Implications of Integrating Test-Driven Development into CS1/CS2 Curricula

Chetan Desai
Intuit Inc.
San Diego, California USA
chetan_desai@intuit.com

David S. Janzen, John Clements
California Polytechnic State University
San Luis Obispo, California USA
{djanzen,clements}@calpoly.edu

ABSTRACT

Many academic and industry professionals have called for more testing in computer science curricula. Test-driven development (TDD) has been proposed as a solution to improve testing in academia. This paper demonstrates how TDD can be integrated into existing course materials without reducing topic coverage. Two controlled experiments were conducted in a CS1/CS2 course in Winter 2008. Following a test-driven learning approach, unit testing was introduced at the beginning of the course and reinforced through example. Results indicate that while student work loads may increase with the incorporation of TDD, students are able to successfully develop unit tests while learning to program.

Keywords

Test-driven learning, test-driven development, cs1, cs2

1. INTRODUCTION

Industry typically spends 50% or more of software project resources on testing. Despite this, students rarely learn the value of testing in early programming courses, or indeed at any point in the computer science curriculum [12]. The result is students that are ill-prepared for work in the software industry. Unfortunately, proposals to add courses on testing may be infeasible due to curriculum constraints at many universities, and even if feasible, may reinforce misperceptions that testing is somehow different or separate from the rest of computing. The alternative is to incorporate testing into existing courses, using test-driven development (TDD).

In TDD [6], students specify tests before implementing the corresponding program fragment. This gives the students the chance to consider the desired result of the program fragment before plunging into the implementation details, and guarantees that the students can validate their newly written code immediately upon its completion.

Edwards [4] identified several concerns when adopting TDD practices in a university environment. First, many instructors believe that introductory students are not ready for testing until they have basic programming skills. Second, instructors feel that they do not have enough lecture hours to teach a new topic like software testing. Third, course staff already has its hands full grading code correctness, so it may not be feasible to assess test cases too.

Fortunately, our experiments suggest that by introducing testing gradually and by leveraging the students' tests to help simplify grading, adding TDD to an introductory course may be less traumatic than these concerns would suggest. This paper reports on a set of techniques for introducing TDD into existing course materials, and the implications thereof.

2. TRANSFORMING CS1/CS2

It can be difficult to determine when and how to introduce TDD practices into a curriculum. Most experiments reported in the literature[9, 6, 2] introduced TDD at the beginning of sophomore through graduate level courses. Introductions usually consisted of general explanations of unit testing and TDD, documentation on a test harness (e.g. JUnit), and examples of how to write test cases, execute test cases, and interpret results. Introduction lengths varied from a thirty-minute lecture[3] to a three-week topic[11].

The goal of this study is to obtain empirical evidence on the cost and reward of integrating TDD into a first-year programming course. What kind of curriculum changes are required? How drastically must existing projects change? What are the effects on the students? In order to gain acceptance from skeptical peers, we elected to minimize the changes to the existing course, and to perform controlled experiments to measure the results of the change in student code quality, productivity, and test coverage.

Under the guidance of two faculty, a first year master's student and a senior undergraduate student modified lab and project descriptions from a Winter 2007 non-TDD-based

course at California Polytechnic State University, San Luis Obispo. The course, CSC102, was the second in a three-course first-year sequence. Each course lasted ten weeks. The first course in the sequence teaches the basics of programming in the C language. This second course introduces object-oriented programming in the Java language. The team modified the lab and project descriptions to encourage a TDD approach to development. Nearly all of the topics and structure of the programming assignments stayed the same (exceptions are noted below).

In order to introduce testing from the beginning of the course without burdening the students, students were given full JUnit test suites for projects and labs early in the course. 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. The concrete examples of test methods in the *Triangle* class led the students to similar tests for the corresponding method in the *Rectangle* class. Later projects removed the "training wheels" by requiring the students to write all tests themselves.

Also, students learned the value of reusable automatic unit tests as projects 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 provided a test suite to ensure the student did not break any functionality in the process. The labs and projects are all available online[1].

3. EVALUATION

We evaluated three questions regarding the introduction of TDD into a first year programming course.

First, can TDD be integrated into early programming courses with minimal effort on the part of instructors?

Second, what effect does the grading of test-code have on students' tests? 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 [1] 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.

Third, what effects does TDD have on quality of code and productivity of students? 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?

3.1 TDD Integration Cost

Two student researchers revised six of the original nine weekly lab assignments and eight project assignments from a previous offering of the CSC102 course. The course materials were prepared for an instructor who had taught the CSC102 course for several years. This instructor had no experience with TDD and did not adjust his lectures to include TDD instruction. As a result, the instructor spent minimal time integrating TDD into this course.

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

One student researcher was tasked with revising the labs to include instruction on TDD and JUnit, developing test suites for the labs, and developing new grading scripts for both the labs and projects. He spent a total of seventy-two hours on these tasks. The researcher had limited prior TDD experience and no curriculum development experience. As might be expected, the first three labs required more changes than the labs that came later in the course. In fact two of the later labs required no changes at all. The most significant changes were made to the second lab. The original second lab introduced console input/output (I/O) through the Scanner and System.out classes. This lab involved reading input from the console, parsing it, and identifying chunks that were `ints`, `doubles`, and `strings`. The new lab was proposed to preserve the parsing and type identification components while dropping the I/O aspects of the lab. I/O is traditionally difficult to test automatically so the researchers suggested delaying I/O to later in the course. Upon the instructor's insistence, the second lab eventually combined the two approaches, requiring students to write a testable parser and type identifier as well as an I/O driver.

The second student researcher was tasked with revising the project descriptions to include TDD requirements, developing test suites (both student and instructor versions), and developing grading criteria. In addition this student planned the experiment design and gathered all associated artifacts and metrics. This student spent a total of 126 hours on these tasks.

The instructor reported no decrease in coverage of course topics nor significant extra time required in lecture or lab in order to include the TDD material. Despite his initial skepticism, the instructor embraced TDD and voluntarily opted to use the new TDD-based lab and project materials in the subsequent quarter, rather than his original non-TDD-based materials.

3.2 Effects of Grading Student Test Code (E1)

We examined three sections of the Winter 2008 CSC102 course in Experiment 1 (E1). Eighty-three students enrolled in three sections with 24, 29, and 30 students respectively. A single instructor taught all of the sections using the same labs, projects and lectures as described previously.

Two sections of the course, totalling 59 students, were randomly chosen as the experimental group. This group's tests were graded for 10% of their assignment grade, and the students were told of this. We call this group the "graded tests group" (GT). The other section of the course (24 students) was the control group. Their grades did not depend on their test cases, and they were also told of this. We call this group the "ungraded tests group" (UT).

Experiment 1 sought to discover the effects of grading students on their test code versus not awarding points to test code. We considered three dependent variables: productivity, quality, and comprehension of the course material.

3.2.1 Productivity

Table 1 displays the average number of hours that students worked on the given project. 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

Proj. #	Avg Hrs Worked UT	GT	p-val	Sig? (p <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 1: Experiment 1 Productivity

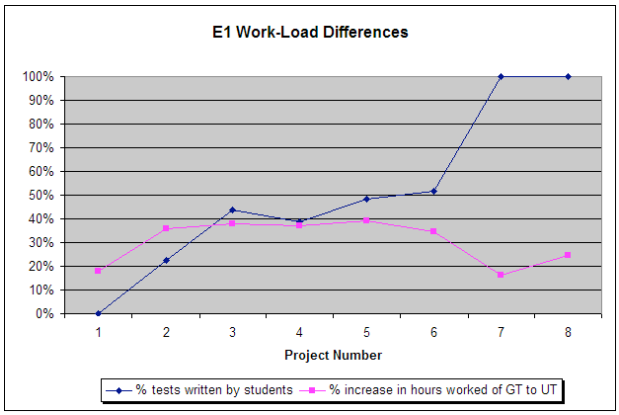

Figure 1: Experiment 1 Work-Load Differences

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 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. Project 8 was optional, but most students completed it for extra credit. 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.

Figure 1 compares the fraction of extra time taken by the GT group in each project to the fraction of methods whose test cases were not provided by the teacher. For the very first project, when neither group had to write tests, we see the GT group taking 18% longer than the UT group. This suggests that the GT group has 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 was required of them. However, by the end of the course, the trend suggests 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.

Proj. #	Avg % Tests Passed UT	GT	p-val	Sig? (p <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 2: Experiment 1 JUnit Tests Passed

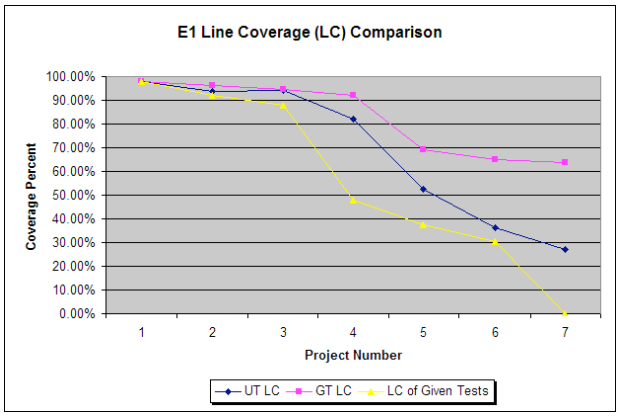

Figure 2: Experiment 1: Line Coverage Comparison

3.2.2 Quality

The quality of the projects was determined using the number of passed JUnit tests, the subjective score assigned by the instructor, and code coverage. We ran the student's source code against an instructor's suite of JUnit tests to determine the number of passed JUnit tests. Table 2 summarizes the percent of JUnit tests passed. Students were not provided a fixed API for project 8 so test data is excluded for that project. We ran students' tests on the instructor's source code to compute code coverage.

To receive credit for each project, a student had to pass all the tests in an acceptance test suite provided one day before the due date. It is therefore no surprise that the average number of JUnit tests passed was above 92% for all projects and around 98% for most projects. Using a two-tailed t-test with a p-value threshold of 0.05, we detected no significant difference in the number of unit tests passed by the GT and UT groups. Similarly, the instructor's subjective grades did not differ significantly between the two groups. Based on the instructor's policy of providing acceptance tests and requiring 100% pass rates in order to get any project credit, the use or non-use of TDD had no effect on project grades.

We measured the quality of student-written tests using code coverage. This measure is computed by dividing the number of lines of code evaluated at least once during evaluation of the test suite by the total number of lines in the code.

Figure 2 plots line coverage percentages for the groups compared to line coverage of the given tests. Line coverage is a common form of code-coverage that reports the lines of source code executed during the tests as a percentage of total lines of source code. For the first four projects, it looks

Proj. #	Avg % Tests Passed CT	TF	p-val	Sig? (p <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 3: Experiment 2 JUnit Tests Passed

as if all the students kept up great coverage percentages as the coverage 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. In projects five through seven, the differences between the code coverage measures of the two groups are statistically significant (proj. 5 $p = 0.005$, proj. 6 $p = 0.0004$, proj. 7 $p = 0.009$).

The instructor graded tests based on method coverage, so as long as they achieved 100% method coverage, they received full marks. Nonetheless, 60% line coverage is not bad for introductory students. Grading based on line coverage might ensure higher quality tests.

3.2.3 Comprehension

To measure comprehension of course material, we compared the exam scores of the GT and UT groups. Using a t-test with a p-value of 0.05, we detected no significant difference between the exam scores of the GT and UT groups.

3.3 Effects of TDD/JUnit Exposure (E2)

The second experiment, labeled E2, compares the 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. Unlike the students in Winter 2008, students in Winter 2007 were not exposed to TDD or JUnit. The 2007 group is called the classical-test (CT) group, and the 2008 group is denoted the test-first group (TF).

3.3.1 Quality

We determined the quality of the projects using the number of passed JUnit tests and the student's overall project grade. Because the project requirements and policies were the same in both years (outside of the JUnit requirements), the student's source code was run against an instructor's suite of JUnit tests.

Table 3 summarizes the percent of JUnit tests passed. Because a student was required to pass all provided acceptance tests in order to receive a nonzero correctness score, the number of passed unit tests was again high. However, the TF group passed a significantly higher number of tests than the CT group for projects 3, 4, 5, and 6. The CT group did have slightly higher percentages of tests passed on projects 1 and 7, but the differences were very small.

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.

4. THREATS TO VALIDITY

Our experiments contained both internal and external threats to validity. The internal threats are specific to the experiments, but the external threats are shared.

In experiment 1, students self-selected the sections that they were in, though they had no advance knowledge of the experiment. It is therefore possible that some of the class sections might have been biased by external factors. For instance, 42.3% of the students in the UT group reported having a GPA over 3.0, where the GT group had 56.9%.

Experiment 2 considers the hypothesis that adding testing to the course could affect the students' performance on the projects. However, the addition of the skeletal test suites could have altered the difficulty of the projects.

There are a set of external threats common to the two experiments; most significantly, this instructor decided to release the acceptance test suite to the students a day before the assignment was due. Also, this study concerns a set of students coming out of a quarter-long course in C, taught by an instructor without prior experience in teaching the use of TDD. Results in other scenarios will naturally differ, though we see the experiment conditions as being moderately biased against the relative success of the test-based learning, and we would expect others' results to be–if anything–more positive than ours.

5. RELATED WORK

This work is believed to be the first to demonstrate that TDD can be integrated into a first-year programming course without reducing topic coverage, reducing student code quality, or severely increasing student effort. The test-driven learning (TDL) [7] pedagogical approach that was applied takes a reinforced learning approach in which TDD is not taught as a separate topic, but is incorporated into traditional computing topics and reinforced through subsequent examples. The efficacy of TDL in CS1 and CS2 courses was demonstrated previously[8], finding that test-first programmers wrote more tests and scored higher on project grades than their test-last counterparts when taught in a TDL fashion. However, these original CS1/CS2 experiments covered only short time periods (two or three weeks), involved completely new course materials, and the author was also the instructor, and therefore potentially biased.

A reinforced learning approach such as TDL could be key to successfully introducing TDD. In cases where students were just briefly introduced to testing at the start of the semester, TDD was not preferred[10] and only 10% of the students wrote test cases[1].

Numerous TDD-based controlled experiments have been reported and surveyed[9, 6, 2]. However, nearly all of the controlled experiments looked at sophomore- to graduate-level classes, not CS1 or CS2 courses which present unique challenges. As Keefe et al. note[10], students starting to learn what programming is and how it works find it tough to find purpose in the code, so testing it is difficult.

Erdogmus [5] 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 number of tests written, independent of the development strategy used.

Yenduri and Perkins [13] 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.

Edwards [4] conducted an experiment in a junior-level course with 118 students at Virginia Tech University. In this course, TDD practices were introduced briefly at the start of the semester, but then used in the classroom throughout the entire experiment to model behavior. This course also used an automated grading system named Web-CAT (http://web-cat.cs.vt.edu/) 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.

6. CONCLUSIONS AND ADVICE

Test-driven learning was applied to a first-year course to demonstrate that traditional course materials could be adapted to incorporate TDD with minimal impact. 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. We were able to reuse existing materials and have students develop them in a TDD approach without increasing the work load of students. Outside of the one-time setup cost, instructor effort is also not increased with TDL, and may in fact decrease thanks to the opportunity of automated grading.

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. Also, 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.

The authors believe that simply rewriting course materials to incorporate TDD, while effective, is not the ideal situation. Rather, some re-ordering and re-emphasis of topics is recommended. For instance, we believe that TDD-based courses should delay and de-emphasize instruction on input/output. Many interesting projects can be assigned with tests and perhaps instructor-provided user interfaces that avoid the details and challenges of input/output.

Using TDD addresses both testing and design skills of beginning programmers. 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, we believe they are likely to realize that testing is an integral part of programming, benefitting them throughout their academic and professional careers.

7. ACKNOWLEDGEMENTS

Special thanks to Lockheed Martin for their generous support of this research.

8. REFERENCES

[1] 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.

[2] C. Desai, D. Janzen, and K. Savage. A survey of evidence for test-driven development in academia. *ACM SIGCSE Bulletin*, 40(2):97–101, 2008.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

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

[10] 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.

[11] 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.

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

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