A PEDAGOGICAL APPROACH TO INTRODUCING TEST-DRIVEN DEVELOPMENT

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Chetan Desai

June 2008

AUTHORIZATION FOR REPRODUCTION OF MASTER'S THESIS

I reserve the reproduction rights of this thesis for a period of seven years from the date of submission. I waive reproduction rights after the time span has expired.

_____

Signature

_____

Date

APPROVAL PAGE

TITLE: A Pedagogical Approach to Introducing Test-Driven Development

AUTHOR: Chetan Desai

DATE SUBMITTED: June 2008

Dr. David Janzen	
Advisor or Committee Chair	Signature
Dr. Gene Fisher	
Committee Member	Signature
Dr. John Clements	
Committee Member	Signature

Abstract

A Pedagogical Approach to Introducing Test-Driven Development

by

Chetan Desai

Students rarely learn the value of testing in early programming courses. In fact, testing skills are poorly taught in computer science curricula altogether. Proposals to add courses on testing may be infeasible due to curriculum constraints at many universities. Integrating test-driven development (TDD) into courses has been proposed as an alternative to teaching a separate topic or course on testing. Controlled experiments can help determine when to introduce TDD in education and identify optimal teaching plans, feedback mechanisms, and tools. Four experiments were devised to gather empirical evidence on the effects of incorporating TDD into an introductory-level programming course. Effects observed include productivity of students, quality of code, attitudes towards testing, comprehension of course material, and differences in awarding points for test-code. Results indicate that students required to write tests do not spend significantly longer on assignments and quality of their source-code does not increase, but code-coverage is much better than that of students not required to write tests. Being exposed to TDD and JUnit increased the quality of projects as compared to students who took the course the prior year. Attitudes towards testing and comprehension of course material did not differ between any groups.

Acknowledgements

I would like to thank Lockheed Martin, whose grant made this experiment possible. Special thanks to Dr. Janzen for spending countless hours supporting this experiment and my thesis. I would like to further thank Dr. Clements and Professor Mammen for their flexibility and openness to allowing this experiment to be conducted on their courses, and Ben Woskow for his contributions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

Students rarely learn the value of testing in early programming courses. In fact, testing skills are poorly taught in computer science curricula altogether. Industry typically spends 50% or more of software project resources on testing. This is poorly reflected by the amount of testing in academia [24] and may convey unrealistic expectations to students. Industry professionals and university professors alike do not want graduating students starting a career with this misconception. Proposals to add courses on testing may be infeasible due to curriculum constraints at many universities. Incorporating test-driven development (TDD) into courses has been proposed as an alternative to teaching a separate topic or course on testing.

TDD tends to help students with the design of complex projects, and increases student confidence in correctness of their code and when making changes to their code[7, 16, 21]. By writing tests before code, programmers are forced to "differentiate between the functionality to implement and the base condition under which the implementation has to work" [22]. Controlled experiments can help determine when to introduce TDD in education and

identify optimal teaching plans, feedback mechanisms, and tools. Test-driven development reveals valuable software testing and design skills to fledgling programmers; the next step is figuring out how and when to introduce it into a curriculum.

1.2 Overview of Solution

The approach the author and colleagues take is to devise four experiments to analyze the effects of incorporating TDD into an introductory-level programming course. Effects observed include productivity of students, quality of source- and test-code, student attitudes towards testing, and comprehension of course material. Four experiments were conducted simultaneously to isolate several independent variables:

- *Experiment 1*: Compared a group of students being graded for the test-code they wrote to a group of students who were not required to write tests.
- *Experiment 2*: Compared current students who were exposed to TDD and JUnit to students in the course one-year earlier, who did classical manual test-last or no testing at all.
- *Experiment 3*: Compared a common project given to students in the same course but under different professors to analyze differences in teaching styles.
- *Experiment 4*: Similar to experiment 1, compared a group of students graded on test-code to a group not required to write tests, but under a different professor.

Results indicate that students required to write tests do not spend significantly longer on assignments and quality of their source-code does not increase, but code-coverage is much better than that of students not required to write tests. Being exposed to TDD and JUnit increased the quality of source-code as compared to students who took the course the prior

year. Attitudes towards testing and comprehension of course material did not differ between any groups.

1.2.1 Executive Summary

This research examined an experimental approach of introducing TDD that slowly introduced testing through example and gradually led students to write tests. Previous experiments in this area required students to learn a new programming language and test-harness such as JUnit at the same time. The approach used was faster because existing course materials were reused in a TDD fashion, and minimal setup time was required to grade test-code. Our approach was successful because it took no additional teaching time to introduce TDD and JUnit, and students did not spend significantly longer on projects when writing both source- and test-code. This exposure to TDD and JUnit produced higher quality code (in terms of the number of unit tests passed) when compared to the previous year where neither TDD nor JUnit were taught.

1.3 Limitations to Initial Scope

There were two limitations to the initial scope of the thesis. For the common project given in experiment 3, code-quality analysis was dropped from the study. Differences between requirements issued by the two professors made it difficult to gather enough comparable data. Fortunately, other aspects of experiment 3 could still be measured and compared. Lastly, project grade data for the two groups in experiment 2 had too many uncontrolled variables to reject or accept the hypothesis which addressed that aspect of the experiment. Quality of the source-code was still analyzed through a JUnit test-suite.

1.4 Outline of Thesis

The remainder of the thesis is outlined as follows. Chapter 2 will overview a background of TDD and related work in academia. Chapter 3 will present the design of the four experiments conducted during the Winter 2008 quarter. Chapter 4 analyzes the resulting data from the experiments, and Chapter 5 summarizes the conclusions drawn and discusses future work.

Chapter 2

Background & Related Work

The idea of test-driven development (TDD) has been around since the early 1960's with NASA's Project Mercury[18]. TDD received its current name and popularity after being introduced as a practice in eXtreme Programming (XP), created by Kent Beck and Ward Cunningham. XP takes twelve important fundamental software engineering practices and does them to the extreme. Testing is a fundamental practice, and XP took it to the extreme by iteratively developing tests in tandem with writing code. More recently, TDD has gained popularity outside of XP with the publication of Kent Beck's book *Test-Driven Development by Example*[3]. This led many professionals and academic instructors alike to start integrating TDD into their workplaces and universities, respectively.

2.1 Test-Driven Development

TDD develops tests and code in a unique order. TDD procedures work with units of program functionality. Units are the smallest module of functionality and are usually in the form of methods. The sequence of TDD is [3]:

1. Add a new test for an unimplemented unit of functionality.

2. Run all previously written tests and see the newly added test fail.

3. Write code that implements the new functionality.

4. Run all tests and see them succeed.

5. Refactor (rewrite to improve readability or structure).

6. Start at the beginning (repeat).

As development continues, the programmer creates a suite of unit tests that can be run automatically with testing frameworks such as JUnit. As larger modules (entire classes or packages, as opposed to single methods) are completed, integration tests may be added. The programmers now have a full regression test suite to run whenever changes are made to the system. Changes can be made with confidence, since if something breaks in another part of the system, the regression tests are likely to catch it. Furthermore, low-level design decisions are constantly being made as the programmer develops tests and source code. By making constant decisions and refactoring based on these decisions, a stable and flexible program design is produced.

2.1.1 Example TDD Iteration

To demonstrate TDD, a single iteration of the process described in the previous section is carried out. For this example, a programmer wants to develop an *Alarm.java* class to act like an alarm clock and hold the time that the alarm should go off. The time that the alarm is set for will need to be set and retrieved. Step 1 of the TDD process is to add a test for an unimplemented unit of functionality. The programmer will first need to create an *Alarm* object to work with, so he/she creates a test to make sure our object is created correctly (see test code in in Figure 2.1). The programmer decides the alarm needs to be able to hold an hour and a minute on a 24-hour time scale. When clocks are first plugged in, the

Test Code

```
1  import static org.junit.Assert.*;
2  import org.junit.Before;
3  import org.junit.Test;
4
5  public class AlarmTest {
6
7     /* Test the Alarm constructor */
8     @Test
9     public void testAlarm() {
10       Alarm myAlarm = new Alarm();
11       assertEquals(0, myAlarm.getHour());
12       assertEquals(0, myAlarm.getMinute());
13    }
14 }
```

Source Code

Figure 2.1: Artifact after step 1 of the TDD process.

alarm default is always 12:00AM. So in the test, the programmer creates an Alarm object which should be initialized to 0 hours and 0 minutes on a 24-hour time scale (see line 10 in the test code). He/she now has an idea for a constructor, the next step is the ensure the object was created correctly. The programmer needs a way to retrieve the two fields. He/she tests it using `get()` methods (lines 11-12 of the test code). Two `get()` methods are needed; one for each field. Remember that the programmer has not written a single line of source-code yet. By writing a test first, he/she is driving out low-level design decisions which allows him/her to understand the conditions in which the implementation should work.

Step 2 has one run all tests and watch the new functionality fail since the source-code is not implemented yet. The programmer currently can not run the code since there is none. Thus, he/she stubs out the new methods so that the program can compile, run, and fail the tests. Figure 2.2 shows the addition of the source-code.

Now the tests run and fail, since the `get()` methods only return -1. It is time to move to

Test Code

```
1  import static org.junit.Assert.*;
2  import org.junit.Before;
3  import org.junit.Test;
4
5  public class AlarmTest {
6
7     /* Test the Alarm constructor */
8     @Test
9     public void testAlarm() {
10       Alarm myAlarm = new Alarm();
11       assertEquals(0, myAlarm.getHour());
12       assertEquals(0, myAlarm.getMinute());
13    }
14 }
```

Source Code

```
1  public class Alarm {
2
3     /* Constructor to initialize our alarm */
4     public Alarm() {
5
6     }
7
8     /* Retrieve the hour */
9     public int getHour() {
10       return -1;
11    }
12
13    /* Retrieve the minute */
14    public int getMinute() {
15       return -1;
16    }
17 }
```

Figure 2.2: Artifact after step 2 of the TDD process.

Test Code

```
1  import static org.junit.Assert.*;
2  import org.junit.Before;
3  import org.junit.Test;
4
5  public class AlarmTest {
6
7     /* Test the Alarm constructor */
8     @Test
9     public void testAlarm() {
10       Alarm myAlarm = new Alarm();
11       assertEquals(0, myAlarm.getHour());
12       assertEquals(0, myAlarm.getMinute());
13    }
14 }
```

Source Code

```
1  public class Alarm {
2
3     private int hour;
4     private int minute;
5
6     /* Constructor to initialize our alarm */
7     public Alarm() {
8        hour = 0;
9        minute = 0;
10    }
11
12    /* Retrieve the hour */
13    public int getHour() {
14       return hour;
15    }
16
17    /* Retrieve the minute */
18    public int getMinute() {
19       return minute;
20    }
21 }
```

Figure 2.3: Artifact after step 3 of the TDD process.

step 3. In this step the new functionality is implemented. An updated version of the source code is in Figure 2.3. The programmer needs a way to store the hour and minute and thus makes private instance variables to hold these values (lines 3,4 of the source code). The constructor can now initialize these variables (lines 8,9 of the source code).

Step 4 says to re-run the tests written in step 1, and verify that they all pass. In this case, the tests now pass since the `get()` method returns the expected value. Step 5 is to refactor and improve any readability or structure. The constructor and `get()` methods are both fairly simple, so no refactoring is needed. Step 6 concludes an iteration of TDD and the programmer can start over on a new unit of functionality. Why was developing code in that manner so important? By writing the `testAlarm()` method before implementing the actual constructor, the programmer was forced to think about conditions that the implementation should work under through specifying expected outputs with given inputs. Furthermore, it drove out design decisions; the programmer needed a way to retrieve the values from the alarm, so he/she created utility methods (`get()`).

2.2 Common Misconceptions

There are many misconceptions about test-driven development. First, TDD is not a testing technique, it is a software development process. A popular misconception is that people think all the testing is done before any code is produced. This is wrong; units of test and code are interleaved during the development process. Ambler summarizes several additional misconceptions in the following list[1]:

- *You create a 100% regression test suite:* This is not realistic, especially if you are reusing assets that don't have test suites, you are testing user interfaces, you do not have a database regression tester, or you are using a legacy system which does not

have all the tests for it.

- *Unit tests form 100% of your design specification:* TDD still requires external documentation describing your design, but significantly less than on a traditional project.

- *You only need to unit test:* There is still a need for acceptance testing, user testing, system integration testing, and a host of other testing techniques.

- *TDD does not scale:* Partly true, but easily addressed. If your test suite takes too long to run, then separate your test suite into two components, one that contains the tests for the functionality that you are currently working on and the other which contains all tests. If some developers have trouble testing, then try getting them some appropriate training or pairing up with a colleague that has unit testing skills. Lastly, everyone on the team must participate, so either have those who are not taking a TDD approach leave the team, or abandon the TDD aspect altogether.

2.3 Empirical Evidence of TDD in Academia

Table 2.1 summarizes most of the studies on TDD in academia. However, side by side comparisons have inherent difficulties. Many of the studies have different independent and dependent variables, with the common purpose of finding the effects of one or more aspects of TDD. Each result should be understood within the context and environment of the study. For example, controlled experiments have different control group characteristics. In cases where the control group used iterative test-last (write a unit of code, write a unit test, repeat), many quality results did not differ as much, since continuous testing was still occurring. In cases where the control group applied a traditional test-last approach (write all the code then write all the tests) or conducted no programmatic testing at all, defect counts varied significantly. Furthermore, techniques for measuring quality and productivity differed from study to study. The most common way to measure functional quality was the number of

Study	Type	Student Level	Subjects	Productivity of Students	Quality of Programs	Other Findings
Müller [22]	Case Study	Graduate	11			87% stated regression testing increased confidence.
Edwards [7]	Cont. Exp.	Junior	118 (59 TDD / 59 Control)		45% fewer defects	Increased student confidence.
Erdogmus [9]	Cont. Exp.	Junior	24 (11 TDD / 13 Control)	52% increase	No effect	Minimum quality increased linearly with number of tests.
Janzen [11]	Cont. Exp.	Freshman	27 (13 TDL / 14 non-TDL)			TDL students had slightly better comprehension, scoring 10% higher on a quiz.
Janzen [13]	Cont. Exp.	Freshman	CS1: 106 (40 TDD, 66 Control) CS2: 36 (6 TDD, 30 Control)	CS1: Slower but not stat. sig. CS2: Faster but not stat. sig.	CS1: TDD Students wrote more asserts. CS2: TDD projects superior to control group.	TDD students felt more confident in their code w.r.t. quality, change, and reuse.
Kaufmann [16]	Cont. Exp.	Sophomore - Senior	4 (2 TDD / 2 Control)	50% more NLOC	Better CCCC metrics	Increased student confidence.
Madeyski [19]	Cont. Exp.	Sophomore - Graduate	188		External code quality stat. sig. lower	
Müller [21]	Cont. Exp.	Graduate	19 (10 TDD / 9 Control)	Faster but not stat. sig.	Less reliable w.r.t. passed assertion tests but not stat. sig.	Better program understanding w.r.t. code reuse.
Pancur [23]	Cont. Exp.	Senior	34 (19 TDD / 15 Control)	2.5% slower		90% students would accept TDD in industry
Yenduri [27]	Cont. Exp.	Senior	18 (9 TDD / 9 Control)	25.4% faster	34.8% fewer defects	
Barriocanal [2]	Exp. Report	Freshman	100			Only 10% students wrote test cases by choice
Keefe [17]	Exp. Report	Freshman	12			Of XP practices, TDD not preferred
Melnik [20]	Exp. Report	Freshman - Graduate	240	78% agreed on improvement	76% agreed on improvement	Correlation between age and attitude towards TDD.
Spacco [25]	Exp. Report	Freshman	20 - 30			Students need incentives to adopt test-first mentality early.
Janzen [12]	Field Study	Freshman - Graduate	160 (130 beginners / 30 mature)			87% mature programmers prefer TDD, 86% beginner programmers prefer test-last.

***NLOC**: non-commented lines of code* ***CCCC**: C and C++ Code Counter* ***Cont. Exp.**: controlled experiment*

***Stat. Sig.**: statistically significant* ***Exp. Report.**: experience report* ***w.r.t**: with respect to*

***TDL**: test-driven learning*

Table 2.1: Comparison Grid

unit tests passed during acceptance tests. Test quality was commonly measured through the total code-coverage obtained by the written tests on an instructor's code solution. To measure productivity, many experiments had students log the time they worked, or they counted non-commented lines of code. Student confidence levels and preferences towards using TDD were measured through pre- and post-experiment surveys.

2.3.1 TDD Benefits

By writing tests before code, programmers are forced to "differentiate between the functionality to implement and the base condition under which the implementation has to work" [22]. This forces programmers to make better design decisions during development. Most controlled experiments between TDD and other testing practices show an increase in quality of code, or minimal differences. Depending on what control group the TDD group was being compared against, results were between a 35%[27] and 45%[7] reduction in defects. Changes in productivity varied by experiment. Some experiments found vast improvements in productivity, between 24.5%[27] and 50%[16]. Others found less hopeful results of a 5-10% decrease in productivity [14]. Surveys from students have indicated an increase in program understanding [21] and confidence in making changes to the code and code correctness [22]. These results tended to be more positive in advanced courses. Mature programmers noticed the benefits of TDD and could conduct its practices correctly, where beginner programmers struggled to understand the purpose of testing.

2.3.2 Popular Frameworks

All of the examined experiments in Table 2.1 used Java except one, which used Pascal [2]. Java is a widely used language for TDD, along with JUnit, its popular test harness. JUnit was developed by Kent Beck, along with Erich Gamma. However, TDD is not limited to Java and JUnit, as there are other frameworks under the name of xUnit, used for various

programming languages. JUnit provides assertions for expected results, test fixtures for prepping and cleaning up data to perform one or more tests, and test runners to orchestrate execution of tests and report results. These abilities allow users to smoothly interchange between developing tests and code.

2.3.3 Related Work

TDD in academia has moved on from its conception as a practice in eXtreme Programming (XP), created by Kent Beck and Ward Cunningham. Several studies have tried to prove correlations and effects of adopting TDD. Most claims come from experience reports [2, 5, 17, 20, 25], case studies [22], field studies [12], and surveys [10, 14, 15]. However, there have been a handful of controlled experiments to bring forth empirical-based evidence.

Edwards [7] conducted an experiment in a junior-level course with 118 students at Virginia Tech University. He used an automated grading system named Web-CAT[1] to provide students with feedback on correctness and quality of both source- and test-code. Half of these students used TDD and submitted programs via Web-CAT during the course in Spring 2003. The other half did not use TDD and used output-based correctness for feedback on their programs in Spring 2001. Edwards found that the TDD group's programs contained 45% fewer defects and those students felt more confidence in the correctness of their code and when making changes to their code than the non-TDD control group students.

Erdogmus [9] compared a TDD group to an iterative test-last group. The controlled experiment was conducted on 24 junior-level students programming a bowling score-keeper. He found that the test-first students wrote more tests on average, and tended to be more productive. Furthermore, the quality of programs seemed to increase linearly with the num-

[1] http://web-cat.cs.vt.edu/

ber of tests written, independent of the development strategy used.

Janzen [13] conducted experiments using the test-driven learning (TDL) [11] pedagogical approach for introducing testing into a CS1 and CS2 course. He found that test-first programmers wrote more tests and scored higher on project grades than their test-last counterparts when taught in a TDL fashion.

Kaufmann and Janzen [16] looked at two small groups of four sophomore to senior students. These students developed a graphical game application, one group using TDD, the other in a test-last fashion. The TDD group wrote 50% more lines of code, had better internal quality metrics, and felt more confident in their code. However, the sample and project sizes were too small to make any general conclusions.

Madeyski [19] had four groups of sophomore to graduate students work on building a finance-accounting system. One hundred eighty-eight students were divided into groups for solo-programming and TDD (TS), pair-programming and TDD (TP), solo-programming with a classical testing approach (CS), and pair-programming with a classical testing approach (CP). Interestingly enough, results indicated that external code quality (determined by number of acceptance tests passed) was worse when TDD was used in both solo- and pair-programming.

Müller and Hagner [21] divided 19 graduate students into a TDD group and traditional development group. The students implemented a main class for a graph library containing only method declarations. The TDD group had higher productivity but passed less assertion tests. These differences were not statistically significant, so they concluded that TDD did not increase productivity or produce higher quality code. It was mentioned that this could be due to the fact that there was only a 64.5% chance to see differences in the groups due

to small sample size. Müller and Hagner did find, however, that TDD enhanced program understanding with respect to code reuse.

Pancur [23] observed a TDD group and an iterative test-last group consisting of 34 senior undergraduates. The study used an eclipse plug-in to gather data on time spent coding, number of development cycles, and number of tests run. Preliminary results show slightly less code-coverage and external code quality for the TDD group, but the results were not statistically significant.

Yenduri and Perkins [27] compared a TDD group of 9 students with a traditional incremental development group of also 9 students. The students were senior undergraduates and were trained in both approaches. The authors measured the number of test cases written, faults found, and hours spent on the project. The TDD group yielded better results in both quality (34.8% fewer defects) and productivity (25.4% faster). However, the authors report that these results need to be validated by larger projects with a larger sample size.

2.4 TDD Challenges

Adopting TDD practices in a university environment comes with several concerns. Edwards outlines five perceived roadblocks [7]:

- *Challenge 1*: Introductory students are not ready for testing until they have basic programming skills.
- *Challenge 2*: Instructors do not have enough lecture hours to teach a new topic like software testing.
- *Challenge 3*: Course staff already has its hands full grading code correctness, so it may not be feasible to assess test cases too.
- *Challenge 4*: To learn the benefits of TDD, students need frequent, concrete feedback on how to improve as they are working.
- *Challenge 5*: Students must see the value in the non-functional code (test code).

JUnit has proven to be a tough barrier in introductory programming courses. When students are learning an entirely new language like Java, trying to understand the concepts and structure of JUnit is difficult. Keefe recommends to first teach testing with sample test data, expected results, simple test plans, and retrieving actual results, before moving into TDD [17].

2.4.1 Introducing TDD

Determining when and how to introduce TDD practices into a curriculum can be difficult. Most of the experiments reported in the literature introduced TDD at the beginning of the semesters. Introductions usually consisted of:

- Explaining automated unit testing
- Describing TDD
- Providing documentation on test harnesses (e.g. JUnit API)
- Supplying examples of how to write test cases, execute test cases, and interpret results

Introduction lengths varied from a thirty-minute lecture[6] to a three-week topic[22]. Looking back on Table 1, promising results came from Edwards[7], where TDD practices were introduced briefly at the start of the semester, but then used in the classroom throughout the entire experiment to model behavior. Reinforced learning could be key to successfully introducing TDD, but controlled experiments will have to be conducted with using TDD in the classroom to model examples as the independent variable. In cases where students were just briefly introduced to testing at the start of the semester, TDD was not preferred[17] and only 10% of the students wrote test cases[2].

2.4.2 GUI Limitations

Graphical user interface (GUI) assignments allow for aesthetically pleasing and sometimes creative projects for students. Furthermore, the students are rewarded by a visually appealing result. GUI components are great candidates for object-oriented programming since concepts such as a button with a color, size, and label can be easily grasped by students[26]. However, trying to develop these in a test-first approach can be challenging.

Testing of GUI components is a challenging endeavor even for professionals. Visual aspects of programming sometimes require a human eye to *test* and ensure components are placed correctly on a screen. However, there has been work in trying to programmatically test this process. Abbot[2] is a popular Java-based testing framework for GUIs at the professional-level. However, many universities like to incorporate a GUI-based project into introductory courses[4, 26].

Bruce[4] created a GUI library called `ObjectDraw` to make Java's Swing library more user-friendly for beginner programmers. With Bruce's simplified GUI development library and an urge to develop in a test-first fashion, Thornton et al.[26] created a GUI testing framework for introductory students to successfully test their GUI applications. Since `ObjectDraw` is based on Java's Swing, it has access to all of Swing's testing components. Thornton's framework uses Abbot internally to test `ObjectDraw`'s components, and hides new testing concepts outside of basic JUnit skills. In Spring 2007, Thornton observed the use of their framework by comparing it to two previous semesters: Fall 2005 where no GUI programming was used, and Fall 2006 where `ObjectDraw` was used, but without any testing. They concluded that the `ObjectDraw` library was successful in allowing students to test their GUI programs. The students wrote more test code for GUI assignments than other assignments but they started and finished it earlier.

[2] http://abbot.sourceforge.net/

2.4.3 Current Concerns

A wide range of studies have looked at the topic of TDD in academia. Many studies try to address concerns such as showing that TDD can be introduced without any additional instruction time, and automated graders like Web-CAT and Marmoset can prevent issues like double grading of both source- and test-code. Java and JUnit are widely used in TDD. Mature-level programmers seem to realize benefits of TDD more easily than beginner-level programmers perhaps due to a higher complexity of projects and familiarity with the programming language. Many different proposals for introducing TDD into a curriculum have been tested, and the test-driven learning approach (explained in Section 2.5.1) is promising since it addresses many perceived roadblocks when introducing testing. Lastly, solutions have been brought forth to test GUI assignments, which allow more interesting projects to be assigned at an introductory level, and continue with the testing methodologies.

Most controlled experiments have looked at sophomore-graduate level classes, and rarely at CS1 or CS2 courses which present more challenges. For students starting to learn what programming is and how it works, they find it tough to find purpose in the code, so testing it is difficult [17]. Many studies simply threw students into programming Java and JUnit at the same time. Students should first learn programming syntax and semantics. Then, move into concepts of test data, test plans, and expected results. Test-harnesses such as JUnit can be used to allow for test runners and smooth transitions between developing test-code and source-code. Following the test-driven learning approach[11], JUnit can be introduced in simple contexts and reinforced through examples.

2.5 TDD Opportunities

A current pedagogical concern of university professors is deciding when to introduce TDD into their curriculum. Experiments have been conducted at all student levels. Studies tend to show that beginner programmers have a hard time using TDD, especially when trying to incorporate frameworks like JUnit. For students starting to learn what programming is and how it works, they find it tough to find purpose in the code, so testing it is difficult [17]. Tools like WebCAT [6] and Marmoset [25] have helped overcome testing hurdles. By providing feedback such as test coverage and number of unit tests passed, writing tests is associated with specific incentives for a programming novice. These incentives help eliminate the need to force beginners to write tests as part of their grade, which could be counterproductive. Students forced into writing tests does not prove they are doing it because of the benefit they get out of testing; they may simply be writing the tests as an afterthought since their grade depends on it. When left to the students to decide to write tests or not, only 10% wrote tests [2].

Results have been much more promising at higher levels of education. Many mature programmers see the benefits of TDD such as increased productivity and quality [20, 12]. As seen in Table 2.1, most of the success stories come from experiments conducted between junior undergraduate and graduate levels of education. One possible explanation is that since advanced courses assign more complex programs, TDD's benefits are more apparent to mature programmers. Full regression suites help in complex class hierarchies, and making low-level design decisions as one programs can help stabilize the architectural solution. Does this mean that TDD should not be used in introductory programming courses? Not necessarily. It means that a lot more work needs to go into CS1 and CS2 course plans.

2.5.1 Test-Driven Learning

With an incremental instructional approach, students would first learn programming syntax and semantics; then move into concepts of test data, test plans, and expected results. Once they are comfortable with that, techniques like TDD can be introduced, using tools such as JUnit, WebCAT, and Marmoset to help facilitate understanding.

In contrast to this incremental approach, test-driven learning (TDL)[11] proposes teaching by example, presenting examples with automated tests, and starting with tests. TDL needs little to no additional instruction time and targets any level of programming student or industry professional. Although TDL is presented as a test-first approach, a test-last approach can be equally beneficial. To achieve its goal of writing good tests, TDL is designed to present testing early and use it as a recurring theme throughout a course. According to [11], objectives behind TDL include:

- Teaching testing for free
- Teaching automated testing frameworks simply
- Encouraging the use of TDD
- Improving student comprehension and programming abilities
- Improving software quality both in terms of design and defect density

A short experiment was conducted and showed that TDL could be introduced with positive feedback at no additional teaching time or student effort. A subsequent TDL experiment on a CS1 course and a CS2 course [13] found that test-first programmers wrote more tests and scored higher on project grades than their test-last counterparts when taught in a TDL fashion. In the next chapter, the author and colleagues apply a TDL-influenced approach in the design of an experiment to address the problems outlined in Section 1.1.

Chapter 3

Experimental Design

Most controlled experiments have looked at sophomore- to graduate-level classes, and rarely at CS1 or CS2 courses which present more challenges. For students starting to learn what programming is and how it works, they find it tough to find purpose in the code, so testing it is difficult [17]. Many studies had students learn and program both Java and JUnit right from the start. Traditionally, students must learn programming syntax and semantics. Then, move into concepts of test data, test plans, and expected results. Following the test-driven learning approach[11], JUnit can be introduced from the beginning in simple contexts and reinforced through examples. Incorporating testing from the very start of a student's programming experience is fundamentally important to teach analytical and comprehension skills needed in software testing. If curricula can get students 'test-infected' from the beginning, they are likely to realize that testing is an integral part of programming.

3.1 Goals

The goal of this study is to obtain empirical evidence on integrating TDD into early programming courses with no extra costs, except the one-time cost of instructors getting up

Concern	Description
C1	Cost of using existing materials in a TDD fashion.
C2	Effectiveness of giving credit to student's test-code.
C3	TDD's affects on productivity of students.
C4	TDD's affects on quality of code.
C5	The effect of instructors using TDD for in-class examples.

Table 3.1: Concerns to Address

to speed. Specific concerns are summarized in Table 3.1. The first question is whether or not TDD can be integrated into an introductory programming course with minimal impact (**C1**). The topics taught should remain the same. Is it possible to take existing materials and tack on a TDD approach? Or is it best to rewrite materials such that a TDD approach can be easily incorporated.

A second concern is what effectiveness does the grading of test-code have on students (**C2**). Is giving credit to tests the best way to teach TDD? Do students write more, higher quality tests if they get feedback through grades on tests? Barriocanal [2] mentioned that grading tests could be counterproductive if students write the tests as an afterthought since their grade depends on it, and are not truly doing TDD.

An area to address is TDD's affects on quality of code and productivity of students (**C3** & **C4**). Does the TDD approach affect the amount of time spent on projects, since students have to write test-code? By writing test-code, students might save time in debugging. Does writing tests lead to higher quality code with respect to the number of acceptance tests passed?

Different teaching approaches can make either a world of difference or a minimal difference, depending on the topic. TDD could be either. If in-class examples are developed using a TDD approach, does it have a higher impact on students than those who do not see testing in class? Concern **C5** hopes to address issues like this.

3.2 Approach

The approach the author and colleagues take to answering the questions presented in the previous section is to devise four experiments, looking at various aspects of TDD effectiveness and incorporation in an introductory-level programming course. The experiments are outlined below.

3.2.1 Experiment 1 (E1)

Experiment 1 took place in a one-quarter first-year course at California Polytechnic State University, San Luis Obispo. The course was the second quarter so it is part CS1, part CS2 (denoted CS1/CS2) and was held in Winter 2008. Eighty-three students took the course divided into three sections with 24, 29, and 30 students respectively. These three sections are all taught by the same instructor denoted by Professor A. All of the students had completed a CS1 course in the previous quarter where they learned the basics of programming in the C language. In this CS1/CS2 course all the students go through a paradigm shift to object-oriented programming in the Java language. All the sections were given the same labs, projects and lectures. Three sections of students were divided into two groups. The first group consisted of two sections of the course, totaling 59 students, and part of their assignment grade was based on the test-code they wrote. This group will be denoted as the graded tests group (GT) from now on. The other group was one section of the course, consisting of 24 students, whose project grades did not depend on the tests they wrote. This group is called the ungraded tests group (UT).

For this experiment, Professor A's lab and project descriptions from a previous quarter were modified, emphasizing and encouraging a TDD approach to development. All of the topics and structure of the programming assignments stayed the same to address concern **C1**. Full

```
/* Test method for move() */
@Test
public void testMove() {
   // Create three vertices for our triangle
   Point vertA = new Point(7, -3);
   Point vertB = new Point(13, 56);
   Point vertC = new Point(-3, 23);

   // Create our triangle
   Triangle toMove = new Triangle(vertA, vertB, vertC, Color.cyan, false);

   // Perform the move operation by the given amount
   toMove.move(new Point(-5, -7));

   // Assert that the new vertices of our triangle are as expected
   assertEquals(new Point(2, -10), toMove.getVertexA());
   assertEquals(new Point(8, 49), toMove.getVertexB());
   assertEquals(new Point(-8, 16), toMove.getVertexC());
}
```

Figure 3.1: A provided JUnit test from `TriangleTest.java`

JUnit suites were also written for each project. The approach the author and colleagues took was to introduce TDD in this course by exposing Java and JUnit at the same time. The students were eased into writing their own JUnit tests over time. Students were given full JUnit test suites for projects and labs early in the course. This allowed them to over-come the hurdle of learning basic syntax and semantics of a new programming language at the beginning of a course, but be exposed to testing through example. To ease students into writing their own tests, the second project supplied JUnit tests for a Java class similar to one the student had to test. For example, JUnit tests were supplied for a *Triangle* class, and the students had to write tests for a *Rectangle* class. Figure 3.1 shows an example of a unit test from the supplied JUnit test case for the *Triangle* class. The `move()` method shifts the vertices of the shape by the supplied amount in the x- and y-axis. By having a concrete example of how one would test the `move()` method, the students could write a similar test for *Rectangle*'s `move()` method. Only slight modifications had to be done to achieve the test they were looking for. Other supplied tests include the expected area or color of a shape.

Week	Lab Topic	Program Topic
1	String Concatenation	Java Classes
2	ArrayList	ArrayList
3	Random Class	Abstract Classes, Inheritance
4	Exceptions, `equals()`	Inheritance, Polymorphism
5	File I/O	Exceptions, I/O, `equals()`
6	Comparator	Comparator, Sorting
7	N/A	LinkedList
8	Iterable, `foreach`, Wrapper Classes	Recursive LinkedList
9	GUI	N/A

Table 3.2: Schedule of Topics in E1

This approach gave students a contextual foundation to base new code on.

The fortune of reusable automatic unit tests was exemplified through projects that built upon one another. In one project, students created different shape objects such as *Triangle*, *Rectangle*, and *Circle*. In a follow up project, students had to extract common functionality and member data from these classes into an abstract class called *Shape*. The previously written tests did not have to change, and it provided a test suite to ensure the student did not break any functionality in the process. This revealed the refactoring benefits of test-driven development.

The schedule of topics covered is outlined in Table 3.2. The assigned labs and projects are all available online[1].

3.2.2 Experiment 2 (E2)

The same projects and labs from E1 were used for the same course exactly one-year earlier (Winter 2007). The only difference between the project assignments was that comments were added to the Winter 2008 descriptions emphasizing a TDD approach as described in E1. Thus, we compare the quality of projects in the course one-year prior where there was

[1] http://users.csc.calpoly.edu/~djanzen/research/TDD08/cdesai/

no JUnit or TDD integrated into the course, to the quality of the Winter 2008 projects which introduced TDD and incorporated the use of JUnit. The Winter 2007 group did classical manual test-last or no testing at all, whereas the Winter 2008 group used TDD and developed JUnit tests. We shall denote the Winter 2007 group as the classical test-last group (CT) and the Winter 2008 group as the test-first group (TF).

3.2.3 Experiment 3 (E3)

Two additional sections of the same CS1/CS2 course were taught during the Winter 2008 quarter but by a different instructor denoted Professor B. There were 28 students in one of his sections, and 25 students in the other. His students were taught in a test-first manner. His sections followed a design recipe presented in "How To Design Programs"[2]. Figure 12 of Chapter 6 in this textbook outlines the recipe as follows:

1. Formulate a structure definition and a data definition.
2. Write the method signature, and describe its purpose to formulate a header.
3. Add tests for the method. Characterize input-output relationships via examples.
4. Write the template for the method to formulate an outline.
5. Complete the method in each class by developing the body.
6. Run the test cases to check the outputs are as predicted.

A common project between all CS1/CS2 students was assigned in week 7 of a 10 week quarter. Two groups, Professor A's students (group A) and Professor B's students (group B) were compared to see how the students performed being taught the test-first development

[2]http://www.htdp.org/

Syntax	`check EXPRESSION expect EXPRESSION`
Example	`check triangle.numEdges() expect 3`

Figure 3.2: ProfessorJ's check-expect Expression

methodology differently. Professor A's students were encouraged in labs to develop in a test-driven style. Little, if any, testing was taught within the classroom lectures. Professor B's students were taught test-first through example. Professor B heavily emphasized testing by walking through the design recipe whenever writing code in class. His sections introduced Java in this course using ProfessorJ[3]. ProfessorJ has direct unit-testing support through a *check-expect* expression shown in Figure 3.2. This environment requires test cases in order to run programs, so it fit well with the test-first approach. About mid-way through the quarter, the students transitioned to the Eclipse[4] development environment with the JUnit testing framework.

3.2.4 Experiment 4 (E4)

Similar to E1, Professor B divided his sections into two groups based on grading student's test-code. One of his sections was the graded-tests (GT) group and the students were assigned a project grade based on both source- and test-code. The other section made up the ungraded-tests (UT) group, whom were assigned project grades based solely on source-code.

3.3 Experimental Variables

The independent and dependent variables per experiment are outlined in Table 3.3. The purpose of experiment **E1** is to compare differences when students are graded on their test-code and when they are not. **E2** looks to see if adding TDD in a introductory-level course

[3]http://www.professorj.org/
[4]http://www.eclipse.org/

Experiment	Independent Variables	Dependent Variables
E1	graded test-code	# tests passed # hours worked code-coverage grade on projects student attitudes toward test-first final exam grade score on final exam code-testing question
E2	use of TDD+JUnit in CS1/CS2	# tests passed grade on projects
E3	teaching style	# tests passed (common project only) # hours worked (common project only) code-coverage (common project only) student attitudes toward test-first final exam grade score on final exam code-testing question
E4	graded test-code	grade on projects student attitudes toward test-first final exam grade score on final exam code-testing question

Table 3.3: Independent and Dependent Variables

affects the quality of student programs. **E3** looks to compare different teaching styles of TDD. Lastly, **E4** also compares differences when students are graded on their test-code and when they are not. To measure our dependent variables, we use several metrics.

3.3.1 Metrics

Of the related work observed, the most popular way to measure quality of programs was through the total number of instructor unit tests passed, or the code-coverage achieved by student tests. The most common way to measure productivity of students was through a time log of hours worked on the project, or the total number of non-commented lines of code produced. To gather empirical evidence to address the concerns in Table 3.1, we establish several metrics outlined in Table 3.4. **Quality** of projects, is determined by three different measurements. We count defects by running every student's code against the instructor's

Metric	Measurement
Quality	# JUnit tests passed project grade code coverage
Attitudes	pre- and post-experiment survey
Productivity	# hours spent per project
Comprehension	final exam grade score on final exam code-testing question

Table 3.4: Measurements and Corresponding Metrics

```
Sally Student
Project 3
7.5 Hours
```

Figure 3.3: Sample `time.txt` File

suite of JUnit tests. The final assigned project grade can help determine functional correctness as well as validity of the approach which only code-reviews can unfold. Code-coverage is measured with the tool Cobertura[5] which gives various number coverage measurements (e.g. line-coverage, branch-coverage, method-coverage). Student **attitudes** toward testing and overall knowledge of different testing approaches is measured through pre- and post-experiment surveys. **Productivity** of the students is gathered by having students submit a `time.txt` file with each project. This file contains the total number of hours worked on the submitted project (see Figure 3.3). Lastly, **comprehension** of the course material and testing purposes is measured through the overall final exam score, as well as scores on specific questions on the final exam having to do with testing and writing tests for given code.

3.4 Hypotheses

Table 3.5 formally defines hypotheses for this experiment. Within experiment E1, hypothesis **H1** looks to see whether grading tests affects the total number of unit tests passed (TEST). Hypothesis **H2** examines the total number of hours (HOURS) worked per project to see

[5]http://cobertura.sourceforge.net/

Experiment	Name	Null Hypothesis	Alternative Hypothesis
E1	H1	$\mid TEST_{GT} \mid = \mid TEST_{UT} \mid$	$\mid TEST_{GT} \mid > \mid TEST_{UT} \mid$
	H2	$\mid HOURS_{GT} \mid > \mid HOURS_{UT} \mid$	$\mid HOURS_{GT} \mid \leq \mid HOURS_{UT} \mid$
	H3	$CVRG_{GT} = CVRG_{UT}$	$CVRG_{GT} \geq CVRG_{UT}$
	H4	$GRADE_{GT} = GRADE_{UT}$	$GRADE_{GT} \geq GRADE_{UT}$
	H5	$ATT_{GT} = ATT_{UT}$	$ATT_{GT} \geq ATT_{UT}$
	H6	$FINAL_{GT} = FINAL_{UT}$	$FINAL_{GT} \geq FINAL_{UT}$
	H7	$QUES_{GT} = QUES_{UT}$	$QUES_{GT} \geq QUES_{UT}$
E2	H8	$\mid TEST_{TF} \mid = \mid TEST_{CT} \mid$	$\mid TEST_{TF} \mid > \mid TEST_{CT} \mid$
	H9	$GRADE_{TF} = GRADE_{CT}$	$GRADE_{TF} \geq GRADE_{CT}$
E3	H10	$\mid TEST_{A} \mid = \mid TEST_{B} \mid$	$\mid TEST_{A} \mid < \mid TEST_{B} \mid$
	H11	$\mid HOURS_{A} \mid = \mid HOURS_{B} \mid$	$\mid HOURS_{A} \mid \geq \mid HOURS_{B} \mid$
	H12	$CVRG_{A} = CVRG_{B}$	$CVRG_{A} \leq CVRG_{B}$
	H13	$ATT_{A} = ATT_{B}$	$ATT_{A} \leq ATT_{B}$
	H14	$FINAL_{A} = FINAL_{B}$	$FINAL_{A} \leq FINAL_{B}$
	H15	$QUES_{A} = QUES_{B}$	$QUES_{A} \leq QUES_{B}$
E4	H16	$GRADE_{GT} = GRADE_{UT}$	$GRADE_{GT} \geq GRADE_{UT}$
	H17	$ATT_{GT} = ATT_{UT}$	$ATT_{GT} \geq ATT_{UT}$
	H18	$FINAL_{GT} = FINAL_{UT}$	$FINAL_{GT} \geq FINAL_{UT}$
	H19	$QUES_{GT} = QUES_{UT}$	$QUES_{GT} \geq QUES_{UT}$

Table 3.5: Formal Hypotheses

effects on productivity. **H3** compares the total amount of line-coverage (CVRG). **H4** looks at instructor assigned grades (GRADE) on the projects. **H5** compares survey information on student attitudes (ATT) towards the test-first approach. All sections of this course, independent of the instructor, took a common final exam. **H6** compares differences in scores on the final exam (FINAL). Similarly, there was a specific question (QUES) on the exam where the students had to write some test-code; **H7** compares the scores achieved on that question.

Experiment E2 has two hypotheses: **H8** compares the total number of unit tests passed between the TF group and the CT group. Hypothesis **H9** compares the assigned grades per project between the two groups.

Experiment E3 looks at all the measurements seen in E1 except grades on the common project, since final scores were given independently by the two different instructors. **H10**

compares the total number of unit tests passed for two groups under different teaching styles (one by Professor A and the other by Professor B). **H11** looks at the hours worked between the two groups, **H12** compares line-coverage, **H13** is on attitudes toward test-first, and lastly **H14** and **H15** utilize comparisons with the final exam.

H16 through **H19** use measurements seen in the previous experiments, but compare the graded-tests group to the ungraded-tests group under Professor B's instruction. The hypotheses are the same, just with a different sample set.

To test the hypotheses outlined in Table 3.5, measurements will be gathered and analyzed on the dependent variables stated in Table 3.3. These statistics will be used to determine differences between the populations. Statistical significance will be measured with a two-tailed t-test using a p-value < 0.05.

3.5 Threats to Validity

There were several threats to validity identified with the experiments. First off, conducting four different experiments at once can be a threat to validity on its own. By looking at too many questions at once, multiple variables may arise in the data, making it difficult to analyze. Internal and external validity concerns are outlined below.

3.5.1 Internal Validity

A few threats to internal validity arise looking at experiments E1 and E2. In E1, the professor had never regularly used JUnit before nor had any formal lessons on TDD. With the professor coming up to speed with these new topics, it was difficult to teach common pitfalls, answer advanced testing questions, and give solid in-class examples. A threat in

E2 is that the productivity of students between the two groups could not be analyzed since there was no time data (# hours worked per project) available from the CT group. In E3, Professor B's students started learning Java with the ProfessorJ development environment, providing them with the *check-expect* utility that was not available to Professor A's students. Furthermore, education levels between Professor A and Professor B differed. Professor A is a Lecturer with a M.S. degree whereas Professor B is an Assistant Professor with a Ph.D.

3.5.2 External Validity

There is one main threat to external validity throughout these experiments. All of the CS1/CS2 students came from a CS1 course that was taught in the C programming language. Thus, the students went through a paradigm shift from the strictly procedural functionality of C to object-oriented programming in Java. Another threat is that all groups were split up by what section of the course they were in. However, cumulative GPA data was analyzed for each section. In E1, 42.3% of the students in the UT group reported having a GPA over 3.0, where the GT group had 56.9%. For E3, 51.9% of the students in Professor A's class had over a 3.0, where only 33.3% of Professor B's students did. Lastly, for E4, 33.3% of both the UT and GT group had a GPA over 3.0.

3.5.3 Proposed Design Enhancements

There are several design enhancements that could lead to a more ideal experiment. First, only conducting one experiment at a time allows a more focused and greater depth of study. Too much may be going on when conducting four experiments at a time. When teaching TDD and JUnit to a class, it is important to have the professor up to speed ahead of time. The professors can more easily address student difficulties this way. Furthermore, proper lab introductions to both TDD and JUnit may help get the students up to speed, instead of relying on descriptions of how to develop in a test-first manner and learning JUnit only by

example.

In E1, an acceptance test was provided at the end of each project, and an interface was specified with each project. With a large number of students, having a specified interface was optimal for automated grading. However, a better route to take would be to not give an acceptance test or interface, so that TDD drives out low-level design decisions. When grading test-code, method-based code coverage was used to award points. Line-based code coverage can provided a more accurate measure of quality of tests. Qualitative grading of tests by the instructor as well as using code-coverage tools is ideal. In this experiment, groups were divided by course section. A better way to divide groups is based on statistical measurements of GPA, past course experience, and other similar attributes.

3.6 Addressing Concerns

By conducting four experiments, we were able to address each of the concerns presented in the goals of the study. For E1, almost all of the existing materials were used and we could just tack on a TDD approach to development, addressing concern **C1**. There are however, two particular structural aspects of a program that makes testing easier for the students. The first is having class constructors which take all instance fields as parameters. Using this approach, testing with `get()` methods are straightforward, since default initialization values are not arbitrary. Expected results for these tests can be seen right in the construction of the object. Perhaps the most beneficial aspect of this approach is when testing the `equals()` method. Second, for long parent/child relationship chains through inheritance, it can be difficult to construct a variety of objects with `set()` methods to test all levels of instance variables. If the constructors of these objects take all the instance fields of parameters, one can localize the testing to the inheriting class, and not have to bother with using tests of parent classes.

Concern **C2** was addressed in E1 and E4 by creating two groups of students, one group graded on their test-code, and the other not. Productivity and quality concerns (**C3** & **C4**) were addressed by gathering data to measure these attributes. The methods for this were outlined in Section 3.3.1. The purpose of E2 was to formally address concerns **C3** & **C4** by comparing a TDD-exposed group, with a group that had not been exposed to TDD or JUnit.

Experiment E3 aimed to address concern **C5**, differentiating between two approaches of teaching TDD. Results from this experiment will shed some insight on two very different teaching approaches and present some empirical data to help answer this concern.

Ideally, we would have liked to conduct an experiment as outlined in Section 3.5.3, but we were able to address all concerns with our given situation.

Chapter 4

Results & Analysis

4.1 Experiment 1: *Graded Tests vs Ungraded Tests*

Experiment 1 sought to find differences between the effects of grading students on their test-code versus not awarding points to test-code. Four different categories of dependent variables were looked at: productivity, quality, attitudes towards testing, and comprehension of the course material.

4.1.1 Productivity

The average number of hours worked between the ungraded-tests group UT and graded-tests group GT is displayed in Table 4.1. A two-tailed t-test is used to check for statistical significance using a p-value of 0.05. For the first project, both groups were given full JUnit test suites to introduce them to the syntax and semantics of JUnit. Therefore, productivity was nearly identical on the first project as expected, since neither group wrote tests. For the second and fourth projects, the GT group spent significantly longer on the projects. In general, the GT group spent more time on all remaining projects than the UT group, but the results were not statistically significant. The third project built upon the second project, so

Proj. #	**UT Avg. Prod.** (hrs worked)	**GT Avg. Prod.** (hrs worked)	***t*-Test Val.**	**Stat. Sig?** (p-val = 0.05)
1	4.18	4.93	0.240	No
2	8.97	12.2	0.023	Yes
3	7.69	10.62	0.060	No
4	9.49	12.99	0.041	Yes
5	11.75	16.33	0.130	No
6	7.35	9.90	0.158	No
7	10.39	12.10	0.384	No
8	6.27	7.80	0.090	No

Table 4.1: Experiment 1 Productivity Analysis: # Hours Worked

Proj. #	**UT Max** (# hrs)	**GT Max** (# hrs)	**UT Min** (# hrs)	**GT Min** (# hrs)	**UT Median** (# hrs)	**GT Median** (# hrs)
1	9	14.14	1.5	2.25	3.5	4
2	15	40	3	3.5	10	11.5
3	15	50	2	2.5	7	9
4	21	50	3	4	8.25	11.5
5	23	60	6.25	5	11	13.6
6	15	50	4	2	6.5	7.5
7	16	50	6.5	1.5	9	10
8	10	25	3.5	2	6	8

Table 4.2: Experiment 1 Productivity Statistics

tests could be reused from project two to three. This could explain why time differences were not as extreme for project three. Project four was the beginning of another series project, so tests were reused in projects five and six, which could explain insignificant differences for these two projects. For projects 7 and 8, students had to write all of the tests on their own and while the GT group spent more time on the projects, it was not significantly more. The trend shows a steep learning curve for when students first have to write tests (projects 2 and 4). However, when students had to write all the tests on their own in projects 7 and 8, the time it took them was not significantly longer.

Additional statistics for the productivity of students on these projects is outlined in Table 4.2. Note that while the medians per project were only around two hours more for the GT

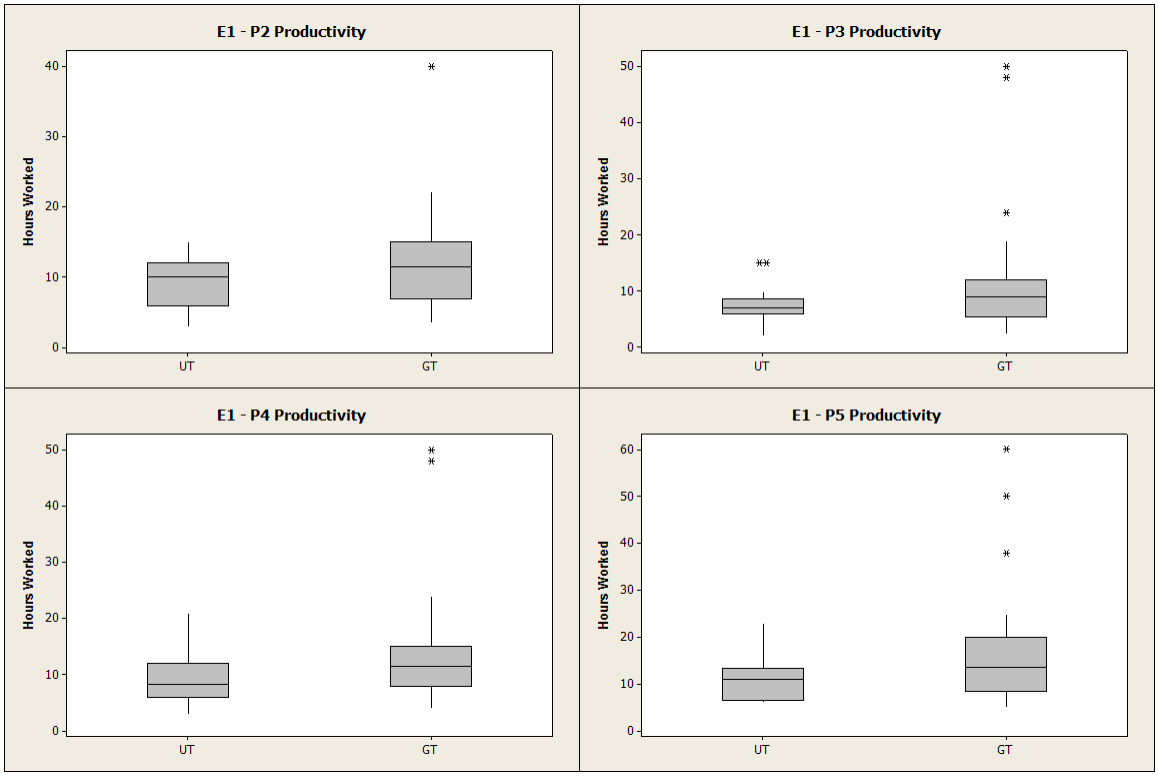

Figure 4.1: Experiment 1: Projects 2-5 Productivity Boxplots

group, the maximums were around twenty to thirty hours more than the UT group. This situation brings up the concern of outliers. Figure 4.1 shows boxplots for projects whose averages were around three hours different. The stars indicate outliers. The GT group clearly had several extreme outliers. This explains why differences in the averages were more extreme than the median differences.

A further productivity analysis is seen in Figure 4.2. This chart outlines work-load differences between the two groups compared to approximately how much additional work the GT group had due to writing tests. For each project, the approximate percentage of tests required to be written by the GT group to fully test their program is plotted. The corresponding percent increase in average number of hours worked per project is displayed. This graph gives insight to the question if the percent increase in number of tests required of the GT students correlated with their percent increase in hours worked. For the very first project,

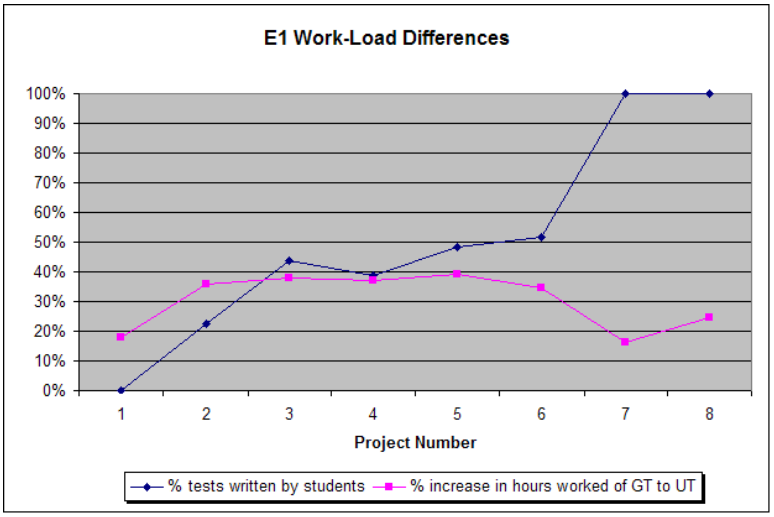

Figure 4.2: Experiment 1: Work-Load Differences

when neither group had to write tests, we see the GT group taking 18% longer than the UT group. This gives the GT group a predisposition to spend more time on projects for unknown reasons. Nonetheless, the graph shows an initial increase in time worked by the GT group as writing tests were required of them. However, by the end of the course, the trend shows that the GT group got the hang of writing tests and was not spending all that much longer on the projects, even as more and more tests were required of them.

4.1.2 Quality

The quality of the projects was determined using the number of passed JUnit tests, the student's overall project grade, and code-coverage. The student's source code was run against an instructor's suite of JUnit tests to determine the number of passed JUnit tests. Code-coverage was determined by running students tests on the instructors source-code.

Passed JUnit Tests

Table 4.3 summarizes the percent of JUnit tests passed. To receive credit, a student has to pass a given acceptance test provided one day before the due date. Thus, we expect

Proj. #	**UT Tests Passed** (avg. % passed)	**GT Tests Passed** (avg. % passed)	***t*-Test Val.**	**Stat. Sig?** (p-val = 0.05)
1	97.31%	98.43%	0.518	No
2	94.14%	97.86%	0.296	No
3	97.38%	97.06%	0.749	No
4	98.23%	98.38%	0.910	No
5	97.70%	97.13%	0.752	No
6	97.42%	96.05%	0.340	No
7	94.62%	92.91%	0.606	No

Table 4.3: Experiment 1 Quality Analysis: JUnit Tests Passed

Proj. #	**Avg. UT Grade**	**Avg. GT Grade**	***t*-Test Val.**	**Stat. Sig?** (p-val = 0.05)
1	91.36%	93.57%	0.423	No
2	78.57%	79.19%	0.903	No
3	81.34%	81.43%	0.982	No
4	82.05%	83.36%	0.698	No
5	80.00%	76.40%	0.460	No
6	86.36%	83.95%	0.594	No
7	78.33%	82.30%	0.439	No
Course Grade	54.41%	62.45%	0.220	No

Table 4.4: Experiment 1 Quality Analysis: Project Grade

the percentages of passed unit tests to be high. When comparing the GT group to the UT group, we see no significant differences in the number of unit tests passed. In most cases, the percentages were nearly equal. Grading test-code did not have an effect on the number of unit tests passed.

Project Grade

The overall score on the project is a good indicator of quality of code as well. A project grade has the additional quality criteria of human inspection from the instructor. Table 4.4 shows that while in most cases the GT group had a higher average grade, the differences were not significant. The average final class grade was again higher for the GT group, but not significant. This trend follows what was noticed in the quality measurement of passed JUnit tests.

Proj. #	**UT Branch Cov.** (avg. coverage %)	**GT Branch Cov.** (avg. coverage %)	***t*-Test Val.**	**Stat. Sig?** (p-val = 0.05)
1	87.23%	87.32%	0.966	No
2	79.59%	85.76%	0.348	No
3	86.69%	86.65%	0.987	No
4	69.34%	81.48%	0.039	Yes
5	57.69%	70.88%	0.071	No
6	35.80%	61.39%	0.003	Yes
7	20.11%	56.10%	0.006	Yes

Table 4.5: Experiment 1 Quality Analysis: Branch Coverage

Code-Coverage

Code-coverage is used to measure the quality of student-written tests. Tests that cover multiple aspects of a program's execution are more likely to catch defects, which simple tests might miss. The author and colleagues measure code-coverage by line coverage and branch coverage. Line coverage ensures that every executable line of code is run at least once. Branch coverage requires that every branch in a program has been executed at least once. Table 4.5 summarizes branch coverage percents between the GT and UT groups, and Table 4.6 presents line coverage percentages. In each of these cases, percentages of coverage drop to low levels over time. The UT group's coverage drops more than the GT group, with significant differences. Tests were graded based on method coverage, so as long as they achieved 100% method coverage, their grade was satisfied. Nonetheless, 60% line coverage is not bad for introductory students. Grading based on line coverage could ensure higher quality tests.

A concern is if the rate of the decreasing coverage coincides with the amount of tests provided. This would mean that students tests were not well-written or written at all, and the provided tests are the crutch for showing any coverage. Figure 4.3 plots out the line coverage percentages for the groups compared to the line coverage of the given tests. For the first four projects, it looks as if all the students kept up great coverage percentages as the coverage

Proj. #	**UT Line Cov.** (avg. coverage %)	**GT Line Cov.** (avg. coverage %)	***t*-Test Val.**	**Stat. Sig?** (p-val = 0.05)
1	94.84%	95.60%	0.565	No
2	93.77%	96.14%	0.506	No
3	94.12%	94.68%	0.675	No
4	81.96%	92.26%	0.056	No
5	52.70%	69.31%	0.005	Yes
6	36.23%	65.08%	0.0004	Yes
7	26.98%	63.92%	0.009	Yes

Table 4.6: Experiment 1 Quality Analysis: Line Coverage

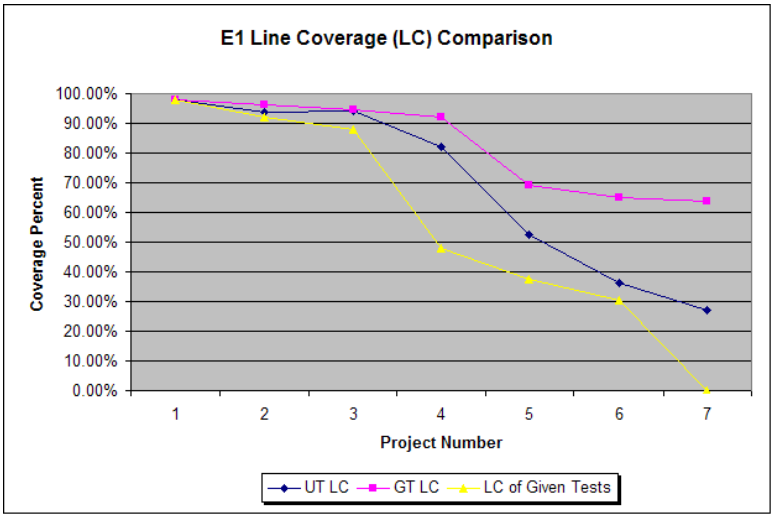

Figure 4.3: Experiment 1: Line Coverage Comparison

of the given tests diminished. Project five had a significant drop in test quality. Concepts of exceptions and file-based streams were introduced in this project; each concept requiring more complex tests. Towards the end of the course, the quality of the UT group's tests dropped significantly lower than the GT group, possibly because they were not required to do any tests and thus put it off. High quality tests could have been sacrificed be due to the work load of computer science students towards the end of a quarter, where many final projects are due in multiple classes, and preparations for final exams take up time.

4.1.3 Attitudes

Attitudes towards test-first programming and test-last programming were tracked through pre- and post-experiment surveys. These surveys were administered through SurveyMonkey[1] and given on the first and last weeks of the quarter. Differences between groups were analyzed with a two-sample t-test and within groups with a paired t-test. Three of the survey questions were analyzed. The first two questions rated importance of testing, and the last question allowed students to choose a testing preference. The questions are outlined in Figure 4.4.

Figure 4.5 graphically displays attitudes towards testing. There were no significant differences between group attitudes. Furthermore, there were no significant differences within groups (students changing their opinions). However, both groups valued test-last programming more highly in the end. This is reflected with the third question analyzed, as more students chose test-last instead of test-first. Most self-reported reasonings for the given choice was because test-last was what the beginner students were used to, and they found it easier to think of test-cases with written source-code in front of them.

[1] http://www.surveymonkey.com

Question 1:
How important is it to write a test before writing the corresponding method?

- `Not Important`
- `Somewhat Important`
- `Important`
- `Very Important`

Question 2:
How important is it to write a test after writing the corresponding method?

- `Not Important`
- `Somewhat Important`
- `Important`
- `Very Important`

Question 3:
If you had a choice to write code with a test-first or test-last approach, which would you choose?

- `Test-First`
- `Test-Last`

Figure 4.4: Three Survey Questions Analyzed

4.1.4 Comprehension

A reasonable way to measure course comprehension is by inspecting scores on the cumulative final exam. All students in this course took the same final exam, independent of which instructor they had. To remove differences in grading between instructors, three questions were re-graded by the author. Details of these questions are outlined below. Results will be discussed afterwards.

Question 1
The first question required students to write code to parse data from a file. Three points were awarded by the author. One point was given if the student dealt with I/O concerns correctly (i.e. File and Scanner class creation and usage). The second point was awarded if the student handled exceptions correctly. Lastly, the student was given an additional point if all parsing logic was correct.

Question 2
The second question was a test plan for three pieces of data. A standard boundary analysis was used to grade this question. For each of the three pieces of data, the author ensured the student gave five different test cases: below the lower bound, on the lower bound, in the range of acceptable data, on the upper

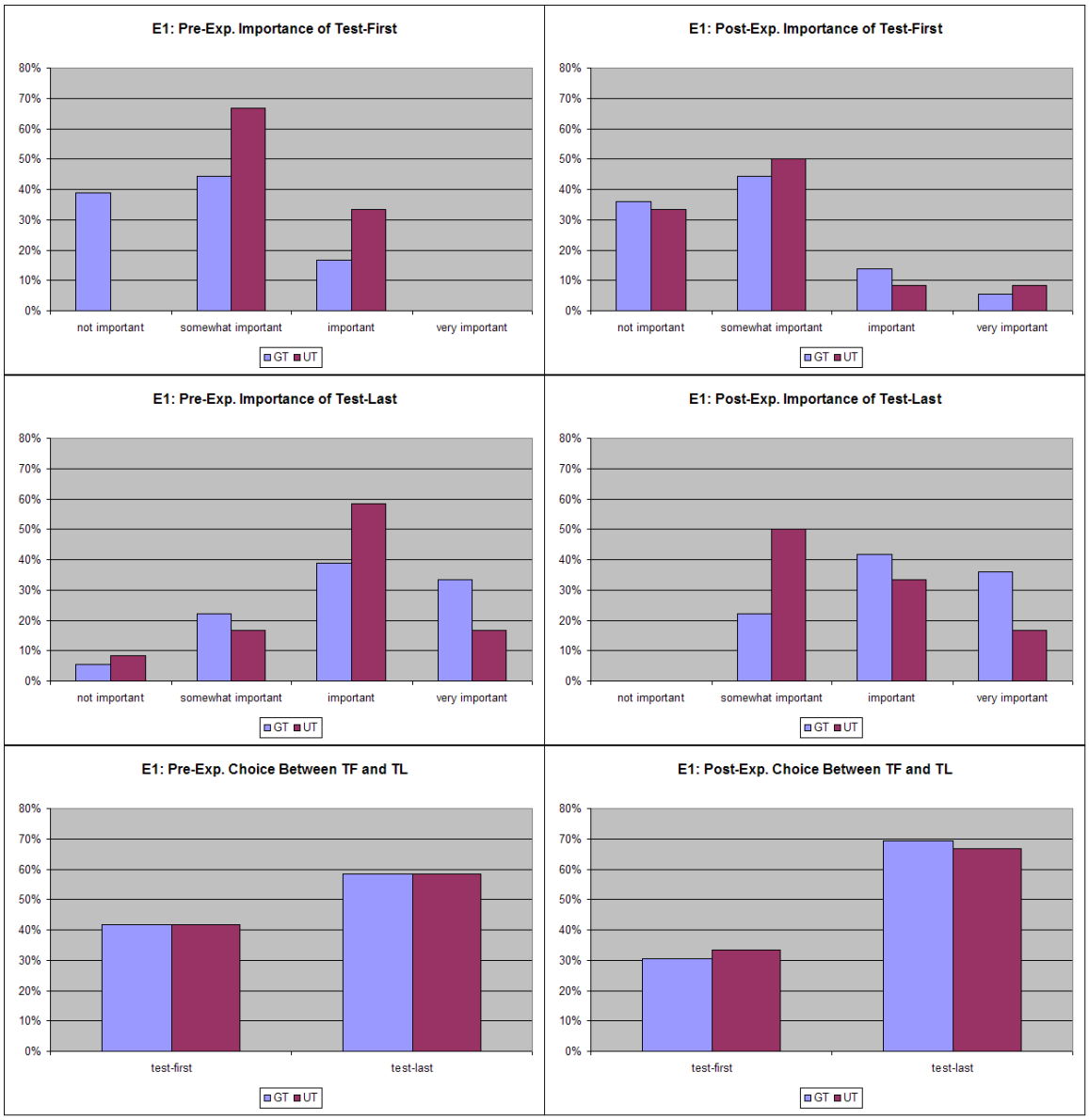

Figure 4.5: Experiment 1: Attitudes Towards Testing

	GT Avg %	UT Avg %	t-Test Val.	Stat. Sig? (p-value = 0.05)
Q1	60.0%	54.4%	0.479	No
Q2	54.7%	51.5%	0.637	No
Q3	72.0%	80.7%	0.315	No
Total	59.2%	57.9%	0.779	No

Table 4.7: Experiment 1 Comprehension: Final Exam Scores

bound, and above the upper bound. There are fifteen possible points for this question, five for each one of three pieces of data. However, the question on the exam provided two sample test cases for each one of the three data pieces, totaling six sample test cases. Therefore, the author graded this problem out of nine possible points, one for each of the additional test cases not already provided.

Question 3
The last question tested design skills of students. They were required to implement a subclass with a parameterized constructor and a method on its superclass's interface. Three possible points could be achieved on this question. One point if the design of the class was set up correctly (i.e. extended the parent class correctly and had correct method signatures). Another point was for proper usage of `super()` calls, and one last point if the logic in the single method was correct.

Final Exam Grade

The total of the three questions outlined above were compared between the GT and UT groups. Average scores for each question and the total of the final exam are outlined in Table 4.7. Overall, the GT group did a little better, but no comparisons were significant.

Code-Testing Question

Question 2 had the students develop a test plan. As seen in Table 4.7, the GT group did slightly better on this question. Thus, they came up with more correct test-cases than the UT group. However, the differences were not significant.

4.1.5 Discussion

When comparing students with graded test-code vs ungraded test-code, a few differences arise. The GT group does spend 20-40% more time on projects than the UT group, but the difference is not statistically significant, partly due to a few outliers. Students can successfully write tests and be exposed to different methods of testing without taking much additional time. As an incentive to write tests, students should be graded on their test-code. Otherwise, the quality and amount of tests written drops as indicated by the code-coverage analysis. Grades and passed unit tests were not much different between the groups. This might be surprising since testing is often correlated with the quality of code. However, in the context of this experiment, all students were required to pass a full acceptance test to submit their code. This test could have very well exposed the UT group to any defects without having to write tests themselves. Both groups had relatively the same attitudes towards testing, and did equally well on the final exam.

Hypotheses **H1-H7** focuses on experiment 1. The number of JUnit tests passed was not significantly different between the groups, so we cannot reject the null hypothesis for **H1**, implying that we cannot accept the alternate hypothesis. Although the time spent by the GT group per project was slightly higher than the UT group, it was not significant in most cases. We therefore cannot reject the null hypothesis for **H2**. As the students wrote more and more unit tests, the quality of tests as measured by code-coverage was significantly higher for the GT group. We therefore accept **H3** alternate hypothesis and reject the null hypothesis. Grades were relatively equal between the two groups, so we cannot reject the null hypothesis for **H4**. Attitudes towards testing and scores on the final exam were relatively the same between both groups, so for hypotheses **H5-H7** we cannot reject the null hypothesis.

4.2 Experiment 2: *Classical-Test vs Test-First*

This experiment compares quality of students' projects in the course taught in Winter 2008 to the projects completed by students who took the course exactly one-year earlier. Students in the Winter 2007 course were not exposed to TDD or JUnit like the students in the Winter 2008 course. The 2007 group is called the classical-test (CT) group, and the 2008 group is denoted the test-first group (TF).

4.2.1 Quality

The quality of the projects was determined using the number of passed JUnit tests and the student's overall project grade. The student's source code was run against an instructor's suite of JUnit tests.

Passed JUnit Tests

Table 4.8 summarizes the percent of JUnit tests passed. As in the data from experiment 1, a student has to pass a given acceptance test to be able to hand in their code. Therefore, we again see high percentages of passed unit tests. The TF group passed a significantly higher number of tests than the CT group for most projects. Projects one and two were introductory projects and are expected to have a low number of defects. However, it is interesting to note that for projects one and seven, the averages for the TF group were lower than that of the CT group, although the differences were not statistically significant. Figure 4.6 shows boxplots for the distributions of passed tests for projects number one and seven. We can see some significant outliers in the TF group. For the first project, these students could have gotten hung up on the new Java language. For the last project, students might have become busy during the end-of-quarter rush and not put enough time to completing the project.

Proj. #	**CT Tests Passed** (avg. % passed)	**TF Tests Passed** (avg. % passed)	***t*-Test Val.**	**Stat. Sig?** (*p*-val = 0.05)
1	99.08%	98.44%	0.566	No
2	95.95%	96.94%	0.448	No
3	94.81%	97.15%	0.004	Yes
4	94.73%	98.34%	0.00012	Yes
5	86.69%	97.22%	0.013	Yes
6	90.00%	96.31%	0.00004	Yes
7	96.15%	93.27%	0.193	No

Table 4.8: Experiment 2 Quality Analysis: JUnit Tests Passed

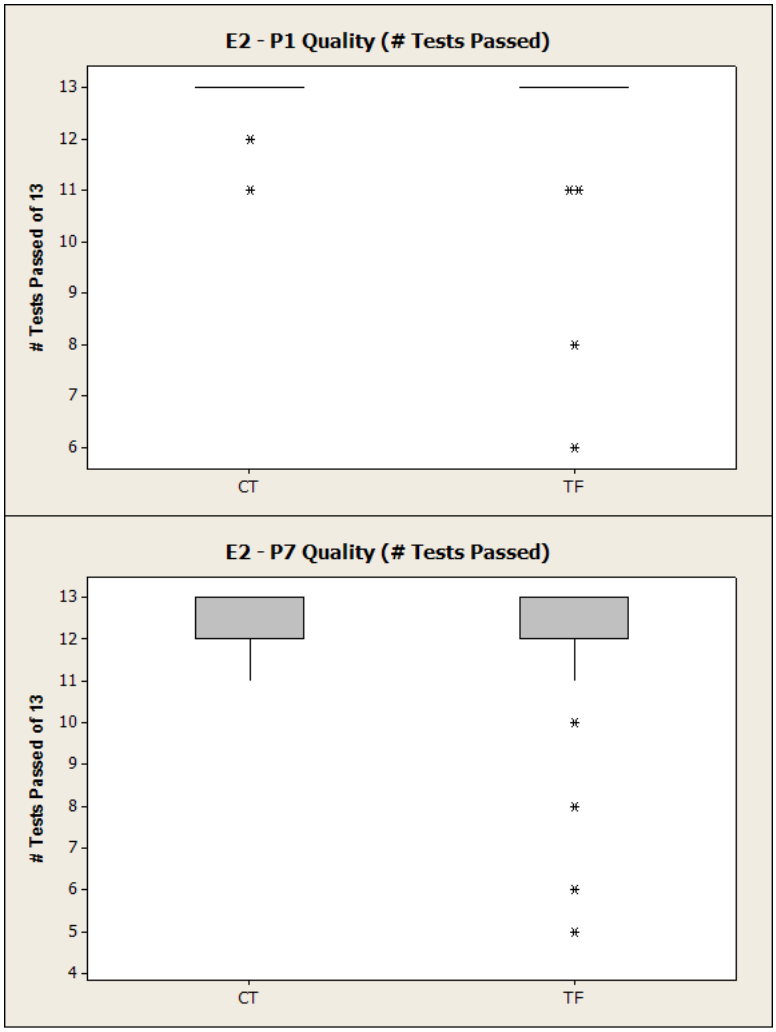

Figure 4.6: Experiment 2: Projects 1 and 7 Boxplots

Proj. #	Avg. CT Grade	Avg. TF Grade	t-Test Val.	Stat. Sig? (p-val = 0.05)
1	94.58%	92.95%	0.407	No
2	88.45%	79.03%	0.0023	Yes
3	89.91%	81.41%	0.0060	Yes
4	87.17%	82.97%	0.136	Yes
5	81.42%	77.14%	0.274	No
6	89.33%	84.46%	0.160	No
7	90.22%	81.33%	0.013	Yes
Course Grade	61.53%	60.12%	0.796	No

Table 4.9: Experiment 2 Quality Analysis: Project Grade

Project Grade

As in experiment 1, project grades were examined in experiment 2 as a measure of quality. Interestingly, Table 4.9 shows the CT group having a higher average grade than the TF group on each project with some differences statistically significant. However, average final course grades between the two groups were nearly identical. This is an opposite trend from the data seen in the percents of JUnit tests passed, where the TF group sometimes averaged significantly higher percentiles than the CT group. It is important to note that several variables exist in this data besides exposure to TDD and JUnit. The CT group received the acceptance test as soon as they started the project, while the TF group got access one day before the project was due. Secondly, test-grades are accounted for in the graded-tests sections of the TF group. These variables add uncontrolled noise to the comparison of project grades.

4.2.2 Discussion

From this experiment we are able to note the importance of exposure to unit testing through JUnit. For the majority of projects, the TF group passed significantly more unit tests than the CT group a year earlier. Too many uncontrolled variables in the project grade analysis are apparent in the data to make any conclusions in the quality analysis.

Hypothesis **H8** addresses the number of passed unit tests between the two groups. We accept the alternate hypothesis for **H8** and reject the null hypothesis. Due to the noise in the project grade data, we will not make any claims to hypothesis **H9**.

4.3 Experiment 3: *Teaching Style*

This experiment sought to find differences in teaching a test-first approach. It compares the effects of instructors using a TDD approach for in-class examples. Professor A did not use a test-first approach for in-class examples, where Professor B strictly followed a design recipe which steps through a test-first process. A common project was given to both groups and results were analyzed.

4.3.1 Productivity

Differences in productivity for the common project are outlined in Table 4.10. Professor B's students spent less time on average than Professor A's group, but the differences were not statistically significant. Furthermore, a boxplot of the data in Figure 4.7 shows several outliers in Professor A's group which could have pulled up the average. The median, 25^{th}-, and 75^{th}-percentiles are very similar. More common projects between the two groups needs to be assessed to make any general conclusions.

4.3.2 Quality

Project submissions were gathered and analyzed for quality by the number of passed unit tests and code-coverage from written tests. Unfortunately, different coding practices between the classes and slight modifications to the API requirements made it difficult to gather enough

	Prof. A Group	Prof. B Group
Average	11.78 hrs	9.68 hrs
Median	10 hrs	8.1 hrs
Std. Dev.	8.42 hrs	5.27 hrs
Maximum	50 hrs	24 hrs
Minimum	1.5 hrs	2 hrs
2-tailed t-test	0.173	
Stat. sig.?	(p-val: 0.05): No	

Table 4.10: Experiment 3 Productivity Analysis: # Hours Worked

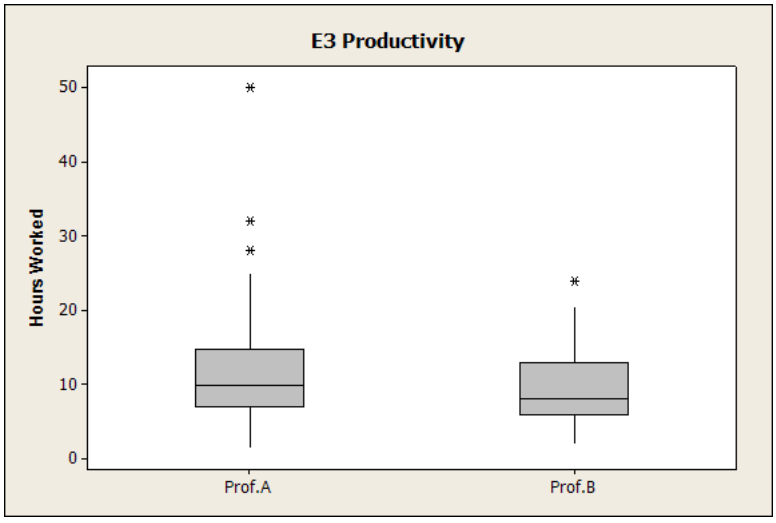

Figure 4.7: Experiment 3: Productivity Boxplots

	A Avg %	B Avg %	t-Test Val.	Stat. Sig? (p-value = 0.05)
Q1	58.5%	46.8%	0.089	No
Q2	53.8%	35.9%	0.00015	Yes
Q3	74.4%	44.7%	$5.17 * 10^{-6}$	Yes
Total	58.8%	39.9%	$7.79 * 10^{-7}$	Yes

Table 4.11: Experiment 3 Comprehension: Final Exam Scores

comparable data. Thus, the author and colleagues had to drop this aspect of experiment 3.

4.3.3 Attitudes

Attitudes towards testing was gathered through surveys as it was in experiment 1. The same questions were analyzed between the A group and the B group as outlined previously in Figure 4.4. Data gathered for the A and B groups is graphically displayed in Figure 4.8. As seen in experiment 1, differences between the groups and within the groups were not significant. Both groups started out split between choosing test-first and test-last. By the end of the experiment they both leaned towards choosing test-last independently of the teaching style taught. Interestingly enough, when taught with a design recipe that enforced test-first, the B group leaned toward test-last. Nonetheless, both groups still valued testing.

4.3.4 Comprehension

Comprehension was measured by looking at the final exams for both groups. Group A's final exams were re-graded in experiment 1, and group B's final exams were likewise re-graded. Scores are outlined in Table 4.11.

Final Exam Grade

Professor A's students averaged much higher on the combined totals of the three questions looked at on the final exam. Looking more in depth, group A did significantly better on

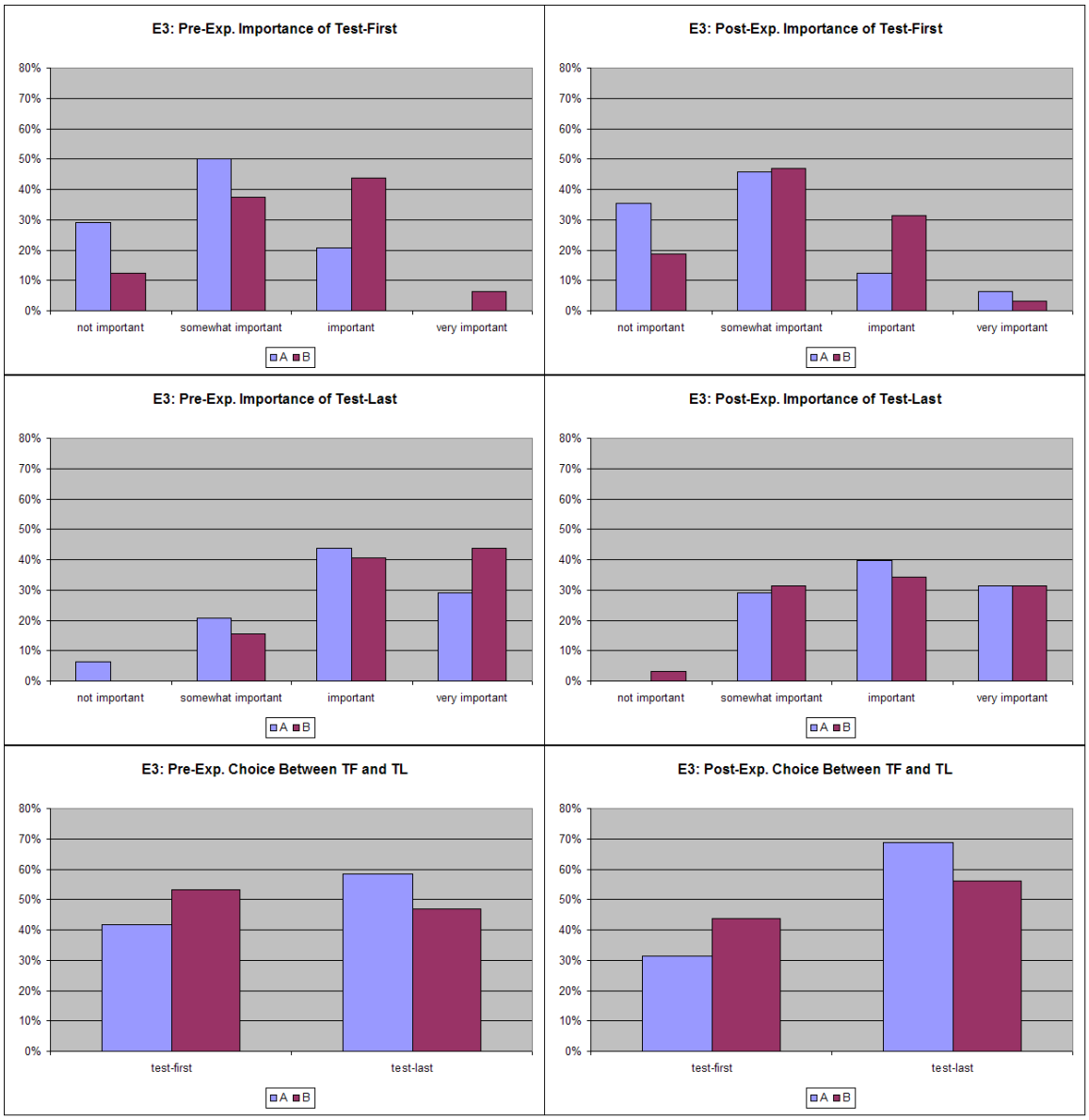

Figure 4.8: Experiment 3: Attitudes Towards Testing

question 2's boundary test-plan and question 3's test of design skills. When broken down into question 3's sub-problems, both groups were about equal on getting the logic of the question correct. Professor A's students did significantly better on getting the design of classes and method signatures correct (t-test val. $= 6.75 * 10^{-5}$) and properly using `super()` in this designed subclass (t-test val. $= 8.95 * 10^{-6}$).

Code-Testing Question

Group A on average came up with a little more than half of all test-cases, where group B came up with about a third. When Professor B was asked about the results, he noted that he did not explicitly teach boundary analysis skills. Group A did significantly better on this boundary test-plan question.

4.3.5 Discussion

While the students in both these groups took about an equal amount of time on the common project and felt relatively the same about testing, group A did better on certain final exam questions. Since the concepts on the final exam were relatively specific (boundary analysis, subclasses and use of `super()`), significant differences should not be correlated only with teaching style. If all labs and projects were the same between both of the groups, a better correlation to teaching style could be drawn. Only one project was common between these two groups, so differences could be attributed to Professor A having projects which more readily relayed concepts such as boundary analysis and extensive use of subclasses and `super()` calls.

Hypotheses **H10-H15** addressed different aspects of experiment 3. Due to the limitation of quality analysis in this experiment, we will not consider hypotheses **H10** or **H12** which have to do with the number of unit tests passed and code-coverage of student tests, respectively.

There were insignificant differences between the number of hours worked on the common project between the two groups. We therefore cannot reject the null hypothesis for **H11**. Both groups had similar attitudes towards testing so we cannot reject the null hypothesis for **H13**. Group A did better overall on the final exam and on the testing question, so we reject both the null and alternate hypotheses for **H14** and **H15**.

4.4 Experiment 4: *Graded Tests vs Ungraded Tests*

Similar to experiment 1, Professor B divided up his two sections based on grading tests. The two groups are also denoted graded-tests (GT) group and ungraded-tests (UT) group.

4.4.1 Quality

Only one metric was used to measure the quality of projects. The metric used was project grades. Table 4.12 shows average grades per project and results from a two-sample t-test. Grade distributions between the two groups hardly differed. In fact, averages were exactly the same in one case. Awarding points for writing tests did not affect the quality of projects for Professor B's students. The similarity between the sections is most likely due to Professor B's teaching approach, explained previously in Section 3.2.3. All of his students were taught in a test-first manner and followed a strict design recipe for developing programs that required writing tests. According to Professor B, giving grades for the tests seemed to have little impact, as all of his students wrote tests for the design recipe.

4.4.2 Attitudes

Survey results from Professor B's GT and UT groups are displayed in Figure 4.9. Both groups valued test-last programming, before and after the experiment. Opinions of test-first

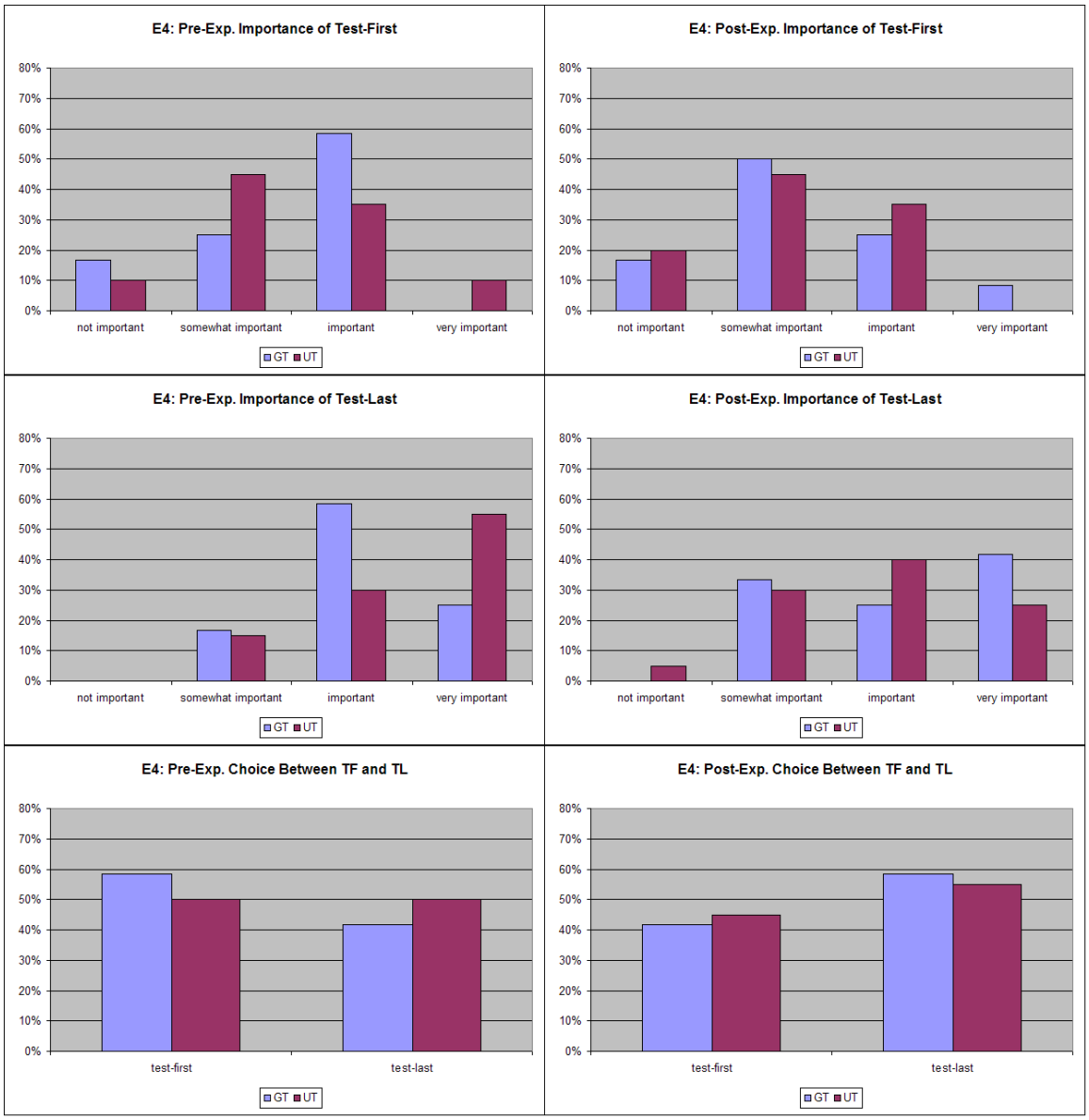

Figure 4.9: Experiment 4: Attitudes Towards Testing

Proj. #	Avg. UT Grade	Avg. GT Grade	t-Test Val.	Stat. Sig? (p-val = 0.05)
1	91.13%	89.97%	0.719	No
2	69.29%	66.24%	0.730	No
3	68.04%	72.41%	0.618	No
4	66.67%	66.67%	1.000	No
5	57.53%	60.65%	0.655	No
6	84.98%	72.12%	0.110	No
7	50.81%	47.83%	0.698	No
8	84.34%	81.45%	0.695	No

Table 4.12: Experiment 4 Quality Analysis: Project Grade

programming stayed mostly the same. No differences in attitudes between the two groups were significant. However, within the UT group, a paired t-test showed that the UT group's feelings towards test-last either stayed the same or dropped (p value = 0.002). The UT group's feelings did not shift toward test-first programming due to the drop in attitudes towards test-last programming, however. Only 25% of the UT students felt more strongly about test-first programming after the experiment, while the rest of the student's attitudes dropped or stayed the same.

4.4.3 Comprehension

Final exam data gathered from Professor B's students in experiment 3 was divided into the GT and UT groups within his classes. This data was then compared to measure any comprehension differences between the two groups.

Final Exam Grade

Both sections did relatively the same on each of the questions on the final exam. Table 4.13 outlines the averages and documents the statistical tests performed. The UT group did slightly better on each question, but the differences were not significant.

	GT Avg %	UT Avg %	t-Test Val.	Stat. Sig? (p-value = 0.05)
Q1	40.0%	54.5%	0.222	No
Q2	35.1%	36.9%	0.797	No
Q3	40.0%	50.0%	0.325	No
Total	37.1%	43.0%	0.321	No

Table 4.13: Experiment 4 Comprehension: Final Exam Scores

Code-Testing Question

On the boundary analysis test-plan question, the groups performed nearly identically. This is most likely due the lack of proper boundary testing taught in the course by Professor B. The UT group only did slightly better for reasons unknown.

4.4.4 Discussion

Awarding points for writing test-code had no significant differences within the style of teaching by Professor B. All of his students wrote tests to follow the specific design recipe used in the course. Both groups scored relatively the same on projects and the final exam. The only real difference to note is that the UT group's attitudes towards test-last programming stayed the same or dropped by the end of the course.

Four hypotheses **H16-H19** were concerned with experiment 4. Grades per project did not differ significantly between groups and scores on the final exam were relatively equal, so we cannot reject the null hypotheses for **H16**, **H18**, and **H19**. While some attitudes towards test-last programming differed within the UT group, they were not significantly different between groups. Therefore, we also cannot reject the null hypothesis for **H17**.

Chapter 5

Conclusions & Future Work

Four experiments were conducted to see effects of incorporating TDD into introductory-level programming courses. More specifically, five concerns with teaching TDD were addressed:

- **C1**: The cost of using existing materials in a TDD fashion was minimal. All projects were directly developed in a TDD approach, and most labs were also directly converted. Labs that originally were designed with I/O were changed to use unit tests for checking expected outputs. I/O was deferred to later in the course.
- **C2**: Awarding points for test-code did not significantly change quality of source-code, time spent on projects, attitudes towards testing, or overall comprehension of material. It did however, give students incentives to write higher-quality code as measured through code-coverage.
- **C3**: Students spent more time per project when developing tests in tandem with source-code, but increases were not significant. Students were able to learn and write unit tests without significantly increasing their work load.
- **C4**: For the scale of projects in introductory-level programming courses, TDD did not affect the quality of projects turned in by students as measured by project grades and

number of passed JUnit tests.

- **C5**: Different teaching approaches did not significantly change the time students spent on projects, attitudes towards testing, or course comprehension. Due to the limitation in our study, project quality aspects were not able to be analyzed.

5.1 Summary of Contributions

Empirical evidence was gathered to address the goals of the experiment. We were able to get the students to write tests and learn a testing framework (JUnit) with minimal changes to existing materials and without significantly increasing the number of hours spent on each project. It is important to reflect on a few of the TDD challenges outlined in Section 2.4.

- *Challenge 1*: Some claim that introductory students are not ready for testing until they have basic programming skills.

  Response: By exposing testing through example and slowly requiring students to write a greater percentage of the tests per project, we were able to prepare the students to write tests completely on their own.

- *Challenge 2*: Instructors often worry about not having enough lecture hours to teach a new topic like software testing.

  Response: We were able to reuse existing materials and have students develop them in a TDD approach without increasing the work load of students. Thus, the students were exposed to unit testing.

- *Challenge 3*: Staff often have their hands full accessing code-correctness so it may not be feasible to assess test cases as well.

  Response: With automated testing frameworks like JUnit and code-coverage tools like Emma, the number of tests and quality of tests can be automatically calculated and

factored into a grade with only a one-time setup cost. Professor A voluntarily continued using the new TDD and JUnit materials in the subsequent quarter.

5.2 Future Work

In experiment 1, the students were only exposed to TDD and testing through descriptions of how the process works and sample JUnit code with simple comments. A set of 1-3 labs needs to be developed to formally introduce students to the TDD process and basic syntax of JUnit. These labs were purposely not developed prior to the experiment to see the cost of reusing existing labs and projects and tacking on a TDD approach (concern **C1**).

A set of self-contained web-based labs should be built that can emphasize benefits of TDD and allow students to get hands-on experience in a simplified lab setting. Currently, the author has developed two such labs, available online[1]. The first lab exposes students to syntax and semantics of JUnit, and the second introduces TDD and provides a step-by-step walk-through of the process. Web-CAT is used as a back-end automated grader that can give students feedback on their assignments. Web-CAT has been used for programming assignments and is currently being used to encourage a test-first approach [6, 7, 8], and therefore seems like an appropriate candidate. Empirical evidence needs to be gathered to access the effectiveness and quality of these labs.

[1] http://users.csc.calpoly.edu/~djanzen/research/TDD08/cdesai/

Bibliography

[1] S. Ambler. Test-Driven Development is the Combination of Test First Development and Refactoring. *Dr. Dobb's Agile Newsletter*, June 12, 2006.

[2] E. Barriocanal, M. Urban, I. Cuevas, and P. Perez. An Experience in Integrating Automated Unit Testing Practices in an Introductory Programming Course. *ACM SIGCSE Bulletin*, 34(4):125–128, December 2002.

[3] K. Beck. *Test Driven Development: By Example.* Addison Wesley, November 2002.

[4] K. Bruce, A. Danyluk, and T. Murtagh. A Library to Support a Graphics-Based Object-First Approach to CS1. In *SIGCSE '01: Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, pages 6–10. ACM, 2001.

[5] S. Edwards. Rethinking Computer Science Education from a Test-First Perspective. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 148–155. ACM, 2003.

[6] S. Edwards. Using Test-Driven Development in the Classroom: Providing Students with Automatic, Concrete Feedback on Performance. In *Proc. Int'l Conf. on Education and Information Systems: Technologies and Applications (EISTA)*, August 2003.

[7] S. Edwards. Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. *ACM SIGCSE Bulletin*, 36(1):26–30, March 2004.

[8] S. Edwards and M. Pérez-Quiñones. Experiences using Test-Driven Development with an Automated Grader. *J. Comput. Small Coll.*, 22(3):44–50, 2007.

[9] H. Erdogmus, M. Morisio, and M. Torchiano. On the Effectiveness of the Test-First Approach to Programming. *IEE Trans. Software Eng.*, 31(3):226–237, March 2005.

[10] D. Janzen and H. Saiedian. Test-Driven Development: Concepts, Taxonomy, and Future Direction. *IEEE Computer*, 38(9):43–50, September 2005.

[11] D. Janzen and H. Saiedian. Test-Driven Learning: Intrinsic Integration of Testing into the CS/SE Curriculum. In *Proc. 37th Technical Symposium on Computer Science Education (SIGCSE)*, pages 254–258. ACM, 2006.

[12] D. Janzen and H. Saiedian. A Leveled Examination of Test-Driven Development Acceptance. In *Proc. 29th Int'l Conf. on Software Engineering (ICSE)*, pages 719–722. IEEE Press, 2007.

[13] D. Janzen and H. Saiedian. Test-Driven Learning in Early Programming Courses. In *Proc. 38th Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 2008.

[14] R. Jeffries and G. Melnik. TDD: The Art of Fearless Programming. *IEEE Software*, 24(3):24–30, May-June 2007.

[15] C. Jones. Test-Driven Development Goes to School. *Journal of Computing Sciences in Colleges*, 20(1):220–231, October 2004.

[16] R. Kaufmann and D. Janzen. Implications of Test-Driven Development: A Pilot Study. In *Companion of the 18th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 298–299. ACM Press, 2003.

[17] K. Keefe, J. Sheard, and M. Dick. Adopting XP Practices for Teaching Object Oriented

Programming. In *Proc. 8th Australian Conf. Computing Education*, volume 52, pages 91–100, 2006.

[18] C. Larman and V. Basili. Iterative and Incremental Developments. A Brief History. *IEEE Computer*, 36(6):47–56, June 2003.

[19] L. Madeyski. Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality. *Software Engineering: Evolution and Emerging Technologies*, 130:113–123, 2005.

[20] G. Melnik and F. Maurer. A Cross-Program Investigation of Students' Perceptions of Agile Methods. In *Proc. 27th Int'l Conf. on Software Eng. (ICSE)*, pages 481–488. ACM Press, 2005.

[21] M. Müller and O. Hagner. Experiment About Test-First Programming. *IEE Proc. Software*, 149(5):131–136, October 2002.

[22] M. Müller and W. Tichy. Case Study: Extreme Programming in a University Environment. In *Proc. 23th Int'l Conf. on Software Eng. (ICSE)*, pages 537–544, May 2001.

[23] M. Pancur, M. Ciglaric, M. Trampus, and T. Vidmar. Towards Empirical Evaluation of Test-Driven Development in a University Environment. In *IEEE Region 8 Proc. EUROCON*, volume 2, pages 83–86. IEEE Press, September 2003.

[24] T. Shepard, M. Lamb, and D. Kelly. More Testing Should be Taught. *Commun. ACM*, 44(6):103–108, 2001.

[25] J. Spacco and W. Pugh. Helping Students Appreciate Test-Driven Development (TDD). In *Companion to 21st. ACM SIGPLAN Conf. Object-Oriented Prog. Systems, Languages, and Applications (OOPSLA)*, pages 907–913, 2006.

[26] M. Thornton, S. Edwards, R. Tan, and M. Pérez-Quiñones. Supporting Student-Written

Tests of GUI Programs. In *SIGCSE '08: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. ACM, 2008.

[27] S. Yenduri and L. Perkins. Impact of Using Test-Driven Development: A Case Study. *Software Engineering Research and Practice*, pages 126–129, 2006.