Software Architecture Improvement thru Test-Driven Development: *An Empirical Study*

A Ph.D. Research Proposal

April 7, 2005

David S. Janzen
M.S., EECS, 1993
University of Kansas

Ph.D. Dissertation Committee Members

Dr. Hossein Saiedian, Professor, *Chair*
Dr. Arvin Agah, Associate Professor
Dr. Perry Alexander, Associate Professor
Dr. John Gauch, Associate Professor
Dr. Carey Johnson, Associate Professor

Abstract

Despite a half century of advances, the software construction industry still shows signs of immaturity. Professional software development organizations continue to struggle to produce reliable software in a predictable and repeatable manner. While a variety of development practices are advocated that might improve the situation, developers are often reluctant to adopt new, potentially better practices based on anecdotal evidence alone. As a result, empirical software engineering has gained credibility as a discipline that provides scientific data about practice efficacy on which developers can make critical decisions.

This research proposes to apply empirical software engineering techniques to evaluate a new approach that offers the potential to significantly improve the state of software construction. Test-driven development (TDD) is a disciplined software development practice that focuses on software design by first writing automated unit-tests followed by production code in short, frequent iterations. TDD focuses the developer's attention on a software's interface and behavior while growing the software architecture organically.

TDD has gained recent attention with the popularity of the Extreme Programming agile software development methodology. Although TDD has been applied sporadically in various forms for several decades, possible definitions have only recently been proposed. Advocates of TDD rely primarily on anecdotal evidence with relatively little empirical evidence of the benefits of the practice. A small number of studies have looked at TDD only as a testing practice to remove defects. However, there is no research on the broader efficacy of TDD. This research will be the *first* comprehensive evaluation of how TDD effects overall software architecture quality beyond just defect density.

My hypothesis is that TDD improves overall software quality including characteristics such as extensibility, reusability, and maintainability without significantly impacting cost and programmer productivity. I intend to examine this hypothesis by designing and administering a series of longitudinal empirical studies with undergraduate students and professional programmers.

Controlled experiments will be conducted in a set of undergraduate courses. Student programmers will be taught to write automated unit-tests integrated with course topics using a new approach which I am calling test-driven learning (TDL). Formal experiments will then compare the quality of software produced with TDD

to software produced with a more traditional test-last development approach. A case study or controlled experiment will also be conducted with more experienced programmers in a professional environment. In all of the studies, programmer performance, attitudes toward testing, and future voluntary usage of TDD will also be assessed.

The combination of studies in academic and professional environments will establish external validity of the research as well as provide valuable information regarding the effectiveness of TDD at various levels of maturity. The research should also produce several important by-products including pedagogical materials, a framework for future studies, and observations regarding TDD's fit in the undergraduate computer science curriculum.

Positive results from these studies have the potential of significantly improving the state of software construction. For the first time, professional developers will be able to examine empirical evidence of TDD efficacy both as a testing and as a design practice. Additionally, computer science faculty will be encouraged to incorporate TDD into curricula, resulting in better student design and testing skills. Improved pedagogy combined with widespread adoption of TDD offer the potential of radically improving the software engineering community's ability to reliably produce, reuse, and maintain quality software.

Table of Contents

List of Figures

List of Tables

1

Problem Definition

This chapter summarizes the problem to be solved, the solution approach, and the expected contributions of this research. It provides a brief introduction to the test-driven development strategy, summarizes the research to be conducted, and identifies the significance of the expected contributions.

1.1 State of Software Construction

Software construction is a challenging endeavor. It involves a complex mix of creativity, discipline, communication, and organization. The Standish Group has been studying the state of software projects since 1985 and their research demonstrates the difficulty organizations have successfully completing software projects. Table 1.1 compares 1995 statistics [1] with those from the third quarter of 2004 [2]. The 2004 numbers result from over 9,000 software projects from all around the world (58% US, 27% Europe, 15% other) developed by a wide-range of organizations (45% large, 35% mid-range, 20% small) in a variety of domains. Successful projects are those that deliver the requested functionality on-time and within budget. Challenged projects are either late, over budget, and/or deliver less than the required features and functions. Failed projects have been canceled prior to being completed or they were delivered and never used.

As the table demonstrates, the state of software construction has improved con-

Year	Successful Projects	Challenged Projects	Failed Projects
1995	16.2%	52.7%	31.1%
2004	29%	53%	18%

Table 1.1: Standish Group Comparison of IT Project Success Rates

siderably since 1994. However, still less than one third of all projects are completed successfully and 18% or nearly one in five projects still fail completely.

Software construction has been compared to constructing buildings, bridges, and automobiles among others. In his 1994 Turing Award lecture, Alan Kay opined that software construction is similar in maturity to building the ancient Egyptian pyramids where thousands of workers toiled for years to build a facade over a rough inner structure. He compared this with the efficiency of constructing the Empire State Building which took just over one year and about seven million man hours to complete. He noted that the process was so efficient that the steel was often still warm from the mills in Pittsburgh when it was being assembled in New York.

While the Empire State Building is a fantastic goal for software construction, there are clearly many differences in the nature of skyscraper construction and software construction. Plus we might note that the Empire State Building set a record for skyscraper construction that still stands today. The point of Dr. Kay's discussion is still quite clear and consistent with the Standish numbers: software construction has much room for improvement.

1.2 State of Software Research

Improving the state of software construction is of considerable interest not just in professional software development organizations. Much research has been and continues to be conducted. However, as Brooks points out in his classic 1987 paper [30], most software research focuses on the wrong topics if we want to improve the state of software construction. Brooks classifies software activities as essential and accidental tasks. Essential tasks focus on conceptual structures and mechanisms for forming abstractions with complex software, while accidental tasks focus more on technologies that facilitate mapping abstractions into actual programs.

In the years since Brooks' paper, there is still much attention on accidental tasks. Web services, modern integrated development environments, and new languages such as Java and C# are just a few examples. Professional training courses are still predominantly focused on new technologies, and undergraduate curriculums continue to emphasize many technical skills while paying relatively little attention to more conceptual and organizational skills such as software design and software development methods.

Attention has been drawn, however, to many essential tasks such as visual modeling, software organization, and development methods. The context for the research proposed in this paper in fact lies in the very iterative and evolutionary types of development models that Brooks was advocating.

Unfortunately few new ideas are thoroughly examined. As Gibbs wrote in 1994, "after 25 years of disappointment with apparent innovations that turned out to be irreproducible or unscalable, many researchers concede that computer science needs

an experimental branch to separate the general results from the accidental." [72]

1.2.1 Empirical Software Engineering

Empirical software engineering has emerged as a valuable research discipline that examines ideas in software engineering. While empirical studies will rarely produce absolute repeatable results, such studies can provide evidence of causal relationships, implying results that will most likely occur in given contexts.

Empirical software engineering projects have received significant government and corporate funding. Research centers have been founded such as the "NSF Center for Empirically-Based Software Engineering" and the "Centre for Advanced Software Engineering Research." Many journals such as IEEE Transactions on Software Engineering specifically request empirical studies and Springer publishes a dedicated journal titled "Empirical Software Engineering: An International Journal."

1.3 Proposed Research

This research proposes to apply empirical software engineering techniques to examine a new approach that holds promise to significantly improve the state of software construction. Test-driven development is a relatively new, unstudied development strategy that has caught the attention of a number of prominent computer scientists. Steve McConnell in his 2004 OOPSLA keynote address included test-driven development as the only yet-to-be-proven development practice among his top ten advances of the last decade.

The next section will briefly introduce test-driven development and the remainder of this proposal will outline how empirical software engineering practices will be applied to examine test-driven developments efficacy or ability to produce desirable results. In particular this research will assess how well test-driven development improves software design quality while also reducing defect density, and whether these improvements come with a cost of increased effort or time.

1.4 Introduction to Test-Driven Development

Test-driven development (TDD) [16] is a software development strategy that requires that automated tests be written prior to writing functional code in small, rapid iterations. Although TDD has been applied in various forms for several decades [48] [33], it has gained increased attention in recent years thanks to being identified as one of the twelve core practices in Extreme Programming (XP) [15].

Extreme Programming is a lightweight, evolutionary software development process that involves developing object-oriented software in very short iterations with

relatively little up front design. XP is a member of a family of what are termed agile methods [14]. Although not originally given this name, test-driven development was described as an integral practice in XP, necessary for analysis, design, and testing, but also enabling design through refactoring, collective ownership, continuous integration, and programmer courage [15].

In the few years since XP's introduction, test-driven development has received increased individual attention. Besides pair programming [75] and perhaps refactoring [29], it is likely that no other XP practice has received as much individual attention as TDD. Tools have been developed for a range of languages specifically to support TDD. Books have been written explaining how to apply TDD. Research has begun to examine the effects of TDD on defect reduction and quality improvements in both academic and professional practitioner environments. Educators have begun to examine how TDD can be integrated into computer science and software engineering pedagogy. Some of these efforts have been in the context of XP projects, but others are independent.

1.5 Significance of Expected Contributions

Test-driven development advocates claim that TDD is more about design than it is about testing. The fact that it involves both design and testing indicates that if it works, there are many benefits to be gained.

Software development organizations are hard-pressed to select the most effective set of practices that produce the best quality software in the least amount of time. Empirical evidence of a practice's efficacy are rarely available and adopting new practices is time-consuming and risky. Such adoptions often involve a significant conceptual shift and effort in the organization including but not limited to developer training, acquiring and implementing new tools, and collecting and reporting new metrics.

In 2000, Laurie Williams completed her PhD at the University of Utah. Her dissertation presented the results of empirical studies she conducted on pair programming. This new approach has since gained significant popularity, largely based on the empirical evidence. Williams has gone on to publish widely on pair programming and related topics, and she has been very successful in attracting both government and corporate funding for her work.

This research should contribute empirical results perhaps even more beneficial than Williams' results on pair programming. While pair programming has been shown to improve defect detection and code understanding, TDD stands to do the same with the advantage of also improving software designs. The results from this study will assist professional developers in understanding and choosing whether to adopt test-driven development. For the first time, it will reveal the effects on software design quality from applying TDD. It will explore many important quality

aspects beyond defect density such as reusability and maintainability.

In addition, this research will make important pedagogical contributions. The research will contribute a new approach to teaching that incorporates teaching with tests called "test-driven learning." The research will demonstrate whether undergraduate computer science students can learn to apply TDD, and it will examine at what point in the curriculum TDD is best introduced.

If TDD proves to improve software quality at minimal cost, and if this research shows that students can learn TDD from early on, then this research can have a significant impact on the state of software construction. Software development organizations will be convinced to adopt TDD in appropriate situations. New textbooks can be written applying the test-driven learning approach. As students learn to take a more disciplined approach to software development, they will carry this into professional software organizations and improve the overall state of software construction.

1.6 Summary of Remaining Chapters

Chapter two will more thoroughly present the context in which TDD has developed and evolved. Test-driven development will be defined more precisely. Iterative, incremental, and evolutionary development processes will be discussed, along with historical references to various emerging forms of TDD. References to TDD in academia will be noted, and particular attention will be given to the recent context in which TDD has gained popularity.

Chapter three will survey the current state of research on TDD, independent of its context. It will not attempt to survey XP research that may provide indirect knowledge of TDD. It will attempt to provide the necessary definitions and background to fully understand TDD. Then it will attempt to establish the current state of evaluative research on TDD. Finally it will propose possible future directions for further research on TDD, based on identified shortcomings in current research.

Chapter four presents the methods by which this research will be carried out. A new pedagogical approach called test-driven learning (TDL) will be incorporated into existing courses, and formal experiments will be conducted. The chapter identifies tools and metrics that will be utilized, and discusses how the results will be analyzed and assessed.

Chapter five outlines the research schedule and identifies potential challenges to be overcome, possible risks, and expected contributions resulting from this research.

Chapter six discusses the significant contributions expected from this research. It discusses how the research will be evaluated and how external validity will be established through peer-reviewed publications and a case study with professional programmers. The chapter ends with a summary of the work to be completed and its potential to improve the state of software construction and pedagogy.

2

Test-Driven Development in Context

This chapter presents the context wherein test-driven development is emerging. It surveys a variety of definitions for test-driven development, and provides a new one for the purposes of this research. It discusses historical and recent events that have contributed to the current understanding of test-driven development.

2.1 Definitions of TDD

Although its name would imply that TDD is a testing method, a close examination of the name reveals a more complex picture.

2.1.1 Significance of "Test" in TDD

As the first word implies, test-driven development is concerned with testing. More specifically it is about writing automated unit tests. Unit testing is the process of applying tests to individual units of a program. There is some debate regarding what exactly is a unit in software. Even within the realm of object-oriented programming, both the class and the method have been suggested as the appropriate unit. Generally, however, we will consider a unit to be "the smallest possible testable software component" [21] which currently [17] appears to be the method or procedure.

Test drivers and function stubs are frequently implemented to support the execution of unit tests. Test execution can be either a manual or automated process and may be performed by developers or dedicated testers. *Automated* unit testing involves writing unit tests as code and placing this code in a test harness [21] or a framework such as JUnit [51]. Automated unit testing frameworks can reduce the effort of testing, even for large numbers of tests to a simple button click. In contrast, when test execution is a manual process, developers and/or testers may be required to expend significant effort proportional to the number of tests executed.

Traditionally, unit testing has been applied some time after the unit has been coded. This time interval may be quite small (a few minutes) or quite large (a few months). The unit tests may be written by the same programmer or by a designated tester. With TDD, however, unit tests are prescribed to be written *prior* to writing the code under test. As a result, the unit tests in TDD normally don't exist for very long before they are executed.

2.1.2 Significance of "Driven" in TDD

Some definitions of TDD seem to imply that TDD is primarily a testing strategy. For instance, according to [51] when summarizing Beck [17],

> Test-Driven Development (TDD) is a programming practice that instructs developers to write new code only if an automated test has failed, and to eliminate duplication. The goal of TDD is 'clean code that works.' [45]

However, according to XP and TDD pioneer Ward Cunningham, "Test-first coding is not a testing technique" [16]. In fact TDD goes by various names including Test-First Programming, Test-Driven Design, and Test-First Design. The *driven* in test-driven development focuses on how TDD informs and leads analysis, design and programming decisions. TDD assumes that the software design is either incomplete, or at least very pliable and open to evolutionary changes. In the context of XP, TDD even subsumes many analysis decisions. In XP, the customer is supposedly "on-site", and test writing is one of the first steps in deciding what the program should do, which is essentially an analysis step.

Another definition which captures this notion comes from The Agile Alliance [7],

> Test-driven development (TDD) is the craft of producing automated tests for production code, and using that process to drive design and programming. For every tiny bit of functionality in the production code, you first develop a test that specifies and validates what the code will do. You then produce exactly as much code as will enable that test to pass. Then you refactor (simplify and clarify) both the production code and the test code.

As is seen in this definition, promoting testing to an analysis and design step involves the important practice of refactoring [29]. Refactoring is a technique for changing the structure of an existing body of code without changing its external behavior. A test may pass, but the code may be inflexible or overly complex. By refactoring the code, the test should still pass *and* the code will be improved.

Understanding that TDD is more about analysis and design than it is about testing may be one of the most challenging conceptual shifts for new adopters of the practice. As will be discussed later, testing has traditionally assumed the existence of a program. The idea that a test can be written before the code, and even more, that the test can aid in deciding what code to write and what its interface should look like is a radical concept for most software developers.

2.1.3 Significance of "Development" in TDD

TDD is intended to aid the construction of software. TDD is not in itself a software development methodology or process model. TDD is a practice, or a way of developing software to be used in conjunction with other practices in a particular order and frequency in the context of some process model. As we will see in the next section, TDD has emerged within a particular set of process models. It seems possible that TDD could be applied as a micro-process within the context of many different process models.

We have seen that TDD is concerned with analysis and design. We don't want to ignore the fact that TDD also produces a set of automated unit tests which provide a number of side-effects in the development process. TDD assumes that these automated tests will not be thrown away once a design decision is made. Instead the tests become a vital component of the development process. Among the benefits, the set of automated tests provide quick feedback to any changes to the system. If a change causes a test to fail, the developer should know within minutes of making the change while it is still fresh in his or her mind. Among the drawbacks, the developer now has both the production code and the automated tests which must be maintained.

2.1.4 A New Definition of TDD

TDD definitions proposed to date assume an unspecified design and a commitment to writing automated tests for all non-trivial production code. Despite TDD's promise of delivering "clean code that works", many developers seem to be reluctant to try TDD. This reluctance is perhaps at least partially a result of the choice of overall development process in an organization. Obviously an organization that is applying XP is willing to attempt TDD. However, an organization that is using a more traditional approach is likely unable to see how TDD can fit. This and other factors affecting this choice will be more fully addressed in chapter three.

To expand the utility and applicability of TDD, I propose the following modification of the Agile Alliance definition:

> Test-driven development (TDD) is a software development strategy that requires that automated tests be written prior to writing functional code in small, rapid iterations. For every tiny bit of functionality desired, you first develop a test that specifies and validates what the code will do. You then produce exactly as much code as will enable that test to pass. Then you refactor (simplify and clarify) both the code under test and the test code. Test-driven development can be used to explore, design, develop, and/or test software.

This definition broadens TDD's sphere of influence by suggesting that TDD can be used to:

- explore a specified or unspecified design
- explore a new or unfamiliar component
- design software
- develop software given a design
- develop tests for software given only its interface

This definition removes the restrictions of working on an unspecified design and working only on production code. It introduces the possibility that TDD could be used as a prototyping mechanism for working out a potential design, without requiring the tests to stick around.

2.2 Survey of Software Development Methodologies

The remainder of this chapter discusses the context that has contributed to the emergence of test-driven development. This section provides a broad survey of software development methodologies to help establish a background for understanding test-driven development.

A software development process or methodology is a framework which defines a particular order, control, and evaluation of the basic tasks involved in creating software. Software process methodologies range in complexity and control from largely informal to highly structured. Methodologies may be classified as being prescriptive [63] or agile [14], and labeled with names such as waterfall [66], spiral [19], incremental [63], and evolutionary [35].

When an organization states that it is using a particular methodology, they are often applying on a project-scale certain combinations of smaller, finer-grained methodologies. For example, an organization may be applying an incremental model of development, building small, cumulative slices of the project's features. In each increment however, they may be applying a waterfall or linear method of determining requirements, designing a solution, coding, testing, and then integrating. Depending on the size of the increments and the time frame of the waterfall, the process may be labeled very differently with possibly very different results regarding quality and developer satisfaction.

If we break a software project into N increments where each increment is represented as I_i, then the entire project could be represented by the equation $\sum_{i=1}^{N} I_i$. If N is reasonably large, then we might label this project as an incremental project. However if $N \leq 2$, then we would likely label this as a waterfall project.

If the increments require the modification of a significant amount of overlapping software, then we might say that our methodology is more iterative in nature. Stated more carefully, for project P consisting of code C and iterations $I = \sum_{i=1}^{N} I_i$, if C_i is

the code affected by iteration I_i, then if project P is iterative, $C_i \cap C_{i+1} \neq \Theta$ for *most* i such that $1 < i < N$. Similarly, with the incremental and waterfall approaches, we might expect a formal artifact (such as a specification document) for documenting the requirements for that increment. If however, the artifact is rather informal (some whiteboard drawings or an incomplete set of UML diagrams), and was generated relatively quickly, then it is likely that we were working in the context of an agile process. Or, the approach and perspective of the architecture and/or design might cause us to label the process as aspect-oriented, component-based, or feature-driven.

Drilling down even further, we might find that individual software developers or smaller teams are applying even finer-grained models such as the Personal Software Process [42] or the Collaborative Software Process [73]. The time, formality, and intersection of the steps in software construction can determine the way in which the process methodology is categorized.

Alternatively, the order in which construction tasks occur influences a project's label, and likely its quality. The traditional ordering is requirements elicitation, analysis, design, code, test, integration, deployment, maintenance. This ordering is very natural and logical, however we may consider some possible re-orderings. Most re-orderings do not make sense. For instance, we would never maintain a system that hasn't been coded. Similarly, we would never code something for which we have no requirements. Note that requirements do not necessarily imply formal requirements, but may be as simple as an idea in a programmer's head. The Prototyping approach [20] has been applied when requirements are fuzzy or incomplete. With this approach, we may do very little analysis and design before coding. The disadvantage is that the prototype is often discarded even though it was a useful tool in determining requirements and evaluating design options.

When we closely examine the phases such as design, code, and test, we see that there are many finer-grained activities. For instance, there are many types of testing: unit testing, integration testing, and regression testing among others. The timing, frequency, and granularity of these tests may vary widely. It may be possible to conduct some testing early, concurrent with other coding activities. Test-driven development, however, attempts to re-order these steps to some advantage. By placing very fine-grained unit tests just prior to just enough code to satisfy that test, TDD has the potential of effecting many aspects of a software development methodology.

2.3 Historical Context of TDD

Test-driven development has emerged in conjunction with the rise of agile process models. Both have roots in the iterative, incremental and evolutionary process models, going back at least as early as the 1950's. In addition, tools have evolved and emerged to play a significant role in support of TDD. Curriculum seems to be lagging in its adoption of TDD, but XP in general has seen some favorable attention in the

academic community.

2.3.1 Early Test-Early Examples

Research on testing has generally assumed the existence of a program to be tested [37], implying a test-last approach. Moving tests, however, from the end of coding to the beginning is nothing new. It is common for software and test teams to develop tests early in the software development process, often along with the program logic. Evaluation and Prevention Life Cycle Models [33] integrated testing early into the software development process nearly two decades back. Introduced in the 1980s, the Cleanroom [27] approach to software engineering included formal verification of design elements early in the development process. There are even claims that some form of TDD was applied as early as the 1950's in NASA's Project Mercury [48].

However, prior to the introduction of XP in 1998, very little if anything has been written about the concept of letting small incremental automated unit tests *drive* the software development and particularly the design process. Despite the lack of published documentation, it is very possible that many developers have used a test first approach informally. Kent Beck even claims he

> learned test-first programming as a kid while reading a book on programming. It said that you program by taking the input tape ... and typing in the output tape you expect. Then you program until you get the output tape you expect. [16]

One might argue then that TDD merely gives a name and definition to a practice that has been sporadically and informally applied for some time. It seems, however, that TDD is a bit more than this. As Beck states, XP takes known best practices and "turns the knobs all the way up to ten." In other words, do them in the extreme. Many developers may have been thinking and coding in a test-first manner, but TDD does this in an extreme way, by always writing tests before code, making the tests as small as possible, and never letting the code degrade (test, code, refactor). As we will see next, TDD is a practice that must fit within a process model. The development of incremental, iterative, and evolutionary process models has been vital to the emergence of TDD.

2.3.2 Incremental, Iterative, and Evolutionary Development

Larman and Basili [48] survey a long history of iterative and incremental development models. Iterative development involves repeating a set of development tasks, generally on an expanding set of requirements. Evolutionary approaches as first presented by Gilb [35] involve iterative development which is adaptive and lightweight. Being adaptive generally refers to using feedback from previous iterations to improve and

change the software in the current iteration. Being lightweight often refers to the lack of a complete specification at the beginning of development, allowing feedback from previous iterations and from customers to guide future iterations. Lightweight can refer to other aspects such as the level of formality and degree of documentation in a process. The spiral model [19] is an evolutionary approach that incorporates prototyping and the cyclic nature of iterative development along with "risk-driven-iterations" and "anchor point milestones"

According to Pressman [63],

> The incremental model delivers a series of releases, called *increments*, that provide progressively more functionality for the customer as each increment is delivered.

It was within the context of such iterative, incremental, and evolutionary models that TDD developed. In fact, it appears that such iterative, incremental, and/or evolutionary approaches are prerequisite process models which are necessary for TDD to work. As we have stated, TDD is most closely associated with XP which is an iterative, evolutionary model. In fact, Beck claims that in order to implement XP, you must apply all of the incumbent practices. Leaving some out weakens the model and may cause the model to fail [15]. In order for TDD to influence software design, TDD requires that design decisions be delayed and flexible. With each new test, something new may be revealed about the code which requires a refactoring and possible change to the design as determined at that point. Automated tests give the programmer courage to change any code and know quickly if anything has broken, enabling collective ownership.

As originally proposed, TDD requires some form of an evolutionary process model. The converse, however, is clearly not true as many iterative, incremental, and/or evolutionary models have been proposed without the mention of TDD.

2.4 Emergence of Automated Testing Tools

Software tools have become important factors in the development of modern software systems. Tools ranging from compilers, debuggers, and integrated development environments (IDEs) through modeling and computer-aided software engineering (CASE) tools have improved and hence significantly increased developer productivity. Similarly testing tools have matured over the years.

Testing tools vary in purpose and scope, and will not be reviewed here. However, it is important to note the role that tools have played in the emergence of TDD. TDD assumes the existence of an automated unit testing framework. Such a framework simplifies both the creation and execution of software unit tests. Test harnesses are basically automated testing frameworks and have existed for some time. A test harness is a combination of test drivers, stubs, and possibly interfaces to other

subsystems [21]. Often such harnesses are custom-built, although commercial tools do exist to assist with test harness preparation [62].

JUnit [31] is an automated unit testing framework for Java developed by Erich Gamma and Kent Beck. JUnit is an essential tool for implementing TDD with Java. In fact, it might be argued that TDD and possibly even XP might not have received such wide popularity if it weren't for JUnit. JUnit-like frameworks have been implemented for a number of different languages, and the family of frameworks is referred to as xUnit [76].

Generally, xUnit allows the programmer to write sets of automated unit tests which initialize, execute, and make assertions about the code under test. Individual tests are independent of each other so that test order does not matter, and total numbers of successes and failures are reported. xUnit tests are written in the same language as the code under test and thus serve as first-class clients of the code. As a result, tests can serve as documentation for the code. On the other hand, because xUnit is implemented in the target language, the tool's simplicity and flexibility are determined somewhat by that language. For instance JUnit is very simple and portable, partly because it takes advantage of Java's portability through the bytecode/virtual machine architecture, it uses Java's ability to load classes dynamically, and it exploits Java's reflection mechanism to automatically discover tests. In addition, it provides a nice, portable graphical user interface that has even been integrated into popular integrated development environments like Eclipse.

A wide range of additional tools have emerged to support automated testing, particularly in Java. Several tools attempt to simplify the creation of mock objects [9] which are essentially stubs which stand-in for needed collaborating objects so that one can only test a particular object. Other tools such as Cactus [10] and Derby [11] can be used in conjunction with JUnit to automate tests which involve J2EE components or databases respectively.

The proliferation of software tools supporting TDD seems to be an indicator that TDD has widespread support and may be on its way to becoming an established approach. A significant factor in the use of TDD particularly in the Java community seems to be the simplicity and elegance of the JUnit tool. Programmers can develop unit-tests easily, and large suites of tests can be executed with a single click of a button, yielding quick results on the state of the system.

2.5 Early Testing in Curriculum

One indicator of the widespread acceptance of a software practice might be the undergraduate curriculum in computer science and software engineering. In some cases, academia has led practice in the field. In others, academia has followed. Software Engineering, iterative development and TDD seem to all fall in with the latter model.

Although much software engineering research has originated in academia, and found its way into common practice, the undergraduate curriculum in computer science and software engineering has tended to reflect and lag behind common practice in industry. Programming Language choice has commonly followed the needs of businesses. Process models have developed in practice and then later been reflected in curriculums.

The 1991 ACM Curriculum Guidelines [5] recommended that a small amount of lecture and lab time be given to iterative development processes (SE2) and verification and validation (SE5) (portions of eight hours each). The 2001 ACM Curriculum Guidelines [6] recommended that a perhaps even smaller amount of time be given to development processes (SE4) and software validation (SE6) (two and three hours respectively).

Undergraduate texts give little attention to comparative process models. Texts have limited coverage of software design and often have minimal coverage of testing techniques. The topics of software design and testing are often relegated to a software engineering course which may not even be required of all students.

There is much debate regarding the place of Extreme Programming in undergraduate education. Some [39] argue strongly in favor of using XP to introduce software engineering to undergraduates. Others [67] argue that XP and agile methods are only beneficial on a very limited basis. Still others [59] report mixed experiences.

Despite the mix of opinions on using XP in the undergraduate curriculum, TDD is receiving some limited exposure at this level. Some educators have called for increased design and testing coverage for some time. Some see TDD as an opportunity to incorporate testing throughout the curriculum, and not relegate it to an individual course [22].

TDD tools have found their way into early programming education. BlueJ [47], a popular environment for learning Java has incorporated JUnit and added helps for building test cases at an early stage in a programmer's learning cycle [61]. JUnit has been advocated for early learning of Java because it abstracts the bootstrapping mechanism of *main()*, allowing the student to concentrate on the use of objects early.

TDD, however, is still far from being widely accepted in academia. Faculty who don't specialize in software engineering are still unlikely to have much familiarity with TDD. Instructional materials on TDD targeted at undergraduate courses are basically non-existent. As we will discuss in section five, several steps need to take place before TDD finds its place in the undergraduate curriculum.

2.6 Recent Context of TDD

Test-driven development has emerged in the context of agile methods. This section notes the significance of agile methods and considers attempts to measure how many development groups are applying agile methods.

2.6.1 Emergence of Agile Methods

The early years of the twenty-first century have seen significant attention given to what are deemed *agile* methods. Agile methods clearly have roots in the incremental, iterative, and evolutionary methods discussed earlier. Abrahamsson et al. [4] provide an evolutionary map of nine agile methods, and describe such methods as focusing primarily on simplicity and speed, emphasizing people over processes [3].

Extreme Programming (XP) [15] is probably the most well-known agile method, and in fact XP is often used in combination with other agile methods such as Scrum. XP proposes the use of TDD as an integral component of developing high-quality software. There is an interesting conflict between the highly disciplined practice of TDD and the simple, lightweight nature of agile processes. In fact, one of the primary concerns of potential adopters of TDD seems to be the overhead or cost/time of writing and maintaining the unit tests. Although he concedes that automated unit tests are not necessary for absolutely everything (some things are still hard to automatically test), Beck insists that TDD is necessary for XP to work. It seems that TDD may provide the "glue" that holds the process together.

2.6.2 Measuring Adoption of Agile Methods

It is hard to measure the use of a particular software development methodology. Many organizations may be using the methodology, but not talking about it. Others might claim to be using a methodology, when in reality they may be mis-applying the methodology, or worse yet, advertising its use falsely. Surveys might be conducted to gauge a methods use, but often only those who are enthusiastic about the methodology (either in favor or opposed) will respond.

A 2002 survey [65] reported that out of 32 survey respondents across ten industry segments, fourteen firms were using an agile process. Of these, five of the firms were categorized in the E-business industry. Most of the projects using agile processes were small (ten or fewer participants) and lasting one year or less. Another 2003 survey [68] reported 131 respondents claiming they were using an agile method. Of these, 59% claimed to be using XP, implying that they were using TDD. Both surveys revealed positive results from applying agile methods with increases in productivity and quality, and reduced or minimal changes in costs.

A substantial body of literature regarding XP has accumulated since its inception. Most of this literature admittedly involves promotion of XP or explanations of how to implement XP. Many experience reports present only anecdotal evidence of benefits and drawbacks of XP. However, their existence indicates that XP is being adopted in many organizations. It is not clear yet if these same organizations will continue to use XP over time, or if they have or will move on to other (or old) methods.

We are unaware of any measure of how widespread is the use of TDD. The popularity of XP, however, seems to imply a growing adoption of TDD. It is possible

that organizations are adopting XP without adopting all of the practices, or they are applying some practices inconsistently. Rasmusson reports on a project at ThoughtWorks, an early adopter of XP, in which he estimates that one-third of the code was developed using TDD [64]. In the same report, though, he states,

> If I could only recommend one coding practice to software developers, those who use XP or otherwise, it would be to write unit tests.

In this ThoughtWorks project, 16,000 lines of automated unit tests were written for 21,000 lines of production code. It appears that many tests were written in both a test-first and test-last manner.

Despite the possibility of adopting XP without TDD, TDD seems to be a core practice in XP and anecdotal evidence seems to indicate that TDD is commonly included when only a subset of XP is adopted.

Another possible indicator of the use of TDD is the use of the xUnit testing frameworks. JUnit was the first such framework and it has enjoyed widespread popularity. As Martin Fowler stated regarding JUnit,

> Never in the field of software development was so much owed by so many to so few lines of code [31].

No adoption statistics are directly available for JUnit. However, JUnit is included in the core distribution of Eclipse, a popular integrated development environment which is primarily used for Java development. A February, 2004 press release [23] states that the Eclipse platform has recorded more than 18 million download requests since its inception. Although duplicate requests likely occur from the same developer requesting new releases, the figure is still substantial. Certainly not all Eclipse developers are using JUnit, nor are all JUnit adopters using TDD, but it seems likely that the combination of XP, JUnit, and Eclipse popularity implies some degree of TDD adoption.

3

Related Work

Since the introduction of XP, many practitioner articles and several books [12,17,51] have been written describing how to apply TDD. Relatively little evaluative research, however has been published on the benefits and effects of TDD.

The sections below will summarize and classify the research discovered to date that specifically evaluates TDD. There are a number of publications on XP and agile methods, many anecdotal and some empirical. However, this discussion will exclude research on XP or agile methods as a whole. Such research might prove informative when examining TDD, but it fails to prove any individual merits or shortcomings of TDD.

Research on TDD can be categorized broadly by context. In particular, TDD research will be classified as "Industry" if the study or research was primarily conducted with professional software practitioners. Alternatively, the research will be classified as "Academia" if the software practitioners are primarily students and the work is in the context of a course or some academic setting. Studies in which students work on a project for a company but as the requirements and in the context of some course will be classified with "Academia".

3.1 Evaluative Research on TDD in Industry

A very limited number of evaluative research studies have been conducted on TDD with professional practitioners. North Carolina State University (NCSU) seems to be the only source of such studies to date. Researchers at NCSU have performed at least three empirical studies on TDD in industry settings involving fairly small groups in at least four different companies [34, 52, 74]. These studies primarily examined defect density as a measure of software quality, although some survey data indicated that programmers thought TDD promoted simpler designs. In the George study, programmer experience with TDD varied from novice to expert, while

Study	Type	No. of Companies	No. of Programmers	Quality Effects	Productivity Effects
George [34]	CE	3	24	TDD passed 18% more tests	TDD took 16% longer
Maximilien [52]	CS	1	9	50% reduction in defect density	minimal impact
Williams [74]	CS	1	9	40% reduction in defect density	no change

Table 3.2: Summary of TDD Research in Industry

the other studies involved programmers new to TDD.

These studies revealed that programmers using TDD produced code which passed between 18% and 50% more external test cases than code produced by the corresponding control groups. The studies also reported less time spent debugging code developed with TDD. Further they reported that applying TDD had from minimal impact to a 16% decrease in programmer productivity. In other words, applying TDD sometimes took longer than not using TDD. In the case of that took 16% more time, it was noted that the control group also wrote far fewer tests than the TDD group.

These studies are summarized in Table 3.2. Each experiment is labeled as either a case study (CS) or a controlled experiment (CE).

3.2 Evaluative Research on TDD in Academia

A number of studies are reported from academic settings. Most of these examine XP as a whole, but a few specifically focus on TDD. Although many of the publications on TDD in academic settings are primarily anecdotal [13, 56], five were discovered which report empirical results. When referring to software quality, all but one [46] of the empirical studies focused on the ability of TDD to detect defects early. Two [25, 46] of the five studies reported significant improvements in software quality and programmer productivity. One [26] reported a correlation between number of tests written and productivity. In this study, students using test-first wrote more tests and were significantly more productive. The remaining two [57, 60] reported no significant improvements in either defect density or productivity. All five studies were relatively small and involved only a single semester or less. In all studies, programmers had little or no previous experience with TDD.

Although not included here, the anecdotal studies are also beneficial to examine. For instance, the Barriocanal study reports that only 10% of the 100 students involved

Study	Type	No. of Programmers	Quality Effects	Productivity Effects
Edwards [25]	CE	59	54% fewer defects	n/a
Kaufmann [46]	CE	8	improved information flow	50% improvement
Müller [57]	CE	19	no change, but better reuse	no change
Pančur [60]	CE	38	no change	no change
Erdogmus [26]	CE	35	no change	improved productivity

Table 3.3: Summary of TDD Research in Academia

actually wrote unit tests, indicating that motivation is a serious concern.

The empirical studies are summarized in Table 3.3. All studies involved controlled experiments (CE).

3.3 Research Classification

Vessey et al. [36, 70] present a classification system for the computing disciplines. This system provides classification of research by topic, approach, method, reference discipline, and level of analysis. The previously mentioned studies are summarized in Table 3.4. This table applies the Vessey classification system, and the table contents are described in the following sections. This table summarizes all experimental studies found, plus two anecdotal studies. A number of additional anecdotal studies were discovered. Although some of these do have useful information as mentioned in the previous section, they reveal little concerning classification and thus are not included here.

3.3.1 Definition of "Topic" Attribute

This research concentrates solely on the topic of TDD which fits in category 3.0 Systems/Software Concepts and subcategory 3.4 Methods/techniques.

3.3.2 Definition of "Approach" Attribute

Research approaches may be descriptive, evaluative, or formulative. A number of publications were referenced in previous sections which originally presented and explained TDD. These would be considered formulative and descriptive research. This research focuses primarily on evaluative research of the TDD software method. Such research attempts to evaluate or assess the efficacy of TDD.

Evaluative approaches may be divided into the following four sub-categories: deductive (ED), interpretive (EI), critical (EC), or other (EO). From Table 3.4 one can see all of these studies are classified as evaluative-deductive.

3.3.3 Definition of "Method" Attribute

Nineteen research methods are proposed ranging from Conceptual Analysis through Simulation. The research under consideration was determined to use either Case Study (CS), Laboratory Experiment - Human Study (LH), or Field Study (FS).

3.3.4 Definition of "Reference Discipline" Attribute

Research bases its theories on other disciplines. In the case of the computing disciplines, computer science and particularly software engineering have been found to overwhelmingly be self-referential. In other words, most computing research is based on other computing research, and it borrows little from other disciplines such as Cognitive Psychology, Science, Management, or Mathematics. This trend is true with TDD as well as all of the research under consideration is considered to be Self-Reference (SR).

3.3.5 Definition of "Level of Analysis" Attribute

The final area of classification deals with the "object on which the research study focused." [36] These objects determine the level of analysis which is almost the granularity of the object. Levels are grouped into technical and behavioral levels. These studies focused on Project (PR), Group/Team (GP), or Individual (IN) which are all behavioral levels. It might be argued that the research also focused on the technical levels of Abstract Concept (AC) because we are looking at software quality, and Computing Element (CE) because we are looking at unit tests.

3.4 Factors in Software Practice Adoption

A variety of factors play into the widespread adoption of a software practice. Motivation for change, economics, availability of tools, training and instructional mate-

Study	Context	Approach	Method	Reference Discipline	Level of Analysis
George [34]	Industry	ED	LH	SR	GP
Maximilien [52]	Industry	ED	FS	SR	PR
Williams [74]	Industry	ED	FS	SR	PR
Barriocanal [13]	Academia	ED	CS	SR	IN
Mugridge [56]	Academia	ED	CS	SR	GP
Edwards [24]	Academia	ED	LH	SR	IN
Kaufmann [46]	Academia	ED	LH	SR	IN
Müller [57]	Academia	ED	LH	SR	IN
Pančur [60]	Academia	ED	LH	SR	IN
Erdogmus [26]	Academia	ED	LH	SR	IN

Table 3.4: Classification of TDD Research

rials, a sound theoretical basis, empirical and anecdotal evidence of success, time, and even endorsements of the practice by highly regarded individuals or groups can all influence the decision on whether or not to adopt a new practice.

The current state of TDD is mixed regarding this list of factors. With regard to some factors, TDD seems to be poised for growth in adoption. The state of software development practice provides a clear motivation for change. Software development is a complex mix of people, process, technology, and tools which continues to struggle to find consistency and predictability. Projects continue to run over schedule and budget, and practitioners seem eager to find improved methods.

As was noted in earlier sections, tools such as JUnit, MockObjects, and Cactus are mature and widely available. Although much of the tool development has targeted the Java Programming Language, Java is an increasingly popular language both in commercial applications and academia. Further, tool support for TDD is good and improving for most modern languages.

Economic models have considered XP and TDD [58] and note the potential for positive improvements, but recognize that additional research is needed. As was seen in the previous section, empirical and anecdotal evidence is still quite sparse, and limited to fairly small, disparate studies. This research will extend the examination of TDD extensively first by looking at software quality more broadly, and second by looking at a much larger, more diverse population over a longer period of time.

The interplay of acceptance between academics and industry practitioners is a very interesting one. Some reports indicate that it takes five to fifteen years for research developments to make it into commercial practice. The reverse pathway seems to be similar. Some research has shown how TDD can improve programming pedagogy, yet there are few instructional resources available. JUnit incorporation

into BlueJ and the corresponding programming textbook indicates that improvements may be on the way in this area.

There are a number of challenges to adopting TDD. Perhaps first and foremost is that TDD requires a good deal of discipline on the part of the programmer. Hence programmers may require compelling reasons before they are willing to give it a try. Secondly, TDD is still widely misunderstood. Perhaps its name is to blame, but many still erroneously think that TDD is only about testing, not design. Third, TDD doesn't appear to fit in every situation. Section three described iterative, incremental, and evolutionary process models which work best with TDD. Developers and managers must then determine when to apply TDD and when to do something else.

It is not clear how widespread TDD will be adopted. Additional research and the availability of training and instructional materials may play an important role. Such work is the topic of the next section.

4

Research Methodology

This chapter presents the test-driven development approach and details how this research will examine it. In the first section, TDD will be introduced with a small sample application, giving an examples in Java. Particular attention will be given to how TDD informs design decisions.

Next, test-driven learning (TDL) will be introduced with an example of how it might be incorporated into the undergraduate curriculum.

Finally, the design of the formal experiment will be detailed.

4.1 TDD Example

This section will present an example of how developing an application with TDD might proceed. The application to be developed is a television channel guide as described by the use cases in Figure 4.1. We will only start the application assuming that there is only one channel and the user can only move left and right. In other words, we will not attempt the use case "Shift Channel Selection Up/Down".

In the Java implementation, the application should provide a graphical user interface that displays a window of maybe three hours worth of shows. It allows the user to select a show and scroll the window of shows to the left and right with the arrow keys.

The C++ implementation will provide a character-based user interface and allow the user to move left and right by entering 4 and 5 respectively. Screen shots of possible Java and C++ implementations are given in Figure 4.2 and Figure 4.3.

4.1.1 Java Example

First we will do a Java example. As discussed in chapter 2, JUnit is the de facto standard testing framework for Java so our example will use JUnit and TDD to develop

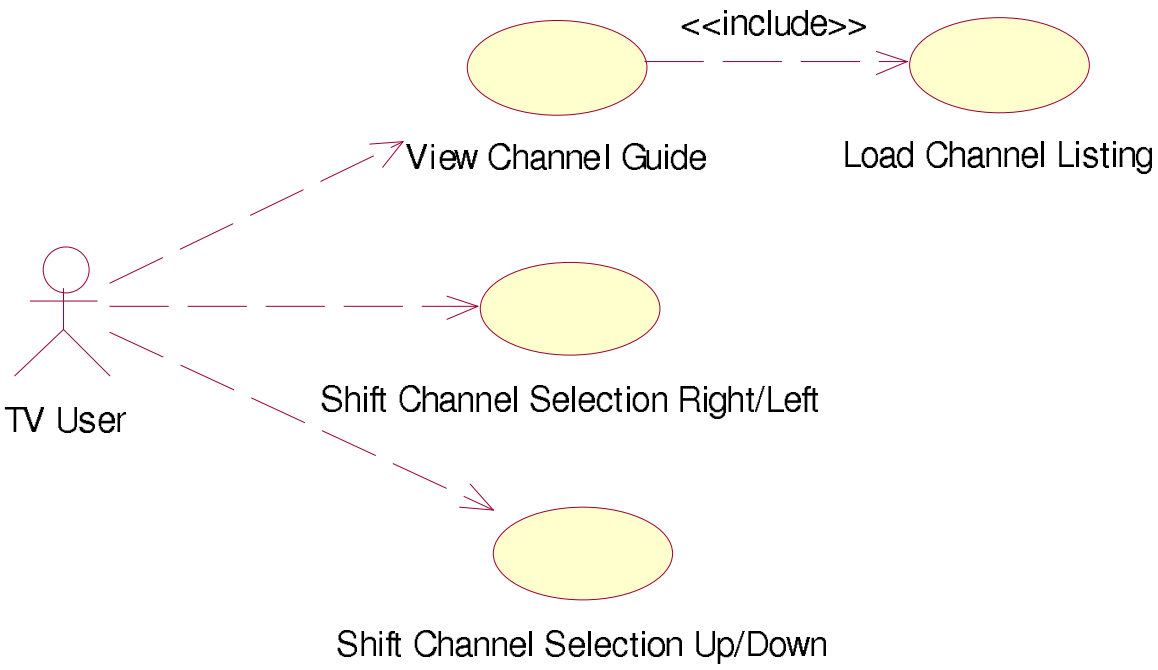

Figure 4.1: Television Channel Guide Use Cases

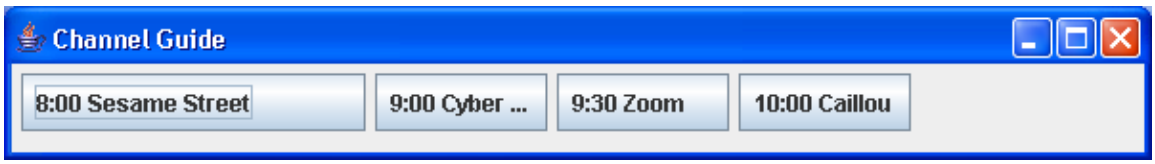

Figure 4.2: Television Channel Guide Java GUI

```
8:00            9:00            10:00   10:30
+==============++==============++======++======+
|Sesame Street ||Cyber Chase   ||Zoom  ||Arthur|
+==============++==============++======++======+
Enter 4 to move left one show, 5 to move right one show, and -1 to quit.
```

Figure 4.3: Television Channel Guide C++ Screen Shot

```
import junit.framework.TestCase;
public class TestShow extends TestCase {
  public void testShowConstructor() {
    Show oneShow = new Show("Sesame Street",60,8,0);
    assertEquals(oneShow.getTitle(),"Sesame Street");
    assertEquals(oneShow.getDuration(),60);
    assertEquals(oneShow.getStartHr(),8);
    assertEquals(oneShow.getStartMins(),0);
  }
}
```

Figure 4.4: Testing Show in Java

this application. To get started, the first test might be to instantiate a television show and access appropriate members. In so doing, we have identified that Show is a likely object and we must specify the interface for inserting and retrieving members. The first test might look something like the code listed in Figure 4.4.

Immediately we see the structure of a JUnit test. We gain access to the JUnit package through the import statement. Then we create a subclass of *TestCase* and write methods that begin with "test". Tests are executed with the *assertEquals()* method. We will see that there are a number of *assertXXX()* methods available to us in JUnit.

At this point, our program will not even compile because the *Show* class has not been written. Because we have only specified very simple methods to this point, we can go ahead and implement the constructors and four accessor methods, then run JUnit to see if they all pass the test. We would not implement multiple methods at once with TDD, except when they are as trivial as these. A screen shot of JUnit after all tests completed successfully is given in Figure 4.5. At this point the code for Show might look like that in Figure 4.6.

Once the *Show* class has been implemented and the test passes, we might write another test to see how *Show* handles bad input. We might specify in the test that we want *Show* to throw an exception if the duration, start hour, or start minutes is out of range. Exceptional behavior can be difficult to test with integration and functional tests, but JUnit enables simple exception testing. The JUnit approach is as follows:

- Force an exception to be thrown
- Follow with a fail statement to detect if the exception is not thrown
- Catch the exception and assert that it was caught

The test in Figure 4.7 specifies that the constructor should throw the exception. Notice the use of the fail method following the line that is expected to throw the

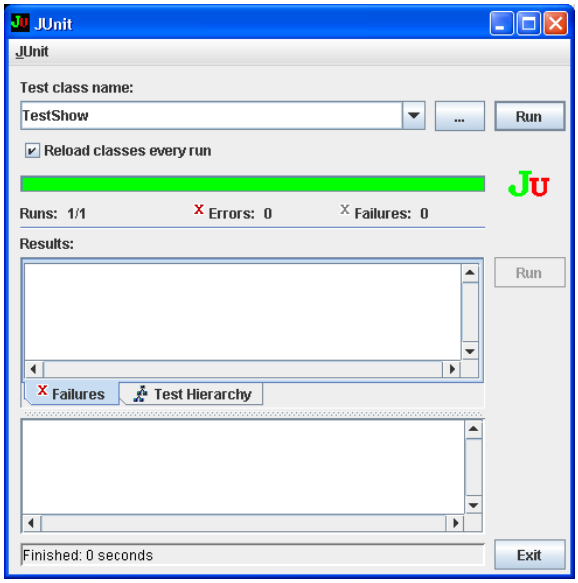

Figure 4.5: JUnit GUI - All Tests Pass

exception. This technique ensures that the exception was thrown and execution did not reach the fail method. In the exception handler, the assertTrue method may be unnecessary, but it provides documentation that execution should reach this point.

Because we have not yet implemented this functionality, this test will fail as shown in Figure 4.8. We would now proceed to implement the desired exception throwing and check to see if we need to refactor to improve either the code or the tests. We would continue to repeatedly write a test, write the code to make the test pass, and refactor until we are satisfied that the *Show* class has the interface and behavior that we desire.

Next we consider whether the *Show* class was the correct place to start. It was the first thing that came to mind, but maybe we were thinking at too low a level. After reviewing the use cases, we might decide to tackle the "Load Channel Listing" use case. We might start with the test shown in Figure 4.9.

In this test we have defined the file format, identified the *ChannelGuide* class, and specified a constructor that accepts the name of the file containing the television show listings. We might step back and consider how the test drove us to make the filename a parameter to this class. Had we been designing with a UML class diagram, we likely would have included a filename member in this class, but we may not have considered passing the name as a constructor parameter. Because we are thinking of how to use and test the class from the beginning, the class is naturally more testable.

As development progresses one might notice the emphasis placed on the underlying model of the application. Because the graphical user interface is difficult to test automatically, TDD encourages placing as much functionality as possible in the model, minimizing what will exist in the GUI. We will conclude this example by

```
public class Show {
  public Show() {}
  public Show(String title, int hr, int min, int duration){
    this.title = title;
    this.duration = duration;
    this.startHr = hr;
    this.startMins = min;
  }
  public String getTitle() {
    return title;
  }
  public int getDuration() {
    return duration;
  }
  public int getStartHr() {
    return startHr;
  }
  public int getStartMins() {
    return startMins;
  }
  private String title;
  private int duration;
  private int startHr;
  private int startMins;
}
```

Figure 4.6: Java Show Class

```
public void testBadMins() {
  try {
    Show oneShow = new Show("Cyber Chase",30,7,70);
    fail("Non-default constructor should throw an Exception if the\n"
        + " minutes parameter is greater than 59 or less than 0");
  }
  catch (Exception expected) {
    assertTrue(true);
  }
}
```

Figure 4.7: Testing Java Exceptions

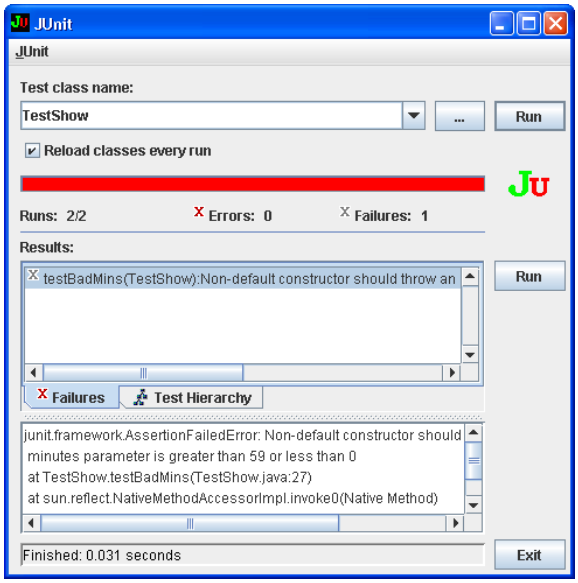

Figure 4.8: JUnit Exception Failure

```
public void testChannelGuideFromFile() {
  try {
    PrintWriter dout = new PrintWriter(
                        new FileWriter("tvlistings.txt"));
    dout.println("Sesame Street:8:0:60");
    dout.println("Cyber Chase:9:0:30");
    dout.println("Zoom:9:30:30");
    dout.println("Caillou:10:0:30");
    dout.println("Mr. Rogers:10:30:30");
    dout.println("Zooboomafoo:11:0:30");
    dout.println("Arthur:11:30:30");
    dout.close();
  } catch(IOException e) { System.out.println(e);}

  ChannelGuide cg = new ChannelGuide("tvlistings.txt");
  assertEquals(cg.numShows(),7);
}
```

Figure 4.9: JUnit Test

looking at some of the event handling code in the GUI.

The GUI needs to react to two types of events: pressing the right arrow key should shift the television listing one show to the right, and pressing the left arrow key should shift the listing one show to the left. Prior to even writing the GUI code, we can write tests for the event handlers. The code in Figure 4.10 and Figure 4.11 utilizes the setUp() method to create a test file prior to each test. The first test called *testMoveRight()* creates the GUI with the specified file, then checks to see if the first show is the first one about to be displayed by the GUI ("Sesame Street"). Next the test forces the action of pressing the right arrow key to be performed by extracting the *MoveRightAction* object from the GUI and performing the action. Finally the test checks to see if the new first show is what used to be the second show ("Cyber Chase").

The code under test is given in Figure 4.12 and Figure 4.13 along with the event handling code in Figure 4.14. Notice that no GUI components are tested directly. The *ChannelGuideGUI* object is a *JFrame*, but it is instantiated and tested without actually showing it. We do observe some improvements that could be made. For instance, the *MoveRightAction* and the *MoveLeftAction* classes are so similar that they could probably be combined, perhaps in a common parent that implements the Template Method [32] design pattern. The tests give us courage to refactor to such a pattern. We can make small, incremental changes such as changing a class name, adding a method parameter, or eliminating a class, using the tests to quickly determine if we have broken anything.

4.1.2 C++ Example

Next we will do a C++ example. Unlike with Java, there is no de facto standard unit testing framework for C++. There may be a number of reasons for this [69], not least of which is the lack of reflection capabilities like that in Java.

In CS2 and above I propose using the CxxTest [71] framework as it seems to have the simplest interface. CxxTest is to be included with the standard libraries. Unfortunately it does require an installation of perl and an extra step in compilation.

To minimize the intrusion to the learning programmer, In CS1 I propose using simple assert statements from the standard library cassert. The example in Figure 4.15 and Figure 4.16 demonstrates a CS1 appropriate implementation where students only know about classes, arrays, and assert. The class declarations, *main()*, and three tests are given in Figure 4.17.

4.2 Test-Driven Learning

This section will introduce a novel approach to teaching programming concepts. Unit tests will be used to present new concepts. Students will then write unit tests

```
public class TestChannelGuideGUI extends TestCase {
  public void setUp() {
    try {
      PrintWriter dout = new PrintWriter(
                          new FileWriter("tvlistings.txt"));
      dout.println("Sesame Street:8:0:60");
      dout.println("Cyber Chase:9:0:30");
      dout.println("Zoom:9:30:30");
      dout.println("Caillou:10:0:30");
      dout.println("Mr. Rogers:10:30:30");
      dout.println("Zooboomafoo:11:0:30");
      dout.println("Arthur:11:30:30");
      dout.close();
    } catch(IOException e) { System.out.println(e);}
  }
  public void testMoveRight() {
    ChannelGuideGUI cgui = new ChannelGuideGUI("tvlistings.txt");
    ListIterator it = cgui.cg.currentStartIterator();
    assertEquals(((Show)it.next()).getTitle(),"Sesame Street");
                                //create move right action
    cgui.showPanel.getActionMap().get("panel.right").
       actionPerformed(new ActionEvent(this, 0, ""));
    it = cgui.cg.currentStartIterator();
                                //verify new start
    assertEquals(((Show)it.next()).getTitle(),"Cyber Chase");
                                //verify button text
    assertEquals(cgui.showButtons[0].getText(),"9:00 Cyber Chase");
  }
  public void testMoveLeft() {
    ChannelGuideGUI cgui = new ChannelGuideGUI("tvlistings.txt");
    ListIterator it = cgui.cg.currentStartIterator();
    assertEquals(((Show)it.next()).getTitle(),"Sesame Street");
                                //create move left action
    cgui.showPanel.getActionMap().get("panel.left").
       actionPerformed(new ActionEvent(this, 0, ""));
    it = cgui.cg.currentStartIterator();
                                //verify start didn't change
    assertEquals(((Show)it.next()).getTitle(),"Sesame Street");
                                //verify button text
    assertEquals(cgui.showButtons[0].getText(),"8:00 Sesame Street");
  } ...
}
```

Figure 4.10: Testing Events in Java GUI

```
public void testMoveLeft2() {
  ChannelGuideGUI cgui = new ChannelGuideGUI("tvlistings.txt");
  ListIterator it = cgui.cg.currentStartIterator();
  assertEquals(((Show)it.next()).getTitle(),"Sesame Street");
  cgui.showPanel.getActionMap().get("panel.right").
     actionPerformed(new ActionEvent(this, 0, ""));
  it = cgui.cg.currentStartIterator();
  assertEquals(((Show)it.next()).getTitle(),"Cyber Chase");
  cgui.showPanel.getActionMap().get("panel.right").
     actionPerformed(new ActionEvent(this, 0, ""));
  it = cgui.cg.currentStartIterator();
  assertEquals(((Show)it.next()).getTitle(),"Zoom");
  cgui.showPanel.getActionMap().get("panel.left").
     actionPerformed(new ActionEvent(this, 0, ""));
  it = cgui.cg.currentStartIterator();
  assertEquals(((Show)it.next()).getTitle(),"Cyber Chase");
}
```

Figure 4.11: Testing Events in Java GUI cont.

to explore these concepts. This approach was inspired by the Explanation Test [17] and Learning Test [17] testing patterns proposed by Kent Beck, Jim Newkirk, and Laurent Bossavit. These patterns were suggested as mechanisms to coerce professional programmers to adopt TDD.

Test-driven learning (TDL) expands significantly on this idea both in its approach and its audience. Novice programmers will be presented with unit tests as examples to demonstrate how programming concepts are implemented. Further, programmers will be taught to utilize automated unit tests to explore new concepts. Typically, novice programmers use some form of direct input and output to test their programs, and relatively little attention is usually given to individual unit testing. TDL replaces input/output statements with automated unit tests.

For example, if a student is learning to write *for* loops in C++, they might be presented with the program in Figure 4.18. Notice how simple assert functions from the standard C library are used, rather than a full-featured testing framework as discussed earlier. This approach minimizes the barriers to introducing unit testing. Of course there are disadvantages to this approach. For instance, if a test/assert fails, no further tests are executed. Also, there is no support for independent tests or test suites. However, because the programs at this level are so small, I think the simplicity of assert statements is the better choice.

To continue the example, in a lab setting, the student might then be asked to write additional unit tests to understand the concept. For instance, they might add

```
public class ChannelGuideGUI extends JFrame {
  public static void main(String [] args) {
    ChannelGuideGUI cgui = new ChannelGuideGUI("tvlistings.txt");
    cgui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    cgui.show();
  }
  public ChannelGuideGUI(String fn) {
    cg = new ChannelGuide(fn);
    setTitle("Channel Guide");
    setSize(WIDTH, HEIGHT);
    showPanel = new JPanel();
    showPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
    showButtons = new JButton[5];
    for(int a=0;a<5;a++) {
      showButtons[a] = new JButton();
      showPanel.add(showButtons[a]);
    }
    Container contentPane = getContentPane();
    contentPane.add(showPanel);
    addActions();
    displayShows();
  }
  public static final int WIDTH = 600;
  public static final int HEIGHT = 80;
  ChannelGuide cg;
  JPanel showPanel;
  JButton [] showButtons;
```

Figure 4.12: Java GUI

```
  private void addActions() {
    InputMap imap =
      showPanel.getInputMap(JComponent.WHEN_IN_FOCUSED_WINDOW);
    imap.put(KeyStroke.getKeyStroke(KeyEvent.VK_RIGHT,0),
                                        ''panel.right'');
    imap.put(KeyStroke.getKeyStroke(KeyEvent.VK_LEFT,0),
                                        ''panel.left'');
    // associate the names with actions
    ActionMap amap = showPanel.getActionMap();
    amap.put("panel.right", new MoveRightAction(cg,this));
    amap.put("panel.left", new MoveLeftAction(cg,this));
  }

  void displayShows() {
    ListIterator i = cg.currentStartIterator();
    for(int a=0;a<5;a++) {
      showButtons[a].setPreferredSize(new Dimension(0,0));
      showButtons[a].setHorizontalAlignment(SwingConstants.LEFT);
      showButtons[a].setMargin(new Insets(5,5,5,5));
    }
    int duration = 0;
    int c=0;
    while(i.hasNext() && duration < 150) {
      Show s = (Show)i.next();
      int mins = s.getStartMins();
      String t = "" + s.getStartHr() + ":";
      if (mins<10)
        t += "0" + mins;
      else
        t += "" + mins;
      t += "␣" + s.getTitle();
      showButtons[c].setText(t);
      showButtons[c].setPreferredSize(
                        new Dimension(s.getDuration()*3,30));
      c++;
      duration += s.getDuration();
    }
    repaint();
  }
}
```

Figure 4.13: Java GUI cont.

```
class MoveRightAction extends AbstractAction {
  MoveRightAction(ChannelGuide cg, ChannelGuideGUI c) {
    this.cg = cg;
    comp = c;
  }
  public void actionPerformed(ActionEvent a){
    cg.advanceOne();
    comp.displayShows();
  }
  private ChannelGuide cg;
  private ChannelGuideGUI comp;
}

class MoveLeftAction extends AbstractAction {
  MoveLeftAction(ChannelGuide cg, ChannelGuideGUI c) {
    this.cg = cg;
    comp = c;
  }
  public void actionPerformed(ActionEvent a){
    cg.backupOne();
    comp.displayShows();
  }
  private ChannelGuide cg;
  private ChannelGuideGUI comp;
}
```

Figure 4.14: Java GUI Event Handling

```
class Show
{
  public:
    Show() : startHrs(0),startMins(0),duration(0)
       { strcpy(title,""); }
    Show(char[],int,int,int);
    Show(istream&);
    void getTitle(char[]);
    int getStartHours();
    int getStartMins();
    int getDuration();
    void displayTimeHeaders(ostream& out);
    void displayTopBottomLine(ostream& out);
    void displayMiddleLine(ostream& out);
  private:
    char title[21];
    int startHrs;
    int startMins;
    int duration;
};

class Listing
{
  public:
    Listing() {}
    Listing(istream&);
    int getNumShows();
    void setCurrent(int,int);
    Show getCurrent();
    Show getNext();
    Show getPrev();
    bool hasNext();
    bool hasPrev();
  private:
    int getShowIndex(int,int);
    Show shows[20];
    int numShows;
    int current;  // index of current show
};
```

Figure 4.15: C++ Channel Guide

```
class ChannelGuide
{
  public:
    ChannelGuide();
    void display();
    void move(int);
  private:
    Listing listing;
};

int main()
{
  run_tests();
  ChannelGuide cg;
  int input=0;
  do
  {
    cg.display();
    cout << "Enter 4 to move left one show, "
         << "5 to move right one show, "
         << "and -1 to quit" << endl;
    cin >> input;
    cg.move(input);
  } while(input>=0);
  return 0;
}
```

Figure 4.16: C++ Channel Guide cont.

```
void run_tests()
{
 { //test 1
  Show showOne("Seinfeld",9,0,60);
  char t[20];
  showOne.getTitle(t);
  assert(strcmp(t,"Seinfeld") == 0);
  assert(showOne.getStartHours() == 9);
  assert(showOne.getStartMins() == 0);
  assert(showOne.getDuration() == 60);
 }
 { //tests with input file
  ofstream out;
  out.open("test2.out");
  assert(!out.fail());
  ifstream in;
  in.open("test2.out");
  assert(!in.fail());
  out << "Arthur 9 0 60" << endl
      << "Barney 10 0 30" << endl
      << "Zoom 10 30 30" << endl;
  Listing channelOne(in);
  { //test 2
   assert(channelOne.getNumShows() == 3);
   char t[20];
   channelOne.setCurrent(10,0);
   Show curShow = channelOne.getCurrent();
   curShow.getTitle(t);
   assert(strcmp(t,"Barney") == 0);
   assert(curShow.getStartHours() == 10);
  }
  { //test 3 tests getting a show already in progress
   channelOne.setCurrent(9,30);
   Show curShow = channelOne.getCurrent();
   char t[20];
   curShow.getTitle(t);
   assert(strcmp(t,"Arthur") == 0);
   assert(curShow.getStartHours() == 9);
   assert(curShow.getStartMins() == 0);
  }
 }
}
```

Figure 4.17: C++ Channel Guide Tests

```
#include <iostream>
#include <cassert>
using namespace std;

int sum(int min, int max);

int main()
{
  assert(sum(3,7)==25);

  cout << "No errors encountered" << endl;
  return 0;
}

// This function sums the integers from min to max inclusive.
// Pre: min < max
// Post: return-value = min + (min+1) + ... + (max-1) + max
int sum(int min, int max)
{
  int sum = 0;
  for(int i=min;i<=max;i++)
  {
    sum += i;
  }
  return sum;
}
```

Figure 4.18: C++ Loop Example

the assert statements

```
assert(sum(-2,2)==0);

assert(sum(-4,-2)==-9);
```

Then they might be asked to write unit tests for a new, unwritten function. In doing so, they will have to design the function signature and implement a function stub. This makes them think about what they are going to do before they actually do it.

Once the programmer ventures beyond the lab into larger programming projects, tests can be isolated into a separate function like that shown in Figure 4.19.

I believe test-driven learning is a powerful pedagogical approach because it accomplishes multiple goals simultaneously. TDL focuses students on design, testing, and behavior early-on. TDL encourages a rigorous, Design by Contract [55]-like approach to learning to program. It could be tailored to either a test-first or test-last approach.

Although the primary focus of this research is on the efficacy of test-driven development, it is possible that test-driven learning will be a powerful side-effect worthy of significant further development, refinement, and study. A separate paper that expands the idea of test-driven learning and documents a small formal experiment is attached in the appendix. This paper was recently submitted for acceptance at the 2005 OOPSLA conference Educator's Symposium.

4.3 Experiment Design

This section will outline the details of the formal experiment. It will discuss the hypothesis, independent and dependent variables, and the methods of making and analyzing observations. Comments on the possibility of conducting a case study for external validity will be presented. The chapter will end with a discussion on methods likely to be used to analyze the experiment data and likely conclusions to be drawn.

4.3.1 Hypothesis

The null hypothesis of this experiment is:

> software constructed using the test-driven development approach will have similar quality at higher cost to develop when compared to software constructed with a traditional test-last approach.

The independent variable is the use of test-driven versus test-last development. The dependent variables are software quality and software cost. Additional dependent variables will be observed such as student performance on related assessments

```
#include <iostream>
#include <cassert>
using namespace std;

int sum(int min, int max);
void runTests();

int main()
{
  runTests();

  return 0;
}

// This function sums the integers from min to max inclusive.
// Pre: min < max
// Post: return-value = min + (min+1) + ... + (max-1) + max
int sum(int min, int max)
{
  int sum = 0;
  for(int i=min;i<=max;i++)
  {
    sum += i;
  }
  return sum;
}

// This function executes all of the unit tests.
void runTests()
{
  assert(sum(3,7)==25);
  assert(sum(-2,2)==0);
  assert(sum(-4,-2)==-9);

  cout << "No errors encountered" << endl;
}
```

Figure 4.19: C++ Loop Example with Tests

and subsequent voluntary usage of TDD. Additional qualitative data will be gathered such as student attitudes toward testing and TDD.

It is expected that the hypothesis will be proven incorrect in the context of larger programming projects. Because small projects such as those developed in early programming courses have relatively little opportunity to vary significantly in design and number of defects, it is expected that test-driven development will have little or no effect at these levels. I conjecture that student discipline, maturity, and ambition are more significant factors than development approach with novice programmers.

4.3.2 Observations and Data Gathering

Undergraduate students from three computer science courses, CS1/CS101 (Computer Programming 1), CS2/CS102 (Computer Programming 2 /Data Structures), and SE/CS391 (Software Engineering) will simultaneously participate in this study. Students in each course will take a pre-experiment aptitude test and will complete a pre-experiment survey on their attitudes toward software testing. The pre-experiment survey will request demographic information so that results can be analyzed for significant differences in women and minority population groups [28]. Students will then be divided into control and study groups within each class, subject to the constraints of the course.

Course-adjusted instructional materials on software testing and test-driven development will be incorporated into the lab sections of these three courses. Both the control and the study groups will complete the same set of labs on testing, writing unit tests, and using automated test frameworks. The study group will complete additional labs on test-driven development. The instructional materials will be reviewed by a set of faculty with experience in CS1, CS2, and SE education and/or test-driven development.

The instructional materials will include incremental lab-based exercises that teach software testing in the context of other course-appropriate topics. Each course (CS1, CS2, SE) will include the same set of instructional module topics, but with course-specific examples. For instance, students in CS1 may be taught how to use automated testing frameworks when first learning about functions, whereas students in CS2 will be learning about automated testing frameworks as they investigate the operations on a stack abstract data type.

Some instructional materials, data gathering, and assessment tools will be pilot-tested in one course prior to the full experiment. The students in the pilot test will not be included in the full experiment the following semester. Several faculty will be asked to provide advice and guidance in the development, administration, and assessment of the study.

Students will then be required to complete two programming assignments. The study group will be asked to use test-driven development techniques while the control group will be asked simply to test their programs with no indication to exactly

when. The assignments will be as large as possible within the constraints of the course and the abilities of the students, and the second assignment will build on or reuse significant parts of the first.

At the beginning of the second project, students will be provided a solution to the first project that includes a full set of automated unit tests. In the second project, students may choose to build on either their own solution, or the solution provided.

Students will be required to submit all of the code that they have completed to date at multiple points throughout the project. This code will be evaluated to determine the degree of testing, the degree of reuse, and the quality of code. Some code will likely not compile, but even this will be a good indicator of the use (or non-use) of test-driven development. At the end of the project all code will again be evaluated for testing, reuse, and code quality, but also it will be subjected to full integration and acceptance tests to determine unit- test quality. Some unit tests will be evaluated for code coverage to complete the examination of unit-test quality. During the coding process, a random sample of students will be observed and interviewed regarding their use of test-driven development.

Students will also be required to track the amount of time they spend on programming projects. Some mechanism will be provided to simplify the collection of effort information. This may be a web-based time tracking tool, or perhaps some kind of automated logging scripts associated with logon/logoff and compilation. Aggregate information may be provided to all students correlating use of tests with software quality and student effort.

At the end of the semester, students will be asked to complete a survey indicating their attitudes toward testing and test-driven development. Student exam and course grades will then be compared to determine if any correlation exists between test-driven development and academic performance.

The following semester or year, a sample of students from both the control and study groups will again be examined to determine the voluntary use of test-driven development in course programming assignments. Students from CS1 will be examined in CS2. Students from CS2 will be examined in the SE course. Students from the SE course will be examined in a subsequent course if a significant enough number of them enroll in a common programming-based course.

Selecting Study Groups

A pre-experiment aptitude test will be conducted to inform the selection of the study and control groups. Using results from the aptitude test, the student population will be grouped into three tiers, with random selection separating the three tiers into the control group and the study group. Some boundaries of the course structure may also influence the composition of the two groups.

Ratio of Unit-Tests

Code samples will be gathered at multiple points in the development process to determine if tests are being developed along with the production code or as an after thought. Unit-test per class, unit- test per member function, and unit-test per lines-of-code ratios will be calculated for each student. Because the cycle of test, then code is very short (minutes or even seconds), this study will determine if students are actually writing tests immediately before rather than immediately after writing production code through random direct observation.

Testing ratios will be determined by counting test cases in an automated testing framework such as cppunit or cppunit lite [54]. Code samples will be examined to determine whether students properly utilized the testing framework, or implemented informal tests. Efforts will be made to count all unit-tests even if they do not conform to testing specifications or are not written using the testing framework.

Software Metrics

Project submissions will be evaluated by a set of dynamic and static software metrics. Defect density will be measured through a set of dynamic black-box acceptance tests.

Software quality will be measured by calculating a set of static metrics. To determine software quality, code samples will be examined with currently available software metrics tools such as CCCC (C and C++ Code Counter) [49]. Traditional and object-oriented metrics will be examined including code size, McCabe's Cyclomatic Complexity [53], and particularly fan-out (i.e. number of other modules the current module uses), fan-in (number of other modules which use the current module), and the Information Flow measure suggested by Henry and Kafura [41], which combines these to give a measure of coupling for the module.

Reuse will be measured statically. Many reuse metrics focus on reuse through inheritance. Although this will be examined, I do not anticipate a significant degree of reuse through inheritance especially in the CS1 and CS2 courses. Software will be evaluated for methods and classes reused with and without modification from one project to the next. If possible, such metrics will be calculated from one version to the next in the same project. This will help determine the degree to which the software evolves and the software's stability.

Student Attitude Survey

Student attitudes towards testing and test-driven development will be evaluated through pre- and post- experiment surveys conducted with both the control and the study groups. In the pre-experiment survey, students will be asked to report on how they perceive the value of testing, how they currently test their programs, if they know what test-driven development is, and how open they are to learning to use test-driven development. In the post-experiment survey, students will be asked

to report again on how they perceive the value of testing, whether they feel like they understand test-driven development, whether they used test-driven development in their assignments, and whether they intend to use the test-driven development approach in the future both in course work, or in any professional programming they may do. Results of the control and study groups will be compared and interpreted in the context of other results.

Subsequent Voluntary Use of Test-Driven Development

A key indicator in whether students agree with the merits of test-driven development is whether they choose to use it. It is anticipated that many students, even if they see significant value in the test-driven development approach, will choose not to use it on course assignments because they do not foresee having to maintain or reuse these assignments. Although students will hopefully see benefits to using test-driven-development in the short-term, in our experience, students will most often take the shortest path to completing an assignment. The shortest path typically involves minimal testing. The value of test-driven development will probably best be realized in long-term projects that will entail future enhancements and maintenance. Nevertheless, programming samples will be collected from both the control and the study groups for programming projects completed after the initial two study projects. These samples will be evaluated to again determine use of automated software tests and software quality.

Student Performance Evaluation

Both project, exam, and overall course grades will be examined to determine if any significant differences exist between the control and the study groups.

4.3.3 Assessment and Internal Validity

Data collected from the experiments will be reported and analyzed statistically. Appropriate graphs such as box plots will be produced to report the data. Statistical tests such as the two-sample t-test will be employed to determine if differences between the control and experimental groups are statistically significant.

Student performance will only be reported in aggregate. In fact, training and approval for the experiments will need to be obtained from the University of Kansas Human Subjects Committee - Lawrence Campus (HSCL) prior to conducting the studies.

A sample analysis of a small experiment conducted on the test-driven learning approach is documented in the paper attached in Appendix A. The lab materials and assessment instruments from this study are also included in Appendix B. These can

serve as an example of how TDL can be integrated into the current curriculum, and how this experiment can be conducted and analyzed, albeit on a much larger scale.

The experiment design and corresponding results should establish internal validity of the experiments. As mentioned earlier, care will be taken to ensure that the control and experimental groups are homogeneous and random. Both groups will be presented with the same instructional material to ensure that no bias is introduced. External validity of the experiments will be also be examined. External validity will be addressed in the last chapter. In particular, though, peer reviewed publications and a case study or small experiment with professional programmers will be conducted to enforce the claims of external validity.

5

Research Plan

The previous chapter summarized the research to be conducted. This chapter outlines a proposed schedule for completing the research and identifies particular challenges and risks anticipated.

5.1 Schedule of Research Activities

The following schedule, shown in Tables 5.5–5.7, is proposed for the completion of major milestones in this research.

5.2 Challenges to Successful Completion

A number of challenges are anticipated with this research. This section describes these challenges and suggests strategies for meeting them. It categorizes the challenges as organizational, technical, motivational, and temporal.

5.2.1 Organizational Challenges

Empirical software engineering requires the cooperation of a number of people. Unlike much computing research which may require hardware and labs but involves only a few people, empirical studies require a population to be studied and the approval to conduct the study.

This research proposes to conduct experiments in approximately seven courses (three in SE, two in CS2, one in CS1, one in a later senior course). Faculty approval and cooperation will be required in all courses. Faculty and Graduate Teaching Assistants will be asked to participate in the studies by presenting new materials, by aiding in

Table 5.5: Remaining Period in Academic Year 2004–2005

Term	Activities
Spring 2005	• Develop and pilot test-driven development lab exercises in a course with students who will not participate in full study • Plan and schedule experiments for Summer and Fall 2005 courses • Adapt and implement data gathering and assessment tools • Complete HSCL training and apply for HSCL approval • Collaborate with reviewers
Summer 2005	• Refine instructional and assessment materials • Conduct pilot controlled experiment in SE

Table 5.6: Academic Year 2005–2006

Term	Activities
Fall 2005	• Conduct controlled experiments in undergraduate courses: CS1, CS2, SE – Conduct pre-learning attitude and aptitude surveys – Deliver test-driven development instruction – Collect code samples – Complete initial assessment results and disseminate to students – Conduct post-learning attitude and aptitude surveys • Analyze and disseminate early results
Spring 2006	• Conduct longitudinal study in undergraduate courses: CS2, SE, one other course – Conduct attitude surveys – Track voluntary usage of test-driven development – Collect code samples – Complete initial assessment results and disseminate to students • Revise and improve instructional and assessment materials if needed
Summer 2006	• Analyze results • Prepare reports • Prepare instructional and assessment materials for dissemination

Table 5.7: Academic Year 2006–2007

Term	Activities
Fall 2006	• Submit reports for publication • Possibly refine and publish instructional and assessment materials
Spring 2007	• Complete dissertation • Present final oral exam

creating and differentiating the control and experiment groups, and providing access to appropriate artifacts such as program submissions and exam scores.

Students will be asked to cooperate by applying the test-first or test-last methods in labs and on programming projects. Students will also be asked to complete surveys at the beginning and end of their courses. In order to track students over two semesters, enrollment information will be needed from faculty or department personnel. Approval will also be needed from the Kansas University Human Subjects Committee - Lawrence Campus (HSCL).

A number of steps have already been taken to approach these challenges. David Hann, the HSCL administrator, has been contacted and fast track approval has already been granted. The process involves an on-line training course, paperwork, and about a one-week approval time.

As is documented in the appendix, I have recently conducted a short formal experiment on test-driven learning in "EECS 138 Computer Programming - C++". This pilot experiment has helped to clarify the types of instructional materials needed, the cooperation needed from instructors, the types of artifacts and observations involved, and the kind of analysis that can be performed.

Dr. Saiedian will be teaching "EECS 448 Software Engineering" this summer and he has offered to allow me to conduct a first experiment there. We have also applied for a grant from the university General Research Fund to support this work over the summer.

I am hopeful that faculty in the department will be receptive to conducting this research in their courses. I will attempt to isolate the new material primarily to labs, and I am hopeful that if I work closely with the professors and I do all the work of developing the lab materials, there will be no objections. If possible, I hope to serve

as a teaching assistant in the CS1 and CS2 courses both semesters.

If possible, a case study or small formal experiment will also be conducted with professional programmers. I have already gained approval from Engenio Information Systems, Inc. (formerly LSI Logic Storage Systems) in Wichita, Kansas to conduct such a study. Engenio has a software development group of around two hundred people, with applications developed in embedded real-time C/C++ systems with Java user interfaces. I have a good relationship with Engenio after providing software training courses to them for the past five years. In general Engenio has not attempted test-driven development, however one of their satellite groups in Tucson, AZ has experimented with it.

Alternatively, it may be possible to conduct a small experiment with students from one or several of the software engineering courses offered at the Edwards Campus. Many of these students work as professional programmers and may be open to participating in such an experiment.

5.2.2 Technical Challenges

A couple of technical challenges are present with this research. The primary one is the introduction of the xUnit testing frameworks. JUnit is widely used, fairly simple to install, and I have a good deal of experience with it. However, CS1 and CS2 are currently taught in C++ at the University of Kansas. As described earlier, assert statements from the standard cassert library can be used to some extent, but a more mature framework such as CxxTest might be preferred in CS2. CxxTest can be installed as a standard library on departmental servers, but if students opt to develop programs elsewhere, they will need to complete the installation. It is unlikely that students will be willing to do this unless it is absolutely required.

The second technical challenge identified is in gather artifacts. Students will be asked to submit code and tests at regular intervals. They will also be asked to track the time they spend on the projects. A variety of mechanisms could be created to aid in this area. Login and logout scripts could be provided that record when students are on the department servers. However we can't be sure they are working on this course or even this project. Plus some students will choose to work on other machines. A better option may be to provide a web application that facilitates code submissions and that provides a time tracking tool for students to record the time they spend on the project.

A third technical challenge involves selecting and interpreting metrics. There are a wide range of potential metrics and tools for calculating such metrics. Cost and language support may limit tool options. Some metrics were mentioned earlier as important measures of software quality, but there is little consensus in the industry regarding quality metrics. A variety of metrics will be calculated and compared, but there may be some disagreement on which metrics are most valuable.

5.2.3 Motivational Challenges

Perhaps the most significant challenge may be motivating students to write tests. Students often do not see the long-term benefits of developing quality software because they know they are unlikely to have to enhance or maintain the software they develop in courses. Not only this, but when students are asked to write tests for their code, they may see this as unnecessary and unreasonably time-consuming. Test-driven development is a more disciplined approach to programming. Such discipline may be unreasonable to expect from novice programmers.

There seem to be a couple of approaches that might overcome these motivational challenges. The first obvious approach is to make tests a part of student grades. Stephen Edwards [24] reports on his experience with determining grades on programming projects as the product of test coverage, tests passed, and acceptance test pass rates. By multiplying the three, in order to get a high score students must not only write a solution that passes the instructors tests, but they must write tests that cover their entire code.

Since grading criteria is a faculty decision and thus cannot be guaranteed. Another option is to find ways to make testing so compelling that students are convinced of its merits. It is unlikely that students will be so compelled at the beginning of a course, but as the course progresses, they may become convinced. In particular if testing is modeled well in the classroom, labs and examples as described in the paper on test-driven learning, then students may be more likely to write their own tests.

Another strategy is to give two projects where the second one builds on the first. At the end of the first project, students could be provided with a solution with tests. On the second project the students may recognize how the tests help them be catching any defects they inject quickly. Plus by measuring student effort and success on subsequent projects, we may be able to report that students who use tests spend less time and have better success than those who don't. This information may be too late for changes in the first semester, but may influence voluntary usage of testing in the second semester.

5.2.4 Temporal Challenges

The final set of challenges deals with time constraints. Test-driven development is a practice that likely develops over time. Novice programmers may struggle to know how to write good tests. Short projects completed individually may not provide compelling motivation for writing tests. The longitudinal aspect of this study will help to measure student testing ability. Maturity will also be an interesting factor.

On a much smaller scale, even when tests are written, it will be very hard to measure *when* the tests were actually written (just before or just after the production code). By looking at the code submissions during the project development, test to

code ratios can help determine if tests were written along with the code or at the very end. Random observations and surveys will be used to get an idea of whether students actually write tests first or last.

Finally there is the challenge of trying to complete studies in all three courses simultaneously and in subsequent semesters. I will be very busy developing instructional materials, then delivering and conducting the studies. I hope to offer my time to teach as many of the TDD-based labs as possible, minimizing the burden on faculty and GTA's, plus providing incentive for them to participate in the studies. Ideally I will be able to serve as a GTA in one lab section of CS1 and one lab section of CS2 each semester which will help keep me involved in the courses.

5.3 Potential Risks

A project of this scale involves a number of risks. The following events could have significant impact on the success of this research:

- professors won't let me conduct experiments in their classes
- system administrators won't install JUnit or CxxTest
- achieving consistent instruction
- achieving unbiased separation into control and experimental groups
- students/programmers don't write tests first
- instructional modules and projects not ready in time
- technical difficulties with automated unit testing tools (e.g. don't support exceptions or templates)
- negative impacts of TDD on student performance prompt early termination of study
- managing lots of data (code, surveys, grade info)
- failure to foresee needed data (survey questions, ...)
- inconsistent code submission tools

Risk avoidance and risk mitigation strategies will be employed to minimize the possibility and impact of these risks. In particular, a number of the most critical risks are already being addressed or will be addressed as early as possible. One advantage is that several of the faculty on my committee will be teaching courses to contain the studies. Hopefully this will improve awareness of the value of the studies and improve the success of the studies.

6

Evaluation, Contributions, and Summary

This final chapter will discuss how this research will be evaluated, and the expected contributions resulting from the research. It will end with a brief summary of the proposal.

6.1 Evaluation and External Validity

External validity involves demonstrating that results discovered in one study can be reproduced elsewhere, and that the results generalize to broader environments. Three approaches will be taken to evaluate this research. First, while the research is still being designed and conducted, reviews will be requested from a number of sources. Computer science faculty and committee members at the University of Kansas will have the opportunity to review the studies in the oral comprehensive exam, as the experiments are being integrated into the courses, and at the end of the first experiment. Additional advice and reviews will be requested from Laurie Williams at North Carolina State University due to her experience with studying pair programming.

The second evaluation of this research will occur with the case study in a professional environment. Similar results in the academic and industry environments will strengthen the results. Differing results will also be valuable, likely leading to new questions exploring environmental, maturity, and possibly product lifetime questions.

The third evaluation of this research will involve peer-review as the results are submitted for publication. Publications and conference presentations will serve as the primary means of disseminating the research results. Reviewer comments and publication acceptance will serve as a meaningful confirmation of the research's validity.

6.2 Expected Contributions

This research should contribute empirical results from the controlled and longitudinal studies to resolve or at least inform five questions:

- Does test-driven development produce higher quality software?
- Can undergraduate students be taught and motivated to use test-driven development?
- Does test-driven development improve immediate student academic performance?
- Are there significant differences in acceptance, use, and effects of test-driven development in the women and minority populations?
- Where is the most appropriate point in the undergraduate programming curriculum to teach test-driven development?

6.2.1 Empirical Evidence of TDD Efficacy

The primary contribution of this research will be empirical evidence of the effects that applying TDD has on software design quality. This research will explore many important quality aspects beyond defect density such as reusability and maintainability.

Quantitative Evidence

Design quality will be measured with a variety of software metrics. Unfortunately there is no common consensus on what constitutes good design. As a result, a number of metrics will be calculated and reported. These metrics will include, but not be limited to:

- fan-in
- fan-out
- information flow
- lines of code per method
- methods per class
- lines of code outside all classes
- cyclomatic complexity

- methods reused without modification
- methods reused with modification
- defect density
- ratio of unit tests to production code

Additional metrics such as development effort (in hours) and student project and exam scores will be calculated.

Qualitative Evidence

A number of qualitative measures will also be examined. Instruments such as pre- and post-experiment surveys will be developed and administered. Questions will cover topics such as:

- attitudes toward testing
- programming experience
- academic performance
- demographics (gender, race, nationality)

6.2.2 Peer-Reviewed Publications

Following each semester, I plan to publish the results of that semester's experiments. These publications will include a statistical analysis of the experimental results. An initial publication on test-driven learning has already been submitted to the OOPSLA Educator's Symposium (see appendix). Future papers may be submitted to *Communications of the ACM*, *IEEE Software*, *IEEE Transactions on Software Engineering*, *Empirical Software Engineering: An International Journal*, and/or the computer science education journal *SIGCSE Bulletin*. I also will target presentations at the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) Educator's Symposium, the International Conference on Software Engineering, the Conference on Software Engineering Education and Training, and/or the SIGCSE annual technical symposium. Additional venues could include professional conferences such as Agile 2006.

6.2.3 Framework for Empirical TDD Studies

A valuable by-product of this research will be a framework for conducting future studies of TDD efficacy. It is unlikely that a single set of studies can explore all aspects of a development approach. Plus as was noted earlier, additional studies

will be necessary to provide external validity. By documenting how this study was conducted and providing instruments, tools, and methods, future studies can be completed more efficiently.

All assessment tools including the aptitude test, pre- and post- experiment attitude surveys, and software quality collection and analysis tools will be made available on the web. I am not aware of a current home for test-driven development education so it is possible that this project will create a wiki-based web site to facilitate the community-driven communication of ideas on test-driven development, particularly in undergraduate education.

6.2.4 Curriculum Materials

In addition, this research will produce instructional materials that incorporate the "test-driven learning" approach into CS1, CS2, and SE courses. These materials could be adapted into other curriculums, or they could be extended extensively, even to the point of producing complete lab or course text books.

6.3 Summary

Despite many significant advances, software construction is still plagued with many failures. Development organizations struggle to make intelligent development method adoption decisions due to a lack of maturity and a general lack of empirical evidence of what methods are best in what contexts. While some individual programmers and organizations have learned to value and apply disciplined, yet flexible methods, students do not generally graduate with these skills.

Test-driven development is a disciplined development practice that promises to improve software design quality while reducing defects with no increased effort. This research proposes to carefully examine the potential of TDD to deliver these benefits. Empirical software engineering methods will be applied in a set of formal controlled longitudinal studies with undergraduate students at the University of Kansas.

If TDD proves to improve software quality at minimal cost, and if this research shows that students can learn TDD from early on, then this research can have a significant impact on the state of software construction. Software development organizations will be convinced to adopt TDD in appropriate situations. New textbooks can be written applying the test-driven learning approach. As students learn to take a more disciplined approach to software development, they will carry this approach into professional software organizations and improve the overall state of software construction.

Bibliography

[1] The CHAOS Report. Technical report, Standish Group International, Inc., 1995.

[2] 2004 third quarter research report. Technical report, Standish Group International, Inc., 2004.

[3] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. Agile software development methods: Review and analysis. Technical Report 478, Espoo, Finland: Technical Research Centre of Finland, 2002.

[4] P. Abrahamsson, J. Warsta, M.T. Siponen, and J. Ronkainen. New directions on agile methods: A comparative analysis. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 244–254, Piscataway, NJ, May 3–10 2003. IEEE Computer Society.

[5] ACM. ACM curricula recommendations volume i: Computing curricula 1991: Report of the ACM/IEEE-cs joint curriculum task force, 1991. http://www.acm.org/education/curricula.html.

[6] ACM. Final report of the joint ACM/IEEE-cs task force on computing curricula 2001 for computer science, 2001. http://www.acm.org/education/curricula.html.

[7] Agile Alliance, 2004. http://www.agilealliance.org.

[8] Steven K. Andrianoff and David B. Levine. Role playing in an object-oriented world. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 121–125. ACM Press, 2002.

[9] Apache. http://www.mockobjects.com.

[10] Apache. http://jakarta.apache.org/cactus/.

[11] Apache. http://incubator.apache.org/derby/.

[12] David Astels. *Test Driven Development: A Practical Guide*. Prentice hall PTR, 2003.

[13] E.G. Barriocanal, M.S. Urb'an, I.A. Cuevas, and P.D. P'erez. An experience in integrating automated unit testing practices in an introductory programming course. *ACM SIGCSE Bulletin*, 34(4):125–128, December 2002.

[14] K. Beck and et al., 2001. http://www.agilemanifesto.org.

[15] Kent Beck. *Extreme Programming Explained*. Addison-Wesley Longman, Inc., 2000.

[16] Kent Beck. Aim, fire. *Software*, 18(5):87–89, Sept.-Oct. 2001.

[17] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[18] Joseph Bergin, Richard Kick, Judith Hromcik, and Kathleen Larson. The object is objects. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 251–251. ACM Press, 2002.

[19] B. Boehm. A spiral model of software development and enhancement. In *Proceedings of the International Workshp on Software Process and Software Environments*. ACM Press, 1985.

[20] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.

[21] Ilene Burnstein. *Practical Software Testing*. Springer-Verlag, 2003.

[22] Henrik Baerbak Christensen. Systematic testing should not be a topic in the computer science curriculum! In *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, pages 7–10. ACM Press, 2003.

[23] Eclipse. http://www.eclipse.org/org/press-release/feb2004foundationpr.html.

[24] S.H. Edwards. Rethinking computer science education from a test-first perspective. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications: Educators' Symposium*, pages 148–155, 2003.

[25] S.H. Edwards. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In *Proceedings of the International Conference on Education and Information Systems: Technologies and Applications EISTA'03*, August 2003.

[26] H. Erdogmus. On the effectiveness of test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(1):1–12, January 2005.

[27] H. Mills et al. Cleanroom software engineering. *Software*, pages 19–25, Sept. 1987.

[28] Allan Fisher and Jane Margolis. Unlocking the clubhouse: the carnegie mellon experience. *SIGCSE Bull.*, 34(2):79–83, 2002.

[29] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[30] Jr. Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[31] E. Gamma and K. Beck. http://www.junit.org.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[33] D. Gelperin and B. Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, 1988.

[34] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337–342, 2004.

[35] T. Gilb. *Software Metrics*. Little, Brown, and Co., 1976.

[36] R.L. Glass, V. Ramesh, and I. Vessey. An analysis of research in computing disciplines. *Communications of the ACM*, 47(6):89–94, June 2004.

[37] J.B. Goodenough and S.L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 2:156–173, June 1975.

[38] Paul Hamill. *Unit Test Frameworks*. O'Reilly, 2004.

[39] Görel Hedin, Lars Bendix, and Boris Magnusson. Introducing software engineering by means of extreme programming. In *Proceedings of the 25th International Conference on Software Engineering*, pages 586–593. IEEE Computer Society, 2003.

[40] Hasko Heinecke and Christian Noack. *Integrating Extreme Programming and Contracts*. Addison-Wesley Professional, 2002.

[41] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.

[42] Watts Humphrey. *Introduction to the Personal Software Process*. Addison-Wesley, 1997.

[43] Sun Microsystems Inc., 2004. http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Class.html

[44] Sun Microsystems Inc., 2005. http://java.sun.com/docs/books/tutorial/uiswing/components

[45] R. Jeffries.

[46] Reid Kaufmann and David Janzen. Implications of test-driven development: a pilot study. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 298–299. ACM Press, 2003.

[47] Michael Kölling and John Rosenberg. Guidelines for teaching object orientation with java. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 33–36. ACM Press, 2001.

[48] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *IEEE Computer*, 36(6):47–56, June 2003.

[49] Tim Littlefair. C and c++ code counter, 2003. http://cccc.sourceforge.net.

[50] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, Inc., 2003.

[51] Vincent Massol and Ted Husted. *JUnit in Action*. Manning, 2004.

[52] E. Michael Maximilien and Laurie Williams. Assessing test-driven development at IBM. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 564–569, Piscataway, NJ, May 3–10 2003. IEEE Computer Society.

[53] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, Dec 1976.

[54] Object Mentor, 2004. http://www.objectmentor.com/resources/downloads/index.

[55] Bertrand Meyer. Applying “Design by Contract". *Computer*, 25(10):40–51, 1992.

[56] R. Mugridge. Challenges in teaching test driven development. In *Proceedings of XP 2003*, pages 410–413, May 2003.

[57] M.M. Müller and O. Hagner. Experiment about test-first programming. *IEEE Proceedings-Software*, 149(5):131–136, 2002.

[58] M.M. Müller and F. Padberg. On the economic evaluation of XP projects. In *Proceedings of ESEC/FSE'03*, pages 168–177, Sept 2003.

[59] James Noble, Stuart Marshall, Stephen Marshall, and Robert Biddle. Less extreme programming. In *Proceedings of the sixth conference on Australian computing education*, pages 217–226. Australian Computer Society, Inc., 2004.

[60] Mataž Pančur, Mojca Ciglarič, Matej Trampuš, and Tone Vidmar. Towards empirical evaluation of test-driven development in a university environment. In *Proceedings of EUROCON 2003. Computer as a Tool. The IEEE Region 8*, volume 2, pages 83–86, 2003.

[61] Andrew Patterson, Michael Kölling, and John Rosenberg. Introducing unit testing with BlueJ. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, pages 11–15. ACM Press, 2003.

[62] R. Poston. *Automating Specification-Based Software Testing*. IEEE Computer Society Press, 1996.

[63] R.S. Pressman. *Software Engineering: A Practitioner's Approach, Sixth Edition*. McGraw Hill, 2005.

[64] Jonathan Rasmusson. Introducing XP into greenfield projects: Lessons learned. *IEEE Software*, 20(3):21–28, 2003.

[65] D.J. Reifer. How good are agile methods? *Software*, 19:16–18, Jul/Aug 2002.

[66] W. Royce. Modeling the development of large software systems. In *Proceedings of Westcon*, pages 328–339. IEEE CS Press, 1970.

[67] Jean-Guy Schneider and Lorraine Johnston. extreme programming at universities: an educational perspective. In *Proceedings of the 25th International Conference on Software Engineering*, pages 594–599. IEEE Computer Society, 2003.

[68] Unknown. Agile methodologies survey results, January 2003. http://www.shinetech.com/download/attachments/98/ShineTechAgileSurvey2003-01-17.pdf?version=1.

[69] Unknown. Why so many cplusplus test frameworks, 2004. http://c2.com/cgi/wiki?WhySoManyCplusplusTestFrameworks.

[70] I. Vessey, V. Ramesh, and R.L. Glass. A unified classification system for research in the computing disciplines. Technical Report TR-107-1, Indiana University, 2002.

[71] Erez Volk, 2005. http://cxxtest.sourceforge.net/.

[72] W. Wayt Gibbs. Software's chronic crisis. *Scientific American (International Edition) 271,*, 271(3):72–81, 1994.

[73] L. Williams. *The Collaborative Software Process*. PhD thesis, The University of Utah, August 2000.

[74] L. Williams, E.M. Maximilien, and M. Vouk. Test-driven development as a defect-reduction practice. In *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering*, pages 34–45, Nov. 2003.

[75] Laurie Williams and Robert Kessler. *Pair Programming Illuminated*. Addison-Wesley Longman, Inc., 2002.

[76] XProgramming.com. http://www.xprogramming.com/software.htm.

A

Test-Driven Learning

Test-driven learning (TDL) is an approach to teaching computer programming that involves introducing and exploring new concepts through automated unit tests. TDL offers the potential of teaching testing for free, of improving programmer comprehension and ability, and of improving software quality both in terms of design quality and defect density.

This paper introduces test-driven learning as a pedagogical tool. It will provide examples of how TDL can be incorporated at multiple levels in computer science and software engineering curriculum for beginning through professional programmers. In addition, the relationships between TDL and test-driven development will be explored.

Initial evidence indicates that TDL can improve student comprehension of new concepts while improving their testing skills with no additional instruction time. In addition, by learning to construct programs in a test-driven manner, students are expected to be more likely to develop their own code with a test-driven approach, likely resulting in improved software designs and quality.

A.1 Introduction

Programmers often learn new programming concepts and technologies through examples. Instructors and textbooks use examples to present syntax and explore semantics. Tutorials [44] and software documentation [43] regularly present examples to explain behaviors and proper use of particular software elements. Examples, however, typically focus on the use or the interface of the particular software element, without adequately addressing the behavior of the element.

Consider the following example from the Java API documentation [43]:

```
void printClassName(Object obj)
```

```
{
  System.out.println("The class of " + obj +
          " is " + obj.getClass().getName());
}
```

While this is a reasonable example of how to access an object's class and corresponding class name, it only reveals the desired interface. It teaches nothing about the underlying behavior. To see behavior, one must compile and execute the code. While it is desirable to encourage students to try things out on their own, this can be time consuming if done for every possible example, plus it significantly delays the presentation/feedback loop.

As an alternative, we can introduce a simple test that demonstrates both the interface and the expected behavior. For instance, we could replace the above example with the following example which uses the assert keyword[1].

```
void testClassName1()
{
  ArrayList al = new ArrayList();
  assert al.toString().equals("[]");
  assert al.getClass().getName()
         .equals("java.util.ArrayList");
}
```

This example shows not only the same interface information as the original example in roughly the same amount of space, but it also shows the behavior by documenting the expected results.

A second example below demonstrates the same interface using an Integer. Notice how these two examples also reveal the toString() results for an empty ArrayList ("[]") and an Integer ("5").[2]

```
void testClassName2()
{
  Integer i = new Integer(5);
  assert i.toString().equals("5");
  assert i.getClass().getName()
         .equals("java.lang.Integer");
}
```

[1] Although assert has existed in many languages for some time, the assert keyword was introduced in Java with version 1.4 and requires extra work when compiling and running:
javac -source 1.4 ClassTest.java
java -ea ClassTest

[2] If the toString() information is deemed distracting, this first assert could simply be left out of the example.

These examples demonstrate the basic idea of test-driven learning:

- Teach by example
- Present examples with automated tests
- Start with tests

Teaching by example has a double meaning in TDL. First TDL encourages instructors to teach by presenting examples with automated tests. Second, by holding tests in high regard and by writing good tests, instructors model good practices that contribute to a number of positive results. Students tend to emulate what they see modeled. So as testing becomes a habit formed by example and repetition, students may begin to see the benefits of developing software with tests and be motivated to write tests voluntarily.

The third aspect of TDL suggests a test-first approach. TDL could be applied in either a test-first or a test-last manner. With a test-last approach, a concept would be implemented, then a test would be written to demonstrate the concept's use and behavior. With a test-first approach, the test would be written prior to implementing the concept of interest. By writing a test before implementing the item under test, attention is focused on the item's interface and observable behavior. This is an instance of the test-driven development (TDD) [16] approach that will be discussed later. Although early research reports mixed results [25, 46, 57, 60], TDD seems to have the potential of producing higher quality software.

A.2 Related Work

Test-driven learning is not a radical new approach to teaching computer programming. It is a subtle, but potentially powerful way to improve teaching, both in terms of efficiency and quality of student learning, while accomplishing several important goals.

TDL builds on the ideas in Meyer's work on Design by Contract [55]. Automated unit tests instantiate the assertions of invariants and pre- and post-conditions. While contracts provide important and rigorous information, they fail to communicate and implement the use of an interface in the efficient manner of automated unit tests. Contracts have been suggested as an important complement to test-driven development [40]. The same could be said regarding TDL and contracts.

TDL was more directly inspired by the Explanation Test [17] and Learning Test [17] testing patterns proposed by Kent Beck, Jim Newkirk, and Laurent Bossavit. These patterns were suggested as mechanisms to coerce professional programmers to adopt test-driven development [16].

The Explanation Test pattern encourages developers to ask for and provide explanations in terms of tests. The pattern even suggests that rather than explaining

a sequence diagram, the explanation could be provided by “a test case that contains all of the externally visible objects and messages in the diagram." [17]

The Learning Test pattern suggests that the best way to learn about a new facility in an externally produced package of software is by writing tests. If you want to use a new method, class, or API, first write tests to learn how it works and ensure it works as you expect it to.

TDL expands significantly on the Explanation and Learning Test ideas both in its approach and its audience. Novice programmers will be presented with unit tests as examples to demonstrate how programming concepts are implemented. Further, programmers will be taught to utilize automated unit tests to explore new concepts. Typically, novice programmers use some form of direct input and output to test their programs, and relatively little attention is usually given to individual unit testing. TDL replaces input/output statements with automated unit tests.

While the former idea of using automated tests as a primary teaching mechanism is believed to be a new idea, the latter approach of requiring students to write tests in lab and project exercises has a number of predecessors. Barriocanal [13] documented an experiment in which students were asked to develop automated unit tests in programming assignments. Christensen [22] proposes that software testing should be incorporated into all programming assignments in a course, but reports only on experiences in an upper-level course. Patterson [61] presents mechanisms incorporated into the BlueJ [47] environment to support automated unit testing in introductory programming courses.

A.3 Test-Driven Learning and Test-Driven Development

Test-driven development (TDD) [16] is a software development strategy that requires that automated tests be written prior to writing functional code in small, rapid iterations. Proponents claim that TDD improves software quality both in terms of design quality and defect density [17, 74]. Although TDD has been applied in various forms for several decades [33, 48], it has gained increased attention in recent years thanks to being identified as one of the twelve core practices in Extreme Programming (XP) [15].

Extreme Programming is a lightweight, evolutionary software development process that involves developing object-oriented software in very short iterations with relatively little up front design. XP is a member of a family of what are termed agile methods [14]. Although not originally given this name, test-driven development was described as an integral practice in XP, necessary for analysis, design, and testing, but also enabling design through refactoring, collective ownership, continuous integration, and programmer courage [15].

Some definitions of TDD seem to imply that TDD is primarily a testing strategy. For instance, according to [51] when summarizing [17],

> Test-Driven Development (TDD) is a programming practice that instructs developers to write new code only if an automated test has failed, and to eliminate duplication. The goal of TDD is 'clean code that works.' [45]

However, according to XP and TDD pioneer Ward Cunningham, "Test-first coding is not a testing technique" [16]. In fact TDD goes by various names including Test-First Programming, Test-Driven Design, and Test-First Design. The *driven* in test-driven development focuses on how TDD informs and leads analysis, design and programming decisions. TDD assumes that the software design is either incomplete, or at least very pliable and open to evolutionary changes. In the context of XP, TDD even subsumes many analysis decisions. In XP, the customer is supposedly "on-site", and test writing is one of the first steps in deciding what the program should do, which is essentially an analysis step.

Another definition which captures this notion comes from The Agile Alliance [7],

> Test-driven development (TDD) is the craft of producing automated tests for production code, and using that process to drive design and programming. For every tiny bit of functionality in the production code, you first develop a test that specifies and validates what the code will do. You then produce exactly as much code as will enable that test to pass. Then you refactor (simplify and clarify) both the production code and the test code.

As is seen in this definition, promoting testing to an analysis and design step involves the important practice of refactoring [29]. Refactoring is a technique for changing the structure of an existing body of code without changing its external behavior. A test may pass, but the code may be inflexible or overly complex. By refactoring the code, the test should still pass *and* the code will be improved.

Understanding that TDD is more about analysis and design than it is about testing may be one of the most challenging conceptual shifts for new adopters of the practice. Testing has traditionally assumed the existence of a program. The idea that a test can be written before the code, and even more, that the test can aid in deciding what code to write and what its interface should look like is a radical concept for most software developers.

In the few years since XP's introduction, test-driven development has received increased individual attention. A number of respected computer scientists [16, 31, 50] have endorsed TDD as a best practice. Tools such as JUnit [31] have been developed for a range of languages specifically to support TDD. Books have been written explaining how to apply TDD. Research has begun to examine the effects of TDD on defect reduction and quality improvements in both academic and professional practitioner environments. As is seen in this work, educators have begun to examine how TDD can be integrated into computer science and software engineering assignments and now pedagogy.

Software testing is perhaps viewed as one of the least exciting topics for many software developers, and particularly for students. There may be many explanations for this perspective, not least of which is the relative lack of attention to testing in the typical undergraduate computer science curriculum. Students are notoriously bad at testing their programs. Perhaps because they know they are not likely to need to maintain or reuse their programs, they often take the shortest path to completing their programs. Rarely does this include thorough testing, much less writing automated tests.

TDD is a very disciplined development approach that on the surface seems to be difficult to teach. How do we motivate students to apply TDD? Edwards [24] has suggested an approach that incorporates testing into project grades, and he provides an example of an automated grading system that provides useful feedback. TDL pushes automated testing even earlier, to the very beginning in fact. In addition to serving as an improved teaching strategy, TDL has the promise of encouraging students to adopt TDD. As testing becomes a habit formed by example and repetition, students will be motivated to apply TDD voluntarily.

A.4 TDL Objectives

Teaching software design and testing skills can be particularly challenging. Undergraduate curriculums and industry training programs often relegate design and testing topics to separate, more advanced courses, leaving students perhaps to think that design and testing are either hard, less important, or optional.

This paper introduces test-driven learning as a mechanism for teaching and motivating the use of testing as both a design and a verification activity, by way of example. TDL can be employed starting in the earliest programming courses and continuing through advanced courses, even those for professional developers. Further TDL can be applied in educational resources from textbooks to software documentation.

Test-driven learning has the following objectives:

- Teach testing for free
- Teach automated testing frameworks simply
- Encourage the use of test-driven development
- Improve student comprehension and programming abilities
- Improve software quality both in terms of design and defect density

A.4.1 Rationale behind TDL

Some have suggested that if objects are the goal, then we should start by teaching objects as early as the first day of the first class [8, 18]. TDL takes a similar approach. If writing good tests is the goal, then start by teaching with tests. If it is always a good idea to write tests, then write tests throughout the curriculum. If quality software design is the goal, then start by focusing on habits that lead to good designs. Test-first thinking focuses on an object's interface, rather than its implementation. Test-first thinking encourages smaller, more cohesive and more loosely coupled modules. [17]

A.4.2 Teach testing for free

Examples with tests take roughly the same effort to present as examples with input/output statements or explanations. As a result, TDL adds no extra strain on a course schedule, while having the benefit of introducing testing and good testing practices. It is possible that the instructor will expend extra effort moving to a test-driven approach, but once mastered, the instructor may find the new approach simpler and more reusable because the examples contain the answers.

A.4.3 Teach automated testing frameworks

By introducing the use of testing frameworks gradually in courses, students will gain familiarity with them. As will be seen later, tests can use simple mechanisms such as standard assert statements, or they can utilize powerful frameworks that scale and enjoy widespread professional support.

A.4.4 Encourage the use of TDD

See section three.

A.4.5 Improve student comprehension

When students observe both the interface and behavior in an example with tests, they are likely to understand a concept more quickly than if they only see the interface in a traditional example. Further, if students get into the habit of thinking about and writing tests, they are expected to become better programmers.

A.4.6 Improve software quality

If TDL encourages TDD, and TDD encourages better quality software, then TDL improves software quality.

A.5 TDL in Introductory Courses

Test-driven learning can be applied from the very first day of the very first programming course. Textbooks often begin with a typical "Hello, World!" example or the declaration of a variable, some computation and an output statement. The following is a possible first program in C++:

```
#include <iostream>
using namespace std;

int main()
{
  int age;
  cout << "What is your age in years?" << endl;
  cin  >> age;
  cout << "That means you are at least "
       << age * 12
       << " months old!" << endl;
  return 0;
}
```

This approach requires the immediate explanation of the language's input/output facilities. While this is a reasonable first step, a TDL approach to the same first program might be the following:

```
#include <cassert>
using namespace std;

int main()
{
  int age = 18;
  int ageInMonths;
  ageInMonths = age * 12;
  assert(ageInMonths == 216);
  return 0;
}
```

Notice how simple assert functions from the standard C library are used, rather than a full-featured testing framework. Many languages contain a standard mechanism for executing assertions. Assertions require very little explanation and provide all the semantics needed for implementing simple tests. The assert approach minimizes the barriers to introducing unit testing. Of course there are disadvantages to this approach. For instance, if there are multiple assert statements and one fails, no

further tests are executed. Also, there is no support for independent tests or test suites. However, because the programs at this level are so small, the simplicity of assert statements seems to be a reasonable choice.

As a later example, suppose a student is learning to write *for* loops in C++. They might be presented with the program in Figure A.1.

In a lab setting, the student might then be asked to write additional unit tests to understand the concept. For instance, they might add the assert statements

```
assert(sum(-2,2)==0);
assert(sum(-4,-2)==-9);
```

Then they might be asked to write unit tests for a new, unwritten function. In doing so, they will have to design the function signature and implement a function stub. This makes them think about what they are going to do before they actually do it.

Once the programmer ventures beyond the lab into larger programming projects, tests can be isolated into a separate function like those in Figure A.2.

Tests can be at least partially isolated from each other by placing them in independent scopes. The example in Figure A.3 demonstrates the use of independent scopes and tests using objects.

A.6 TDL for more advanced students

Test-driven learning is applicable at all levels of learning. Advanced students and even professional programmers in training courses can benefit from the use of tests in explanations.

As students gain maturity, they will need more sophisticated testing frameworks. Fortunately a wonderful set of testing frameworks that go by the name xUnit [38] have emerged following the lead of JUnit [31]. The frameworks generally support independent execution of tests (i.e. execution or failure of one test has no effect on other tests), test fixtures (common test set up and tear down), and mechanisms to organize large numbers of tests into test suites.

The example in Figure A.4 demonstrates the use of TDL when exploring Java's DefaultMutableTreeNode class. Such an example might surface when first introducing tree structures in a Data Structures courses, or perhaps when a more advanced programmer is learning to construct trees for use with Java's JTree class. Notice the use of the *breadthFirstEnumeration* method and how the assert statements demonstrate not just the interface to an enumeration, but also the behavior of a breadth first search. A complementary test could be written to explore and explain depth first searches. In addition, notice that this example utilizes the JUnit framework.

```
#include <iostream>
#include <cassert>
using namespace std;

int sum(int min, int max);

int main()
{
  assert(sum(3,7)==25);

  cout << "No errors encountered" << endl;
  return 0;
}

// This function sums the integers
//   from min to max inclusive.
// Pre: min < max
// Post: return-value = min + (min+1) + ...
//   + (max-1) + max
int sum(int min, int max)
{
  int sum = 0;
  for(int i=min;i<=max;i++)
  {
    sum += i;
  }
  return sum;
}
```

Figure A.1: C++ Function with Assert

```
#include <iostream>
#include <cassert>
using namespace std;

int sum(int min, int max);
void runTests();

int main()
{
  runTests();
  return 0;
}

// This function sums the integers from
//    min to max inclusive.
// Pre: min < max
// Post: return-value = min + (min+1) + ...
//    + (max-1) + max
int sum(int min, int max)
{
  int sum = 0;
  for(int i=min;i<=max;i++)
  {
    sum += i;
  }
  return sum;
}

// This function executes all of the unit tests.
void runTests()
{
  assert(sum(3,7)==25);
  assert(sum(-2,2)==0);
  assert(sum(-4,-2)==-9);

  cout << "No errors encountered" << endl;
}
```

Figure A.2: C++ Program with Several Tests

```
#include <cassert>
using namespace std;

class Exams
{
  public:
    Exams();
    int getMin();
    void addExam(int);
  private:
    int scores[50];
    int numScores;
};

void run_tests();

int main()
{
  run_tests();
  return 0;
}

void run_tests()
{
  { //test 1 Minimum of empty list is 0
    Exams exam1;
    assert(exam1.getMin() == 0);
  } //test 1

  { //test 2
    Exams exam1;
    exam1.addExam(90);
    assert(exam1.getMin() == 90);
  } //test 2
}

//Exams function definitions go here
```

Figure A.3: C++ Program with Objects and Tests in Multiple Scopes

```
import javax.swing.tree.DefaultMutableTreeNode;

import junit.framework.TestCase;

public class TreeExploreTest extends TestCase {
 public void testNodeCreation() {
   DefaultMutableTreeNode node1 =
        new DefaultMutableTreeNode("Node1");
   DefaultMutableTreeNode node2 =
        new DefaultMutableTreeNode("Node2");
   DefaultMutableTreeNode node3 =
        new DefaultMutableTreeNode("Node3");
   DefaultMutableTreeNode node4 =
        new DefaultMutableTreeNode("Node4");
   node1.add(node2);
   node2.add(node3);
   node1.add(node4);
   Enumeration e = node1.breadthFirstEnumeration();
   assertEquals(e.nextElement(),node1);
   assertEquals(e.nextElement(),node2);
   assertEquals(e.nextElement(),node4);
   assertEquals(e.nextElement(),node3);
 }
}
```

Figure A.4: Java Program Demonstrating Tree Traversal with JUnit

A.7 Assessment of TDL

To assess the efficacy of test-driven learning, a short formal experiment was conducted in two CS1 sections.

A.7.1 Experiment Context and Design

The two CS1 sections consisted of students majoring predominantly in math, physics, and various engineering disciplines. While students were not computer science majors they generally seemed accustomed to rigorous courses.

The two sections were taught by the same instructor with one section immediately following the other. The course covered C++ using a popular textbook and consisted of two fifty minute lectures and one fifty minute lab each week. The first section (experiment group) had twenty-four students and the second section (control group) had twenty-three students.

The experiment was conducted in three lectures and one lab. The first two lectures and lab contained the first introduction to classes and the final lecture was the initial introduction to arrays. These occurred during the sixth and seventh weeks of a fourteen week semester. Previous material included selection, looping, and subroutine control structures, input/output, and parameter passing mechanisms.

The independent variable was the presentation and application of automated unit tests. While both sections had been introduced earlier to the assert mechanism from the standard C library, the first section was presented examples in a test-driven manner, utilizing assert statements. The second section was presented examples in a traditional manner using standard output with the instructor explaining the expected results.

The lab presented students with a skeleton class and a main program that invoked a set of member functions and printed the results to standard output. The skeleton class contained three member variables, but none of the member functions. The students in the first section were asked to write unit tests using assert statements for each member function *prior* to implementing the function. The students in the second section were asked to implement the member functions, using the main program to test the functions by viewing the output manually.

The dependent variable was student understanding of basic class and array concepts and applications. To observe the dependent variable, at the end of the experiment, all students were given the same short quiz. The quiz consisted of three questions. The first assessed student understanding of member visibility directly, and indirectly assessed understanding of constructors, member function invocation, and object assignment. The second question assessed student understanding of initializing, looping through, and extracting values from an array. The third question assessed student ability to recognize out-of-bounds array references.

	Students	**Exam 1** 100 total	**Quiz 1** 10 total
TDL	15	80.26	7.33
Non-TDL	20	77.05	6.7

Table A.8: TDL vs. Non-TDL All Students

	Students	**Exam 1** 100 total	**Quiz 1** 10 total
TDL	13	86.15	7.84
Non-TDL	14	86.71	7.14

Table A.9: TDL vs. Non-TDL with Exam above 73

A.7.2 Observations and Analysis

At the conclusion of the lab, the instructor observed that students in the non-TDL section seemed to move faster at first, but questions revealed slightly poorer understanding. Students in the TDL section moved more slowly at first because they were writing tests as they went. However, by the end of the lab, the TDL and non-TDL students all seemed to be at about the same place, while the TDL students appeared to have a better understanding of the class concepts based on their questions.

Students then completed their labs outside of class and returned them one week later. The quality of submissions between the two sections was very similar. Interestingly only two of the fourteen TDL section lab submissions included any tests beyond the examples completed in lab. Clearly motivating students to write tests is a challenge.

On the quiz, the TDL students scored on average more than six percent better than the non-TDL students. As is seen in Table A.8, the TDL students also scored better on the first exam in the course which preceded the TDL experiment.

The difference in the exam scores is easily attributed to a number of very low exam scores in the non-TDL section. In order to make the two sections homogeneous, students who scored very poorly on the first exam were removed from the sample. After removing the two outliers (36 and 48 out of 100) from the TDL section, the lowest score on exam 1 was 74. When we remove all students with scores below 74 on exam 1 in the non-TDL section, we find that the two sections are nearly identical prior to the experiment. Table A.9 compares the exam and quiz scores of the students in the two sections after students who scored poorly on the first exam are removed. Here we see that the exam score averages are almost the same between the two sections, and the TDL students scored seven percent better on the quiz.

As the box-plots in Figure A.5 and Figure A.6 demonstrate, the median values

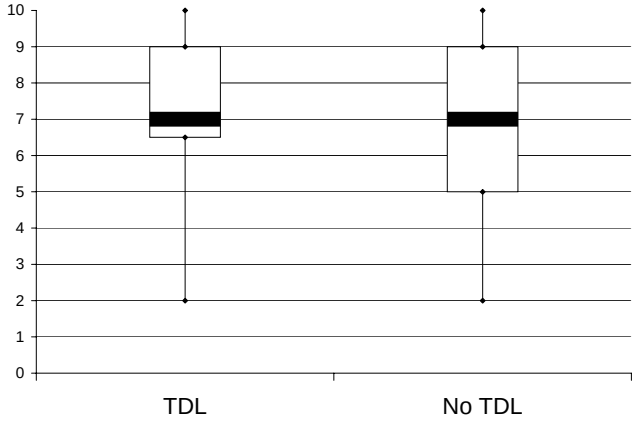

Figure A.5: TDL Quiz 1 All

of the two sections are very close, but TDL student scores clustered more closely around the median than the non-TDL student scores which were much more spread out.

Using the two-sample t-test to determine the difference between the two means, we are unable to reject the hypothesis with 95% confidence, but we come close with $p = 0.4234$. The t-Statistic is 0.8155 with 22 degrees of freedom.

Even if the data had provided greater confidence, these results would be suspect simply because the experiment duration was so short. Data from a complete semester and a replicated study would provide much greater confidence in the ability of TDL to produce improved student understanding. Still the results do point to such a potential.

A.8 Conclusions

This paper has proposed a novel method of teaching computer programming by example using automated unit tests. Examples of using this approach in a wide range of courses have been provided, and the approach has been empirically assessed. Connections between this approach and test-driven development were also explored.

This research has shown that students who were taught for a short time with the test-driven learning approach had slightly better comprehension with no additional cost in terms of instruction time or student effort. In addition, the benefits of modeling testing techniques and introducing automated unit testing frameworks have

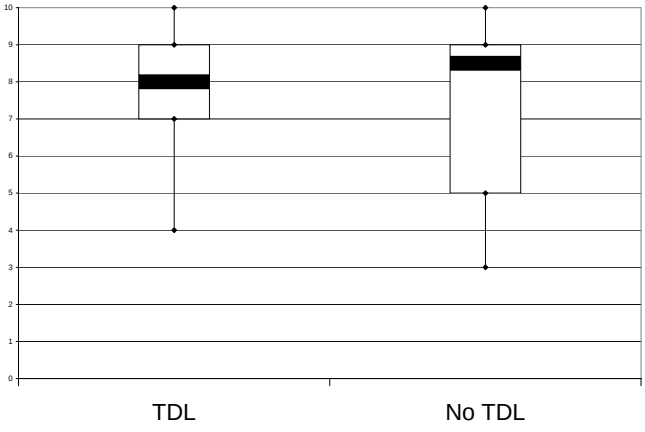

Figure A.6: TDL Quiz 1 with Exam above 73

been noted.

Additional empirical research and experience is needed to confirm the positive benefits of TDL without negative side-effects, but the approach seems to have merit. It seems reasonable that textbooks, lab books, and on-line references could be developed with the test-driven learning approach.