 6.2) the conclusions on the internal code quality do not imply much. However, the results are included for their merits.

It was hypothesized that the TDD approach will yield code with superior internal code quality, as measured by established OOD metrics when compared with code developed with a more traditional waterfall-like model practice. However the experimental results are inconclusive.

6.3.3 Productivity

People can be skeptical about the additional time needed to write and update test cases. On an average, the TDD pairs took approximately 16% more time to develop the application than non-TDD pairs.

Figure 4: Box plot of Time Taken by Professional Developers

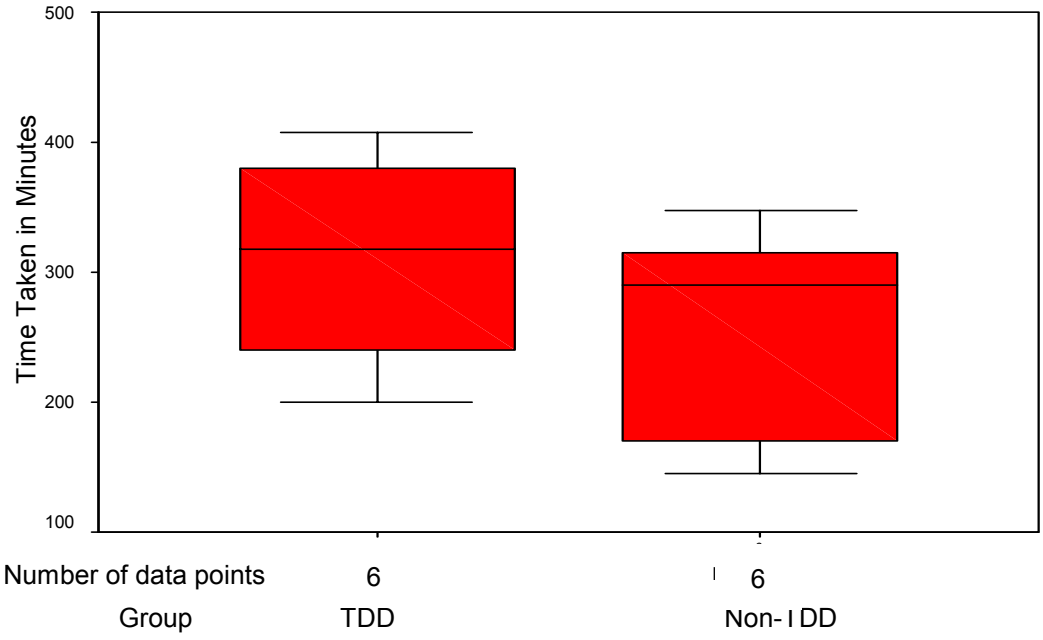


Figure 4 shows the box plot for the time professional developers took to develop the application. The medians of the two groups are nearly equal. However, the upper range value is higher for the TDD developers. The non-TDD pairs were asked to write test cases after they developed code (in a traditional

code-then-test fashion.) However, only one group wrote any test cases. The failure of non-TDD developers to write any test cases resulted in an uneven comparison of the time taken. The failure of the non-TDD pairs to write test cases, even though they were instructed to do so, could be indicative of actual industry practices. While we would like to believe that most developers practice test-then-code, it is likely that they practice test-never. The extra time taken by TDD could be attributed to the time needed to develop test cases.

Although TDD developers took more time in comparison to non-TDD developers (who did not write any test cases), there are many benefits resulting from automated test cases. First, the TDD pairs produced test assets along with the implementation code. These test assets are very valuable in the product life. Second, the code developed is testable. Third, the code that enters subsequent testing phases and that is delivered to the customer is of higher quality. This higher quality reduces testing and field support costs. Finally, the over all software life cycle time might be less in subsequent iterations because changes can be made more easily. It might be professed that TDD, just like OOP, requires more initial development time while reduces the overall life cycle time.

We hypothesized that programmers who practice TDD approach will develop code quicker, as measured by the time (hours) to complete a program, than developers who develop code with a more traditional waterfall-like model practice. In contrary to this hypothesis, the experiment results showed the TDD professional developers took approximately 16% more time than non-TDD

professional developers. However, the validity of results is severely limited, as discussed in Section 6.2.

6.4 Qualitative Analysis

It is fruitful to substantiate quantitative findings with qualitative feedback from the software developers in the experiment. Qualitative analysis of the effectiveness of TDD was done by reviewing the comments of students on the approach and by conducting a survey (see appendix E) with the professional developers.

Summarizing the 63 comments from the students, a majority of the students, 61.9% (39 students) liked the TDD approach, while 14 students (22.2%) had strong reservations. A small percentage of students, 15.9% (10 students) were indifferent to the approach. The most repeated positive comment was that the approach enabled students to understand the requirements properly. Other comments stated that the approach resulted in better structured code, and that it focuses on getting the code work properly the first time, rather than first developing the code and then fixing it.

The most repeated negative comment was that TDD distracted developers from doing any upfront design, which indicated the difficulty in getting into TDD mindset. Some expressed concern that quality of the code depends on the quality of test cases. The concern is logical; in traditional approaches the quality of code depends more on the design, and testing is done to validate the design. Among the pairs who were indifferent, some thought that the approach was time consuming but produced high quality code. The difference between the number

of positive comments vs. negative comments was statistical significant ($p < 0.04$) when evaluated using the normal approximation method.

The survey conducted with the professional developers indicated a strong preference for TDD. Developers responded positively to almost all questions. The survey had three sections, dealing with questions on (1) programmer productivity (2) effectiveness of the approach and (3) difficulty in adopting the approach. On questions on programmer productivity, an overwhelming majority of the developers believed that TDD approach facilitates better requirements understanding (87.5%) and reduces debugging effort (95.8%). However, only half of the developers agreed that TDD needs less code development time. (Based on our quantitative analysis, these developers were justified in this opinion.) Taking the average of all positive comments, about 77.8% of developers thought that TDD improves overall productivity of the programmer. For all questions relating to effectiveness, a majority of the developers (percentage not dipping below 70.8% for any question) responded positively, Ninety-two percent (92%) believed that TDD yields higher quality code. Eighty percent (80%) of developers thought that TDD enhances code quality and 70.8% thought the approach was visibly (overall) effective. However, the responses of developers on questions related to difficulties in adopting the approach indicate that there are some concerns about the approach. Fifty-six percent (56%) of the professional developers thought that getting into the TDD mindset was difficult. A minority (23%) indicated that the lack of upfront design phase in TDD was a hindrance.

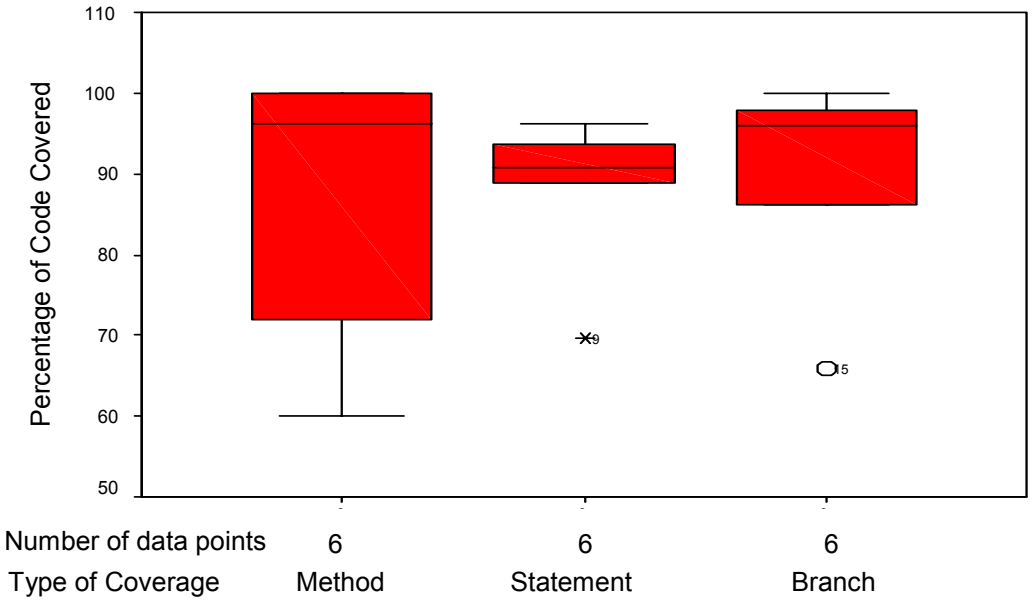
Hence taking average of the responses, 40% of the developers thought that the approach faces difficulty in adoptive-ness.

Based on survey and student comments, it can be concluded that developers feel that TDD is effective in terms of code quality and improves programmers' productivity. However, getting into TDD mindset is difficult for some developers. Lastly, some programmers expressed concerns about the increase in development time needed to write the test cases. More detail of this survey can be found in Appendix C.

6.5 Code coverage

One of the concerns about the TDD approach is the effectiveness of the test cases written by the TDD developers. If the TDD approach is so highly reliant on a good set of test cases to drive design, for refactoring, and for regression testing, a poor set of test cases could deem the practice ineffective. Essentially in TDD, the quality of the tests determines the quality of the code. In order to evaluate the quality of the test cases, the professional TDD developers' test cases were analyzed for code coverage.

Figure 5: Box Plot of Code Coverage by Professional Developers

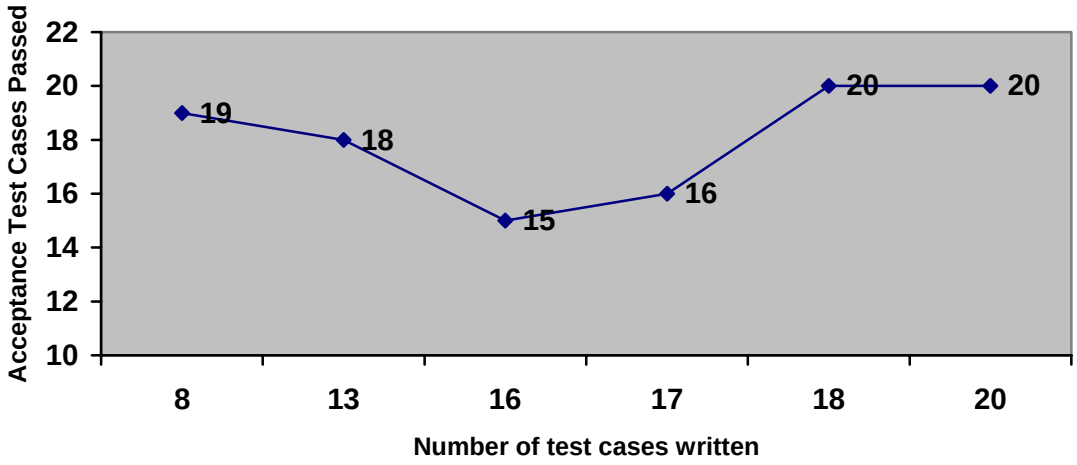


As stated before the industry practice for coverage is in the range 80% to 90%, although ideally the coverage should be 100%. As evident from the above figure, TDD developers did well in all the three types of code coverage. On an average, the TDD developers' test cases achieved 98% method, 92% statement and 97% branch coverage. It must be noted that the testing tool used, JUnit, cannot test the main method (of Java programs), and hence the main method was excluded from code coverage analysis. Even the 25th percentile (noted by the red box) values (lowest value being 75%) are comparably to the lower end of the industry standard (80%), which suggests high quality in the test cases written. In fact, it must be noted that these results must be regarded with extreme caution since they are based on a very simple and small programming assignment.

6.6 Relationship between quality and test cases developed

Just as code coverage indicates the quality of the test cases developed, another important measure might be the relationship between quality and number of test cases developed by the TDD pairs. The figure below indicates the plot between number of black box test cases (user acceptance test cases) passed (which indicates the quality of the code) to the number of test cases the TDD developers wrote during coding phase.

Figure 6: Quality vs. Test Cases Written



The mapping fails to indicate any significant relationship between quality and number of test cases written. However, one interesting aspect to note is that more experienced TDD developers (belonging to RoleModel Software Company) wrote more number of test cases (18 and 20 respectively) in comparison to the intermediate to beginner TDD developers (belonging to John Deere and Ericsson).

7. CONCLUSION

A series of experiments, in both an academic and an industrial environment, were conducted to examine the TDD approach. Specifically the following hypotheses were tested and corresponding conclusions were obtained, subject to the limitations of the study:

- For very small programs, the TDD approach appears to yield code with superior external code quality, as measured by conformance to a set of black box test cases, when compared with code developed with a more traditional waterfall-like model practice.
- For very small programs, the TDD approach does not appear to yield code with superior internal code quality, as measured by the established Object-Oriented Design (OOD) metrics, when compared with code developed with a more traditional waterfall-like practice.
- The experiment results showed that TDD developers took more time (16%) than non-TDD developers. However, the variance in the performance of the teams was so large that this value is only directional and cannot be used to make any conclusions one way or other. Multivariate statistical analysis need be performed to assess the relationship between time and quality to further assess the impact TDD has on programmer productivity.

A survey was conducted to collect developers' opinion about the approach. A majority, 60% of students asserted that TDD was an effective approach. On an average, 80% of the professional developers held that TDD

was an effective approach, and 78% believed that the approach improves the overall productivity of the programmer. The results of the survey are statistically significant.

Qualitative results of this research seem to indicate that TDD approach facilitates simpler design and lack of upfront design is not a hindrance. However, for some developers getting into the TDD mindset is difficult.

However, it must be emphasized that more general validity of the above results must be regarded with extreme caution since they are based on a very simple and small programming assignment, and on a set of very few and limited statistical analysis. However, results appear to be promising and are a strong indication that TDD warrants further research.

7.1 Future Work

XP and TDD software development methodology demand a radical shift from traditional methods. Considerable research needs to be done on how to make this transition a smooth one. Since software engineering considers human factors and humans are resistant to change, research along those lines can help more people to adapt these approaches.

TDD can be adapted to traditional software development methodologies as well. The XP methodology encourages creating design from code, i.e. design evolves as coding progresses. Incorporating TDD into a traditional methodology can aid in fine tuning the initial design. Many software development processes, particularly those involving life- and safety-critical applications, necessitate architectural, high and low level designs. The level of upfront design done with

TDD can be varied from as little as identifying one single object (logical design) to as large as doing a high and low level design. The approach is highly flexible.

In this research work, the statistical analysis was limited to reporting the mean and SD values. More advanced statistical analysis such as multivariate statistical analysis needs to be done on the quantitative data obtained from this study. Further, the experiment conducted in this work, needs to be redone in a more controlled setting to reduce the limitations faced. Reruns of the experiments are easier as the design of those experiments can be adapted from this work.

For this research work, all the experiments were conducted in a pair programming environment. Pair programming has been shown to yield higher code quality [14] in the absence of TDD. An interesting experiment would be to analyze the effectiveness of TDD in a setting where pair programming is not performed. It could be argued that in such environments TDD will be much more effective, because pair programming tends to provide feedback to the person who is coding, resulting in lower defects. When such a feedback mechanism is removed, the chances of defects in non-TDD developers' code might increase.

For this research work, the quality of the code was determined using a series of acceptance (black box) test cases. However the user would be more concerned about the MTBF than the number of defects in the code. Moreover many researchers argue that MTBF is a better measure of than defects/LOC [12]. Hence the code developed should be further analyzed to determine the MTBF.

REFERENCE:

1. Feathers, M., *Test First Design- Growing an application one test at a time*, in *XP Magazine*. Sept 2001.
2. Beck, K., *Test Driven Development: By Example*. 2002: Addison Wesley.
3. Beck, K., *Extreme Programming Explained: Embrace Change.* Reading, Massachusetts. 2000: Addison-Wesley.
4. Auer, K. and R. Miller, *XP Applied*. Reading, Massachusetts:. 2001: Addison-Wesley.
5. Jeffries, R., A. Anderson, and C. Hendrickson, *Extreme Programming Installed*. Upper Saddle River, NJ:. 2001: Addison Wesley.
6. Jeffries, R.E. *Extreme Testing*. in *Software Testing and Quality Engineering*. 1999.
7. Chaplin, D., *Test First Programming*. TechZone, 2001.
8. Beck, K., *Aim, Fire*, in *IEEE Software*. Oct 2001. p. 87- 89.
9. *A survey of System Development Process Models CTG.MFA - 003*. 1998, Center for Technology in Government, University at Albany/SUNY.
10. Sorensen, R., *A Comparison of Software Development Methodologies*. 1994, Software Technology Support Center.
11. Goldsmith, R.F. and D. Graham, *The Forgotten Phase*, in *Software Development*. July 2002.
12. Pressman, S.R., *Software Engineering A Practitioner's Approach*. 2001, 5th Edition, Mc Graw Hill.
13. *IEEE Std 1012-1986 IEEE Standard for Software Verification and Validation Plans*. 1986, IEEE, New york.
14. Williams, L.A., *The Collaborative Software Process*. University of Utah. 2000, Salt Lake City, UT: Department of Computer Science.
15. Martin, C.R., *Advanced Principles, Patterns and Process of Software Development*. 2001, in press: Prentice Hall.
16. Ziv, H. and J.D. Richardson. *The uncertainty principle in software engineering*. in *ICSE*. 1997.
17. Humphrey, W.S., *A Discipline for Software Engineering*. Reading, Massachusetts. 1995: Addison Wesley Longman, Inc.
18. MacCormack, A. *How Internet Companies Build Software*. 2001: MIT Sloan Management Review.
19. Cockburn, A., *Agile Software Development*. 2001: Addison Wesley Longman.
20. Fowler, M., *The new Methodology*, Thoughworks.
21. Anderson, A., et al., *Chrysler goes to Extremes*, in *Distributed Computing*. Oct 1998. p. 24-28.
22. *Wiki, Extreme Programming Roadmap*. 1999, Portland Pattern Repository.
23. Pauli, K., *Pattern your way to automated regression testing*, in *Java world*. Sept 2001.
24. Fowler, M., *Is design dead?* Extreme programming Examined. 2001: Addison Wesley.
25. GassMann, P., *Unit testing in a Java Project*, in *Extreme programming Examined*. 2001, Addison Wesley.

26. Tomayko, J.E., *Managing Evolving Requirements Using extreme Programming*. 2001, Institute of software research, International, CMU.
27. *Developing Object-Oriented Software - An Experience- Based Approach*. 1997, IBM Object-Oriented Technology Center, Prentice Hall PTR. p. 72.
28. Rutherford, K., *Retrofitting Unit tests with JUnit*, in *Extreme programming Examined*. 2001, Addison Wesley.
29. Gamma, E. and K. Beck, *Test Infected: programmers love writing test*. July 1998, The Java Report, 3(7):. p. 37-50.
30. Royce, W.W. *Managing the Development of Large Software Systems:Concepts and Techniques*. in *WESCON*. August 1970.
31. Hanna, M., *Farewell to Waterfalls*, in *Software magazine*. May 1995. p. 38-46.
32. Bradac, M., D. Perry, and L. Votta. *Prototyping a Process Monitoring Experiment*. in *IEEE Transcations on Software Engineering*. October 1994.
33. Beizer, B., *Software Testing Techniques*. ITP, 2nd Edition. 1990.
34. Gelperin, D. and W. Hetzel. *Software Quality Engineering*. in *Fourth International Conference on Software Testing, Washington D.C.* June 1987.
35. Langr, J., *Evolution of Test and Code via Test First Design*. March 2001: Object Mentor.
36. Dustin, E., J. Rashka, and J. Paul, *Automated Software Testing*. Reading, Massachusetts. 1999: Addison Wesley.
37. Ducasse, M. and A.M. Emde, *A Review of Automated Debugging Systems: Knowledge, Strategies, and Techniques*. 1988, IEEE Computer Press.
38. Paulk, M.C., *Extreme Programming from a CMM Perspective*, in *IEEE Software*. November/December 2001. p. 19-26.
39. Fowler, M., et al., *Refactoring: Improving the Design of Existing Code*. Reading, Massachusetts. 1999: Addison Wesley.
40. Mbeki, B.N., *Extreme Programming: My Experience*, CMU.
41. ElEmam, K., N. Goel, and S. Rai, *A Validation of Object-Oriented Metrics*. Oct 1999, National Research Council Canada, NRC 43607.
42. Glasberg, D., et al., *Validating Object-Oriented Design Metrics on a commercial Java Application*. Sept 2000, National Research Council 44146.
43. Basili, V.R. and L.C.B.L. Melo. *A Validation of Object Orient Design Metrics as Quality Indicators*. in *IEEE Transactions on Software Engineering*. 1996.
44. Briand, L., K.E. Emam, and S. Morasca, *Theoretical and empirical Validation of Software Metrics*. 1995.
45. Schneidewind, N.F. *Methodology for Validating software metrics*. in *IEEE Transcations on Software Engineering*. 1992.
46. Chidamber, S.R. and C.F. Kemerer. *A Metrics Suite for Object Oriented Design*. in *IEEE Transactions on Software Engineering*. 1994.
47. Abreu, F.B. *The MOOD Metrics Set*. in *ECOOP '95 Workshop on Metrics.* 1995.
48. Churcher, N.I. and M.J. Shepperd. *Comments on 'A Metrics Suite for Object-Oriented Design'*. in *IEEE Transactions on Software Engineering*. 1995.
49. ElEmam, K., *A Methodology for Validating Software Product Metrics*. June 2000, National Research Council of Canada, Ottawa, Ontario, Canada NCR/ERC-1076.
50. ElEmam, K., S. Benlarbi, and N. Goel, *The Confounding Effect of Class size on the validity of Object-Oriented Metrics*. 1999.

51. Lorenz, M. and J. Kidd, *Object - Oriented Software Metrics A practical Guide*. 1994: PTR Prentice Hall.
52. Boehm, W.B., *Software Engineering Economics*. 1981: Prentice-Hall Inc.
53. Boehm, B.W., *Improving Software Productivity*, in *IEEE Computer*. Sept 1987. p. 43-57.
54. Kit, E., *Software Testing in the real world improving the process*. 1995: Addison Wesley.
55. Berard, V.E., *Understanding the Recursive/Parallel Life-cycle*, The object Agency.
56. Gelperin, D. and B. Hetzel, *The growth of software testing.* Communications ACM, June 1988. **31**: p. 687 - 695.
57. *ANSI/IEEE STD 1008 -1987 Standard for software unit testing*. 1986, IEEE, New york.
58. Hetzel, W., *The complete guide to software testing*. 2nd edition. 1988: QED Information Sciences Wellesley Mass.
59. *A guide to testing in a complex system environment*. 1974, Report GH20-1628.
60. Jia, X., *Object Oriented Software Development Using Java*. 2000: Addison Wesley.
61. *An introduction to Software Testing*. 1996, IPL Information Processing Ltd.
62. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries (610-1990)*. 1990: New York, NY.
63. Voas, J. and K. Miller. *Inspecting and ASSERTing Intelligently*. in *Software Testability: Investing in Testing Proceedings of EuroStar'96*. December 1996. Amsterdam.
64. *IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology*. 1990, IEEE, New york.
65. Voas, J. and K. Miller. *Software Testability: The new verification*. in *Software Testability Measurement for Assertion Injection and Fault Localization Proceedings of 2nd Int'l. Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*. May 1995. St. Malo, France.
66. Keene, S. and C. Lane, *Reliability growth of fielded software.* Proceedings of Reliability and Maintainability Symposium, 1993: p. 360 -365.
67. Littlewood, B., *How to measure software reliability, and how not to.* Proceedings of the 3rd international conference on Software engineering, 1978: p. 37 - 45.
68. Cornett, S., *Code Coverage Analysis*. 2002, Bullseye Testing Technology.
69. Zhu, H. and P. Hall, *Software Unit Test Coverage and Adequacy.* ACM Computing surveys, 29(4), December 1997: p. 346 - 427.
70. Fowler, M., *UML Distilled*. Reading, Massachusetts:. 2000: Addison Wesley.
71. Roberts, F., *Encyclopedia of Mathematics and its Applications*. 1979: Addison Wesley Publishing Company.
72. Coad, P. and E. Yourdon, *Object Oriented Analysis*. Jan 1991: 2nd Edition, Yourdon Press Computing Series.
73. Gamma, E., et al., *Design Patterns Elements of Reusable Object-Oriented Software*. 1994: Addison Wesley.
74. Vienneau, L.R. and R. Senn, *A State of the Art Report: Software Design Methods*. March 1995, Air Force Research Laboratory - Information Directorate.

75. Knoernschild, K., *Pattern Foundations: The open-closed Principle*, in *Java Developers Journal*. March 2002.
76. Bellin, D. and S.S. Simone, *The CRC Card Book*. Reading, Massachusetts. 1997: Addison-Wesley.
77. Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Reading, Massachusetts:. 1999: Addison-Wesley.
78. Curtis, B. *Three Problems Overcome with Behavioral Models of the Software Development Process*. in *11th International Conference on Software Engineering*. 1989.
79. Kerievsky, J., *Stop Over-Engineering!*, in *Software Development magazine*. April 2002.
80. Budinky, F.J., et al., *Automatic code generation from design patterns.* IBM Systems Journal, 1996. **35**(2).
81. Kerievsky, J. *Patterns and XP*. in *XP 2000*. June 2000.
82. Juristo, N. and A.M. Moreno, *Basics of Software Engineering Experimentation*. 2001: Kluwer Academic Publishers.
83. Twaite, J.A. and J.A. Monroe, *Introductory Statistics*. 1946: Scott, Foresman and Company.
84. Wonnacott, T.H. and R.J. Wonnacott, *Introductory statistics*. 5th ed. 1990: Wiley.

APPENDIX A: Research Approach

The quantification and analysis of the effectiveness of TDD approach was conducted through a series of experiments. In order to increase the validity of the experiments, two different categories of developers were chosen, student and professional. The details of these experiments are given in appendix B and appendix C.

The effectiveness of TDD was assessed using four types of evaluation: (1) analysis of survey/comments of programmers on the TDD approach; (2) the time taken to develop an application; (3) black box test cases to evaluate external quality; and (4) OOM to assess internal quality. Additionally the quality of the test cases written by the TDD pairs was measured using code coverage analysis. The code coverage analysis was conducted using the JCover[6] tool, a code coverage analyzer for Java programs.

The metrics (for Java code) were computed using the TogetherSoft Control Center[7] tool, a comprehensive development environment that includes a static software metric analyzer. The metric data hence obtained was further analyzed using Statistical Package for Social Sciences (SPSS) tool, to obtain the mean, median, SD and IQR values. The survey results were also analyzed using SPSS tool for statistical significance using normal approximation and Spearman's Rho test.

[6] http://www.mmsindia.com/JCover.html
[7] http://togethersoft.com

APPENDIX B: Experiments conducted with students

A formal experiment was conducted at NCSU with 138 undergraduate (junior/senior) computer science students who took the software engineering course offered during fall 2001. During the first three weeks of the class, the students were taught the TDD approach and the JUnit tool. They completed two homework assignments utilizing the technique and the tool. The Java expertise of the students was estimated at the beginner level, with some students possessing intermediate level of knowledge because of experience in C++ in prior classes.

The experiment was conducted after receiving the requisite permission from the Institutional Review Board (IRB) at NCSU. IRB permission, required to conduct experiments with human beings, is to ensure that students' rights are not violated. The IRB deemed the experiment exempt from surveillance because the experiment was conducted in an established education setting (school labs) and was evaluated using normal education practices.

A regular lab session of the CSC326 course was used to conduct the experiment. There were seven sessions of the lab, each of 75 minutes duration. The objective of the experiment was to compare the effectiveness of the TDD approach. A program somewhat similar to the one chosen for the experiment was previously completed in approximately 75 minutes. Hence the estimated time for development was 75 minutes. However, the students did not complete the assignment in this amount of time. As a result, the experiment demonstrated

the capabilities of TDD in a time-constrained development environment. The 138 students were randomly assigned to 30 TDD and 37 non-TDD groups of two.

Of the 30 TDD pairs, eight were eliminated from test case evaluation due to various reasons:

- Three groups were eliminated since they had only one member (not consistent with pair programming practice).
- The code produced by five groups either did not compile or run. These groups were only learning to code in Java. Hence, those data points were discarded.

Of the 37 non-TDD pairs, four were eliminated from test case evaluation due to various reasons:

- Two groups were eliminated as the groups had only one member.
- Two more were eliminated from test case evaluation, as their code either will not compile or run.

Seven black box test cases (see Table 3 for details on these test cases) were used to determine extent to which the application was complete and of high quality. Test cases 1 to 3 tests for basic input, multiple frame handling and accurate calculations of scores. The capability of the application to handle strikes and spares were checked in the fourth and fifth test cases, while the sixth and seventh test cases checks for boundary conditions. We considered that passing one to three test cases meant completion of basic functionalities (such accept pins knocked down, advance frame), while passing four to seven tests meant

moderate to complete incorporation of all the functionalities (such as strike, spare, boundary case).

Table 3: Test case used for student code evaluation

Test Case Number	Input Entered	Excepted Score	Remarks
1	5,4	9 for frame 1 total score 9	Used to check whether application can accept any input.
2	5, 4, 7, 2	9 for frame 1 18 for frame 2 total score 18	Used to check whether application can handle multiple frames.
3	3, 7, 3	13 for frame 1	Used to check whether application can calculate score for first frame properly
4	3, 7, 3, 2	13 for frame 1 18 for frame 2 total score 18	Used to check whether application can handle spare special case properly
5	10,3,6	19 for frame 1 total score 28	Used to check whether application can handle strike special case properly
6	10 entered 12 times	Total score is 300	Used to check whether application can handle perfect game score.
7	0 entered 18 times, followed by 2, 8, 10	Total score is 20	Used to check the boundary condition in which all inputs, except for last frame is zero and last frame scores a spare and resultant bonus frame is a strike

An application that passes up to 3 test cases can be thought of as being moderately developed. An application that passes up to six test cases can be thought of as being developed while an application that passes the entire list of test cases is said to have fulfilled all the requirements of the application.

Table 4: Number of Test cases passed

No of test cases passed	TDD pair	Non-TDD pair
0	9%	43%
1 – 3	41%	30%
4 – 7	50%	27%

As noted in Table 4, 91% of the TDD pairs were able to implement at least some functionality, while only 57% of non-TDD pairs managed to achieve the same.

For metrics evaluation, the two TDD pairs and the fourteen non-TDD pairs, whose code didn't pass any test cases, were eliminated. It was assumed that code that does not pass any test case will not have any worthwhile implementation for OOM analysis. Hence, only 20 TDD and 19 non-TDD data points were used for metrics evaluation. Table 5 gives the detailed results of the metrics evaluation.

Table 5: Detailed metrics evaluation of student code

Metrics	Pair type	Mean	Median	SD	IQR
AHF	TDD	48.2	46.5	44.3	100
	Non-TDD	34.2	0	46.9	100
CC	TDD	11.5	11	5.8	9.3
	Non-TDD	13.1	12	5.7	8
LOC	TDD	70.2	73	30.6	48.3
	Non-TDD	94.7	84	45.3	54
MHF	TDD	2.6	0	6.5	0
	Non-TDD	10.5	0	21.6	12
NOC	TDD	1.7	2	0.6	1
	Non-TDD	1.7	1	0.8	1

Table 5 Continued.

NOO	TDD	4.9	4	2.8	3.8
	Non-TDD	4.9	4	2.9	4
RFC	TDD	10.2	8.5	6.9	7
	Non-TDD	11.4	10	3.6	4
WMPC1	TDD	11.5	11	5.8	9.3
	Non-TDD	13.1	12	5.7	8
WMPC2	TDD	9.1	7	4.8	6
	Non-TDD	8.4	8	4.9	5

The detailed results include SD, mean, median and IQR. Since the line of code written by students was very small (TDD developers mean value is 70 and that of non-TDD is 94), unfortunately the metrics evaluation did not yield any significant results. However, some noticeable difference exists in the LOC and AHF metrics, where the TDD developers had higher mean value than non-TDD developers. The non-TDD developers had a noticeably higher MHF value in comparison to the TDD developers.

Qualitative analysis

For conducting qualitative analysis, after the development, the TDD pairs were asked to write a short note (5 lines) on their impression of the approach. Of the 76 students in TDD pairs, 55 gave comments on the approach. The remaining 21 students did not comment on the approach. Eight students from non-TDD groups commented (voluntarily) on the approach and were included (thus making a total of 63 comments). These eight students based their comments on the two homework assignments they had completed. Of the 63

comments on the approach, a majority, 62% (39 students) liked the concept. Twenty-two percent (22%) or 14 students had strong reservations with the approach. A small percentage of students, 16% (10 students) were indifferent to the approach.

The most repeated positive comment about TDD approach was that the approach enabled the students to understand requirements and details of design properly even without doing any formal design. An example comment read, “Thinking of test made me get to the details of design that I usually write down and hand test, now I have it hard tested also.” In other positive comments, the students state that the TDD approach guided through the coding process resulting in better-structured code, since they had to sit down and think on how the code should function. Further, two pairs commented that the approach forced them to build the application in manageable discrete chunks of code that was thoroughly tested. As stated clearly by the students, the focus of TDD approach is to get the code working properly the first time rather than developing the code and later fixing it to work properly.

The most repeated negative comment about TDD approach was that the approach distracted the developer from doing any upfront design. “It … distracted us from the design of the code each time we had to stop and write new test cases”. The reason for such comments could be attributed to the need for getting into a TDD mindset. The developers who use TDD have to think about the tests first, rather than how to design the whole system. A pair expressed the concern that quality of the code depends much on the test cases and there will not be

code for overlooked test cases. However such concerns can be alleviated because the final product should go through user acceptance tests in which the product is validated to meet all the requirements. The quality of code developed through traditional methods depends more on the design than on test cases, hence an overlooked test case will result in an untested code leading to potential defects. In case of TDD such possibility does not arise as the code present is tested by test cases.

Some students were not sure about the effectiveness of the approach as evident in this statement. “It seemed annoying and time consuming but methods stayed small and manageable”. Among the pairs who were indifferent to the approach, some thought that the approach was time consuming. However, the number of test cases passed clearly shows that TDD pairs were able to develop more of the functionality.

APPENDIX C: Experiments conducted with professionals

Three sets of experiments were conducted on small (eight member) group of developers from three companies (John Deere, Ericsson, and RoleModel Software). The developers had varying (novice to expert level) levels of experience with the TDD approach.

The first group of professional developers who participated in the experiment consisted of eight developers from the John Deere Webworks team based in Cary, NC. Of the eight, two did not have any prior experience with TDD approach. The TDD /non-TDD pairs were chosen randomly; fortunately, the two developers with no previous experience were evenly distributed among the TDD and non-TDD groups. The remaining six developers were experienced in the TDD approach in varying levels, from beginner to intermediate level of experience. Before the commencement of the experiment, a short survey was conducted to collect their impression about TDD approach. This survey can be found in Appendix E.

The second group of professional programmers consisted of eight developers from RoleModel Software Company, an XP development company based in Holly Springs, NC. The eight developers had considerable prior experience (intermediate to expert level) with TDD approach. The developers, randomly assigned to two groups, TDD and non-TDD pairs, were asked to develop the same bowling game scoring application. The developers were specifically asked to handle any invalid inputs or error conditions gracefully. This time, the non-TDD group was asked to write test cases after development

(including testing) was over, to ensure an even work load between TDD and non-TDD developers. Although the non-TDD pairs were asked to develop test cases after development, one group did not write any test cases, while the other pairs' test cases failed to execute properly. Before the experiment, a modified survey (with neutral choice removed) was conducted to collect the impression about TDD approach.

The third set of experiment (similar to the one conducted at RoleModel software company) was conducted at Ericsson at RTP, NC with eight developers. The developers had no experience with TDD. Therefore, three weeks prior to the conduct of the experiment, the researcher conduced a training session on TDD to the developers. The developers then practiced the approach in their daily work environment. Hence, these developers had beginner experience with the TDD technique. The developers' prior experience was mostly in C++ with some having intermediate to advanced knowledge in Java. The developers were put into the TDD and non-TDD groups based on their knowledge level in Java. Although the non-TDD pairs were asked to develop test cases after development, only one group developed any test cases.

In all, there were 12 data points from professional developers. Table 6 provides the mean of the total time taken by the developers of each company. The time taken by non-TDD pairs to design, code and then test the code was recorded, while in the case of TDD, since all these phases are tightly coupled to form a single phase only the total time for development was noted. The

developers of RoleModel and Ericsson took more time as they were asked to handle all the possible error conditions.

Table 6: Mean Time Taken by professional programmers

Group type	Design Time	Implementation Time	Testing Time	Total time Non-TDD	Total time TDD
Non-TDD Group 1	30 mins	88 mins	40 mins	158 mins	-
Non-TDD Group 2	14 mins	175 mins	140 mins	329 mins	-
Non-TDD Group 3	10 mins	185 mins	98 mins	292 mins	-
TDD Group 1	-	-	-	-	220 mins
TDD Group 2	-	-	-	-	325 mins
TDD Group 3	-	-	-	-	387 mins

The TDD developers took more time to develop the application, contrary to the research hypothesis. The possible reasons for TDD developers to take more time could include the fact that TDD developers developed test case along with code while non-TDD developers developed only the code. Further, the non-TDD developers failed to develop code according to the requirement specification resulting in faster implementation. The non-TDD developers (excluding John Deere's) were asked to write test cases after code development, however they failed to write any test cases. Only one pair wrote any worthwhile test cases. These test cases could not be evaluated due to lack of sufficient other data points.

We increased the thoroughness of the black box test cases that used to evaluate student programs. The code of the professional programmers was

evaluated using 20 black box test cases. The test cases validated the robustness of the code, such as error handling capabilities and degree to which requirement specifications were incorporated. Specifically, the tests were designed to verify that the application can handle invalid inputs, asks for appropriate number of inputs and handle all the special cases (that occur due to strike and spare conditions) properly. Table 7 summarizes the test cases used for evaluation.

Table 7: Test cases used for professional code evaluation

Test Case Number	Input Entered	Excepted Score	Remarks
1	10	Not available	Used to check whether application displays incomplete scores properly.
2	10, 2	A proper menu	Used to verify that the application has the proper menu.
3	3, 7, 3, 2	13 for frame 1 18 for frame 2 total score 18	Used to verify that the application shows score per frame.
4	-1	Invalid input message	Used to check whether application can handle invalid inputs.
5	All inputs 0	Total score is 0	Used to check whether application asks for appropriate number of inputs.
6	10, rest all 0	Total score is 10	Used to check whether application calculates score properly.
7	0 entered 18 times, 10 ,0 ,0	Total score is 10	Used to check whether application calculates score properly.
8	0 entered 18 times, 8 ,2 ,0	Total score is 10	Used to check whether application can handles spare calculation properly.
9	11	Invalid input message	Used to check whether application can handle invalid inputs
10	6, 6	Invalid input message	Used to check whether application can handle invalid inputs

Table 7 continued.

11	0 entered 18 times ,1, 8	Total score is 9	Used to check whether application handles final frame's special condition properly.
12	0 entered 18 times ,10, 9, 3	Invalid input message	Used to check whether application handles final frame's special condition properly
13	0 entered 18 times ,1, 9, 3	Total score is 13	Used to check whether application handles final frame's special condition properly
14	0 entered 18 times ,10, 10, 12	Invalid input message	Used to check whether application handles final frame's special condition properly
15	0 entered 18 times ,10, 10, 4	Total score is 24	Used to check whether application handles final frame's special condition properly
16	0 entered 18 times ,0, 10, 2	Total score is 12	Used to check whether application handles final frame's special condition properly
17	4g	Invalid input message	Used to check whether application can handle invalid inputs
18	Return key	Display input prompt	Used to check whether application can handle invalid inputs
19	s	Invalid input message	Used to check whether application can handle invalid inputs
20	4 g	Invalid input message	Used to check whether application can handle invalid inputs

An application that passes 15 or more test cases can be thought of as a robust application. As noted in Table 8, 80% of the TDD pairs' code passed majority of the test cases (passing 16 to 20 test cases), while only 50% of the non-TDD pairs managed to achieve the same.

Table 8: Number of new Test Cases Passed

No of test cases passed	TDD pair	Non-TDD pair
0 – 10	-	1
11 – 15	1	2
16 – 20	5	3

The internal code quality was evaluated using OOM. Apart from the OOM metrics used in student code evaluation, the professional developers' code was evaluated using metrics such as LOCOM to measure the cohesion and coupling in the code. These metrics were included as the professional developers were able to completely develop the application. Table 9 provides the detailed result of metrics evaluation. As with students' code, the SD, mean, median and IQR of the measurements were calculated.

Table 9: Detailed metrics evaluation of professional developers' code

Metrics	Pair type	Mean	Median	SD	IQR
AHF	TDD	69.8	84.5	38.8	62.5
	Non-TDD	52.5	58.5	27.6	34.5
CBO	TDD	6.2	7	1.8	2.8
	Non-TDD	5.5	5.5	1.1	1.5
CC	TDD	25	27.5	6.4	8.8
	Non-TDD	24.3	23	4.2	6
LOC	TDD	226	249.5	53.5	98.8
	Non-TDD	235.2	220	54.3	77.3
LOCOM1	TDD	11.8	12	5.5	7.8
	Non-TDD	12.2	9.5	13.7	26
LOCOM2	TDD	89.3	100	20.1	23
	Non-TDD	98.7	100	3.8	2

Table 9 continued.

LOCOM3	TDD	59.2	70	20.5	28.8
	Non-TDD	65.8	68	17.5	22.8
MHF	TDD	2	0	3.2	5.5
	Non-TDD	2	0	4.9	3
NOC	TDD	3.3	3	2.1	3
	Non-TDD	4.7	3	4.1	3.5
NOO	TDD	10	9.5	2.2	3.8
	Non-TDD	9.7	10	3.2	7
RFC	TDD	32.8	27.5	22.2	25
	Non-TDD	20.5	19.5	3.9	6.5
WMPC1	TDD	25	27.5	6.4	8.8
	Non-TDD	24.3	23	4.1	6
WMPC2	TDD	15.5	16	4.0	8.3
	Non-TDD	14.3	15	4.4	8

The results remain inconclusive due to various reasons that include small number of lines of code, small data points, and lack of conclusive differences in the values.

Qualitative analysis

The qualitative analysis on the effectiveness of TDD was done by conducting a survey among the 24 professional developers who participated in the experiments. The survey can be found in Appendix E. Other research methods such as personal interviews were ruled out due to cost and inconvenience. The survey consisted of nine close-ended questions (intended to collect developers' opinion on perceived notions on TDD) and two open-ended

questions (designed to elicit subjective responses on the difficulties in using the approach). Close-ended questions consisted of a statement followed by a multiple of choices, presented using a scale ranging from “strongly agree” to “strongly disagree”. Such questions provide the same frame of reference for all participants to use when determining answers.

The nine close-ended questions were aimed at eliciting the developers’ opinion on three concerns:

(1) How productive is the method for programmers?

(2) How effective is the approach?

(3) How difficult is the approach to adopt?

The productivity concern was evaluated by asking questions on whether they perceive that TDD (1) takes less time for code development; (2) facilitates better requirements understanding; and (3) reduces debugging effort. The effectiveness of the approach was evaluated by asking questions on whether they perceive that the TDD approach (1) yields higher quality and better structured code; (2) promotes simpler design; and (3) is effective overall. The final concern was represented through questions concerning whether the lack of upfront detailed design is a hindrance and whether getting into a TDD mindset was very difficult.

The survey was administrated before the experiment was conducted. The responses to the survey were recorded into a Microsoft Excel worksheet, with responses for the nine closed questions assigned values as follows: strongly agree (value = 2), agree (value = 1), disagree (value = -1), and strongly disagree

(value = -2). In all there were 24 valid responses. A reliability analysis was then performed to determine whether or not it was statistically valid to aggregate the responses into the stated three subscales or indexes (productivity, effectiveness, and demanding) using Cronbach's Coefficient Alpha test to measure the internal consistency of each of these three subscales. The assumption was that all of the questions within a subscale (for example, the productivity subscale) measured the same attribute and therefore individuals should answer all of the questions within the subscale similarly. The Cronbach's Coefficient Alpha measures this level of consistency. If the value on this test is over 0.8, it is considered a valid to aggregate the answers. The statistical significance of each response was then evaluated for each of these sections using Spearman's Rho test.

Table 10: Professional Developers' Survey Results

Topic	In favor of TDD	% Who Strongly Agree	% Who Agree	% Who Disagree	% Who Strongly Disagree
TDD facilitates better requirements understanding.	Yes	41.7	45.8	12.5	-
TDD reduces debugging effort.	Yes	29.2	66.7	4.2	-
TDD approach requires less development time.	No	8.3	41.7	50	-
TDD yields better quality code.	Yes	41.7	50	8.3	-
TDD yields better-structured code.	Yes	29.2	50	20.8	-
TDD promotes simpler design.	Yes	29.2	50	20.8	-
Overall, TDD is visibly effective.	Yes	20.8	50	29.2	

Table 10 continued.

Lack of upfront design is a hindrance.	Yes	-	33.3	54.2	12.5
Getting into TDD mindset is difficult.	Yes	8.3	45.8	45.8	-

The above table summarizes the results of the survey. As noted in the table majority (eight out of nine) of the responses to close-ended questions were in favor of TDD approach. The details of the results are discussed below.

Productivity

The results of the survey indicate a strong support for the TDD approach. The majority of the developers responded positively to the questions about programmer productivity - whether TDD approach facilitates better understanding of requirements and reduces debugging effort considerably (87.5% and 95.8% respectively). However, the respondents were divided equally on the question of whether code development requires less time. The alpha test – which indicates whether it is statistically significant to aggregate the three questions - yielded a value of 0.84, which deems the questions can be aggregated to represent the programmers' productivity. The Spearman Rho test indicated that all the responses (for the three questions) were statistically significant at the 0.01 level ($p < 0.01$). Hence, taking the average of all the positive responses on the three questions (related to programmer productivity), about 78% of the developers thought that TDD improves programmer productivity.

Effectiveness

The majority of developers responded positively to all the questions on effectiveness of the approach. Ninety two percent (92%) of developers agreed that TDD yields higher quality code while 79% said it yields better structured code. Seventy nine percent (79%) of developers said that TDD promotes simpler design. However, only 71% thought that overall the approach was noticeably effective. Again conducting the alpha test (value of 0.95) concluded that it is valid to aggregate the four questions. The Spearman Rho test also indicated that all the responses were statistically significant ($p < 0.01$). Hence it can be interpreted that on an average 80% of developers strongly believed that TDD approach is effective resulting in improved code quality.

Adopting Difficulties

The response of developers to questions regarding adopting difficulties indicate that there some concerns about the approach. On question about difficulty in getting into TDD mindset 56% indicated that the transformation was difficult and a minority (23%) indicated that the lack of upfront design phase in TDD was a hindrance. Hence aggregating and taking average of the negative response, 40% of the developers thought that the approach faces adopting difficulties.

In response to the two open-ended questions developers expressed various concerns about TDD. The most repeated concerns include:

- knowing how and what to test
- testing overhead involved (creation of mock objects and maintaining huge library of test cases)

- not being economical in terms of time taken and the need for constant refactoring.

One developer also expressed the concern about finding the right balance between testing and coding.

APPENDIX D: Raw Data of all Experiments

This section compiles all the data obtained from the four experiments conducted. The table below shows the number of test cases passed by student code. As noted before a total of seven test cases were used to evaluate the student code.

Table 11: Raw Data Table on Test Cases Passed by Student Developers

Pair Type	Pair Number	Number of Test cases passed	Specific Test cases passed
TDD	1	2/7	1,2
TDD	2	5/7	1,2,3,4,5
TDD	3	2/7	1,2
TDD	4	5/7	1,2,3,4,7
TDD	5	4/7	1,2,4,5
TDD	6	2/7	1,2
TDD	7	5/7	1,2,3,4,5
TDD	8	4/7	1,2,3,4
TDD	9	4/7	1,2,4,5
TDD	10	4/7	1,2,3,4
TDD	11	2/7	1,3
TDD	12	4/7	1,2,3,4
TDD	13	2/7	1,2
TDD	14	6/7	1,2,3,4,5,7
TDD	15	2/7	1,2
TDD	16	7/7	1,2,3,4,5,6,7
TDD	17	2/7	1,2
TDD	18	6/7	1,2,3,4,5,6
TDD	19	2/7	1,2
TDD	20	2/7	1,2
TDD	21	0/7	0
TDD	22	0/7	0
Non-TDD	1	2/7	1,2
Non-TDD	2	2/7	1,2

Table 11 continued.

Non-TDD	3	2/7	1,2
Non-TDD	4	2/7	1,2
Non-TDD	5	4/7	1,2,4,5
Non-TDD	6	5/7	1,2,3,4,5
Non-TDD	7	5/7	1,2,3,4,5
Non-TDD	8	5/7	1,2,3,4,5
Non-TDD	9	1/7	1
Non-TDD	10	3/7	1,2,3
Non-TDD	11	1/7	1
Non-TDD	12	6/7	1,2,3,4,5,7
Non-TDD	13	5/7	1,2,3,4,5
Non-TDD	14	1/7	1
Non-TDD	15	2/7	1,2
Non-TDD	16	2/7	1,2
Non-TDD	17	6/7	1,2,3,4,5,7
Non-TDD	18	4/7	1,2,3,4
Non-TDD	19	6/7	1,2,3,4,5,7
Non-TDD	20	0/7	0
Non-TDD	21	0/7	0
Non-TDD	22	0/7	0
Non-TDD	23	0/7	0
Non-TDD	24	0/7	0
Non-TDD	25	0/7	0
Non-TDD	26	0/7	0
Non-TDD	27	0/7	0
Non-TDD	28	0/7	0
Non-TDD	29	0/7	0
Non-TDD	30	0/7	0
Non-TDD	31	0/7	0
Non-TDD	32	0/7	0

Table 11 continued.

Non-TDD	33	0/7	0

The next table compiles the number of test cases passed by professional code. As noted before a set of twenty test cases were used to evaluate the professional code.

Table 12: Raw Data Table on Test Cases Passed by Professional Developers

Pair Type	Pair Number	Number of Test cases passed	Specific Test cases failed
TDD	1	18/20	10 , 12
TDD	2	15/20	2,9,10,12,14
TDD	3	20/20	0
TDD	4	20/20	0
TDD	5	19/20	14
TDD	6	16/20	9,10,12,14
Non-TDD	1	13/20	1,2,4,9,10,12,14
Non-TDD	2	9/20	1,2,4,8,9,10,12,14, 17,18,19
Non-TDD	3	17/20	17,18,19
Non-TDD	4	15/20	15,17,18,19,20
Non-TDD	5	16/20	3,10, 12,14
Non-TDD	6	16/20	2,3,8,20

As noted before, nine metrics were used to evaluate student codes. The results of metric evaluation are included in table below. The legend for the table is as follows:

AHF – Attribute Hiding Factor

CC – Cyclic Complexity

MHF – Method Hiding Factor

NOC – Number of Classes

NOO – Number of Operations

RFC – Response for Class

WMPC – Weighed Methods Per Class

Table 13: Raw Metrics Data Table of Student Code

Group/ Pair No	AHF	CC	LOC	MHF	NOC	NOO	RFC	WMP C1	WMP C2
TDD / 1	0	6	38	0	1	3	9	6	7
TDD / 2	43	11	54	20	2	3	8	11	6
TDD / 3	86	4	25	0	2	3	5	4	5
TDD / 4	0	24	102	0	1	10	18	24	24
TDD / 5	0	15	58	0	1	5	7	15	7
TDD / 6	0	11	49	0	1	4	6	11	10
TDD / 7	0	13	74	0	1	2	2	13	5
TDD / 8	71	14	101	0	2	4	16	14	8
TDD / 9	0	2	33	0	1	2	2	2	5
TDD / 10	0	15	75	0	1	7	15	15	12
TDD / 11	71	19	88	0	2	12	13	19	17
TDD / 12	0	9	43	0	1	3	5	9	6
TDD / 13	100	8	93	0	2	3	8	8	5
TDD / 14	50	18	149	0	3	9	33	18	12
TDD / 15	100	8	45	0	2	4	10	8	6
TDD / 16	100	16	103	0	2	6	13	16	12
TDD / 17	100	5	76	0	2	4	10	5	9
TDD / 18	43	17	84	20	2	7	10	17	13
TDD / 19	100	8	72	11	2	4	8	8	7
TDD / 20	100	6	42	0	2	3	6	6	6
Non-TDD / 1	0	5	58	0	2	2	9	5	3
Non-TDD / 2	0	10	48	0	1	7	12	10	8
Non-TDD / 3	0	11	79	0	1	4	9	11	5
Non-TDD / 4	100	6	81	22	3	4	13	6	5
Non-TDD /5	100	15	85	60	2	3	9	15	4
Non-TDD / 6	100	23	230	0	2	11	18	23	16
Non-TDD /7	0	20	117	0	1	3	8	20	10
Non-TDD / 8	0	12	77	0	1	2	8	12	5

Table 13 continued.

Non-TDD / 9	0	11	85	0	1	4	10	11	6
Non-TDD / 10	100	16	91	0	3	9	19	16	12
Non-TDD / 11	0	13	115	0	2	1	8	13	3
Non-TDD / 12	57	22	125	71	2	6	13	22	10
Non-TDD / 13	100	12	123	12	3	7	14	12	15
Non-TDD / 14	0	4	37	0	1	4	10	4	5
Non-TDD / 15	0	15	63	0	1	3	7	15	7
Non-TDD / 16	0	5	52	0	1	2	9	5	8
Non-TDD / 17	0	18	84	0	1	5	11	18	8
Non-TDD / 18	0	12	84	0	1	6	12	12	8
Non-TDD / 19	92	19	166	35	3	10	18	19	22

The table below compiles the results of the metric evaluation done on professional developers' code. The legend for the table is as follows:

CBO – Coupling Between Objects CF – Coupling Factor

Locom – Lack of Cohesion of Methods

Table 14: Raw Metrics Data Table of professional code

Group / Pair No	**AHF**	**CBO**	**CC**	**CF**	**LOC**	**Locom 1**	**Locom 2**
TDD / 1	83	7	30	150	246	5	100
TDD / 2	100	3	27	200	142	13	100
TDD / 3	50	7	23	267	265	8	50
TDD / 4	86	8	13	110	253	21	86
TDD / 5	0	5	29	-	177	11	100
TDD / 6	100	7	28	133	273	13	100
Non-TDD / 1	50	4	19	100	204	0	100
Non-TDD / 2	70	6	23	83	195	29	100
Non-TDD / 3	61	6	23	14	335	19	100

Table 14 continued.

Non-TDD / 4	56	7	23	300	250	25	100
Non-TDD / 5	0	5	31	250	236	0	100
Non-TDD / 6	78	5	27	250	191	0	92

Remaining Columns of Table 14.

Group/ Pair No	**Locom 3**	**MHF**	**NOC**	**NOO**	**RFC**	**Wmpc 1**	**Wmpc 2**
TDD / 1	70	7	3	12	32	30	20
TDD / 2	70	0	2	10	15	27	14
TDD / 3	21	0	3	9	31	23	11
TDD / 4	50	0	7	7	24	13	11
TDD / 5	71	0	1	9	19	29	18
TDD / 6	73	5	4	13	76	28	19
Non-TDD / 1	33	0	3	6	16	19	10
Non-TDD / 2	79	0	4	13	23	23	17
Non-TDD / 3	82	12	13	9	20	23	13
Non-TDD / 4	70	0	3	13	27	23	20
Non-TDD / 5	65	0	2	6	19	31	9
Non-TDD / 6	66	0	3	11	18	27	17

The final table contains the time taken by professional developers. In case of non-TDD developers, the time taken was recorded into three sections (Design, Code and testing phase).

Table 15: Time Taken by Each Professional Programming Pair

Group / Pair No	Design Time	Implementation Time	Testing Time	Total time Non-TDD	Total time TDD
Non-TDD / 1	40 mins	85 mins	20 mins	145 mins	-
Non-TDD / 2	20 mins	90 mins	60 mins	170 mins	-
Non-TDD / 3	18 mins	180 mins	150 mins	348 mins	-
Non-TDD / 4	10 mins	170 mins	130 mins	310 mins	-
Non-TDD / 5	10 mins	170 mins	90 mins	270 mins	-
Non-TDD / 6	10 mins	200 mins	105 mins	315 mins	-
TDD / 1	-	-	-	-	200 mins
TDD / 2	-	-	-	-	240 mins
TDD / 3	-	-	-	-	366 mins
TDD / 4	-	-	-	-	407 mins
TDD / 5	-	-	-	-	270 mins
TDD / 6	-	-	-	-	380 mins

APPENDIX E: Survey on Test Driven Development

Survey on Test-First Design

Please mark (X) on one in five columns indicating your views about the following topics

Topic	Strongly Agree	Agree	Disagree	Strongly Disagree
Test First Design (TFD) approach develops code in **less time**				
TFD approach yields **better quality code**				
TFD approach yields **better structured code**				
TFD facilitate to **understand requirements/specifications** properly before coding.				
TFD approach leads to **more simple design**				
TFD approach **reduces debugging effort (time)** considerably.				
Lack of upfront **detailed design** in TFD approach is a hindrance				
Getting into TFD **mindset** is **very difficult**.				
TFD approach is **visibly effective** than other methods				

(over)

Please choose one among the following:

Which is the most difficult task associated with TFD?

A. Getting into the test first mindset
B. The overhead associated with writing/running/updating test cases
C. Keeping track of frequent changes occurring in design.
D. Lack of detailed upfront designs to give a complete picture.
E. Others (please specify) :-

What is the biggest disadvantage of TFD?

A. The need to take design decision through out coding
B. Not very practical in terms of time taken
C. Won't allow upfront detailed design.
D. The need to write test cases for all functionality
E. Others (please specify) :-

Please provide any other comments you have on the TFD practice.

APPENDIX F: Problem statement TDD version

You are a Test-First Pair
A Bowling Alley Scoring Game (adapted from [15])

The application must be written in Java and you must use JUnit as the automated testing tool. Please record the following times: (1) Start time and (2) Finish time.

Problem Statement:
The objective is to develop an application that calculates the score of a SINGLE bowling game. The input of the application is simply a sequence of throws. A throw is an integer that tells how many pins were knocked down by the ball. The output is the data on a standard bowling score card -- a set of frames populated with the pins knocked down by each throw, and marks denoting spares and strikes.

Requirements:

- The game must record (1) the score of the game,
- Each game consists of 10 frames.
- The player has two throws in each frame. The game must record how many pins were knocked down by each throw and the current score of the game at the end of each frame.
- The application should be capable of handling strike and spare cases.
    - A strike is when all the 10 pins are knocked down in the first throw. When this occurs the frame is done (no second roll for that frame) and the next two throws are added to the current frame's score. Therefore, the current score for that frame cannot be computed until the next frame or next two frames (in case of triple strike) has completed.
    - A spare occurs when all 10 pins are knocked down in two throws. Then the next throw (of the next frame) is added to the current frame's score. So the current score for that frame cannot be computed until the first throw of the next frame has completed.
    - In the tenth frame, if a strike is bowled, the bowler gets two bonus balls to add on to that strike. The bonus throws score nothing by themselves, they only add on to the tenth frame.
    - If the bowler throws a spare in the tenth frame, he gets one more throw to add on to that spare. This bonus throws scores nothing by itself, they only add onto the previous throw.
- The application is standalone (i.e. it has a main method). The input (i.e. number of pins knocked down in each throw) is entered through keyboard and the output is displayed on the screen. There is no need for a GUI. Create the simplest input-output that could possibly work.
- At the end of each frame, the game must display (1) frame number, (2) how many pins were knocked down in each throw, (3) game score per frame and (4) the current score of the game if available. If a strike or a spare presents computing the final score, display "not available."
- The type and number of automated test cases to be written are decided by developers.
- The application should handle erroneous input (such as negative inputs) and other error conditions gracefully.

APPENDIX G: Problem statement non-TDD version

You are a Traditional (Non-TFD) Pair
A Bowling Alley Scoring Game

The application must be written in Java. You DON'T have to develop GUI for the application. Please record the following times: (1) Start of experiment (2) Coding start time (3) Testing/debugging start time and (4) Finish time.

Problem Statement:
The objective is to develop an application that calculates the score of a SINGLE bowling game. The input of the application is simply a sequence of throws. A throw is an integer that tells how many pins were knocked down by the ball. The output is the data on a standard bowling score card -- a set of frames populated with the pins knocked down by each throw, and marks denoting spares and strikes.

Requirements:

- The game must record (1) the score of the game
- Each game consists of 10 frames.
- The player has two throws in each frame. The game must record how many pins were knocked down by each throw and the current score of the game at the end of each frame.
- The application should be capable of handling strike and spare cases.
    - A strike is when all the 10 pins are knocked down in the first throw. When this occurs the frame is done (no second roll for that frame) and the next two throws are added to the current frame's score. Therefore, the current score for that frame cannot be computed until the next frame or next two frames (in case of triple strike) has completed.
    - A spare occurs when all 10 pins are knocked down in two throws. Then the next throw (of the next frame) is added to the current frame's score. So the current score for that frame cannot be computed until the first throw of the next frame has completed.
    - In the tenth frame, if a strike is bowled, the bowler gets two bonus balls to add on to that strike. The bonus throws score nothing by themselves, they only add on to the tenth frame.
    - If the bowler throws a spare in the tenth frame, he gets one more throw to add on to that spare. This bonus-throw scores nothing by itself, they only add onto the previous throw.
- The application is standalone (i.e. it has a main method). The input (i.e. number of pins knocked down in each throw) is entered through keyboard and the output is displayed on the screen. There is NO need for a GUI. Create the simplest input-output that works.
- At the end of each frame, the game must display (1) frame number, (2) how many pins were knocked down in each throw, (3) game score per frame and (4) the current score of the game. If strike/spare present in final score, display "N/A".
- Write automated test cases using JUnit after design/coding of the application. The type and number of test cases to be written are decided by the developer.
- The application should handle erroneous input (such as negative inputs) and other error conditions gracefully.