A Survey of Evidence for Test-Driven Development in Academia

Chetan Desai, David Janzen, Kyle Savage
Computer Science
California Polytechnic State University
San Luis Obispo, California USA
{cdesai, djanzen}@calpoly.edu
savage@alumni.calpoly.edu

Abstract: University professors traditionally struggle to incorporate software testing into their course curriculum. Worries include double-grading for correctness of both source and test code and finding time to teach testing as a topic. Test-driven development (TDD) has been suggested as a possible solution to improve student software testing skills and to realize the benefits of testing. According to most existing studies, TDD improves software quality and student productivity. This paper surveys the current state of TDD experiments conducted exclusively at universities. Similar surveys compare experiments in both the classroom and industry, but none have focused strictly on academia.

1. INTRODUCTION
The idea of test-driven development (TDD) has been around since the early 1960’s with NASA’s Project Mercury[15]. 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.

1.1 The TDD Process
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. As larger modules (entire classes or packages, as opposed to single methods) are completed, more tests may be added. The programmers now have a full regression test suite to run whenever changes are made to the system. Design changes can be made with confidence, since if something breaks in another part of the system, the regression tests are likely to catch it.

1.2 Common Misconceptions
There are many misconceptions about test-driven development. First, TDD is not a testing technique, it is a 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 misconceptions in the following list[1]:
   • You create a 100% regression test suite: It is not always cost-effective to achieve 100% test coverage with all code (e.g. user interfaces).
   • Unit tests form 100% of your design specification: Some design documentation is usually valuable and necessary even with TDD.
   • You only need to unit test: A quality product also requires acceptance, performance, system integration and other testing techniques.
   • TDD does not scale: Large test suites can be divided in order to achieve reasonable test execution times.
2. CURRENT STATE OF THE ART

Table 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 quality was the number of unit tests passed during acceptance tests. 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 were measured through pre- and post-experiment surveys.

2.1 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 [19]. 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% [22] and 45% [5] reduction in defects. Changes in productivity varied by experiment. Some experiments found vast improvements in productivity, between 24.5% [22] and 50% [13]. Others found less hopeful results of a 5-10% decrease in productivity [11]. Surveys from students have indicated an increase in program understanding [18] and confidence in making changes to the code and code correctness [19]. These results tended to be more positive in more 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.

2.2 TDD Worries

Adopting TDD practices in a university environment comes with several concerns. Edwards outlines five perceived roadblocks [5]:

1. Introductory students are not ready for testing until they have basic programming skills.
2. Instructors do not have enough lecture hours to teach a new topic like software testing.
3. Course staff already has its hands full grading code correctness, so it may not be feasible to assess test cases too.
4. To learn the benefits of TDD, students need frequent, concrete feedback on how to improve as they are working.
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 [14].

2.3 Popular Frameworks

All of the examined experiments in Table 1 used Java except one, which used Pascal [2]. By far, the most widely used language for TDD is Java, along with JUnit, its popular test harness. JUnit was developed by the inventor and advocate of TDD, 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 was popularized by providing 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.

2.4 TDD at Different Experience Levels

A current pedagogical concern of 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 [14]. Tools like WebCAT [4] and Marmoset [21] have helped overcome testing hurdles. By providing feedback such as test coverage and number of unit tests passed, writing tests becomes meaningful to a programming novice. This helps to eliminate the need for counterproductive practices, such as forcing beginners to write tests as a part of their grade. Students writing tests in this manner 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].
### Table 1: Comparison Grid

<table>
<thead>
<tr>
<th>Study</th>
<th>Type</th>
<th>Student Level</th>
<th>Subjects</th>
<th>Productivity of Students</th>
<th>Quality of Programs</th>
<th>Other Findings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Muller [19]</td>
<td>Case Study</td>
<td>Graduate</td>
<td>11</td>
<td></td>
<td></td>
<td>87% stated regression testing increased confidence</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Minimum quality increased linearly with number of tests.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TDD students felt more confident in their code w.r.t. quality, change, and reuse.</td>
</tr>
<tr>
<td>Madeyski [16]</td>
<td>Cont. Exp.</td>
<td>Graduate</td>
<td>188</td>
<td></td>
<td></td>
<td>Increased student confidence</td>
</tr>
<tr>
<td>Muller [18]</td>
<td>Cont. Exp.</td>
<td>Graduate</td>
<td>19 (10 TDD / 9 Control)</td>
<td>Faster but not stat. sig.</td>
<td>Less reliable w.r.t. passed assertion tests but not stat. sig.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Better program understanding w.r.t. code reuse.</td>
</tr>
<tr>
<td>Pancur [20]</td>
<td>Cont. Exp.</td>
<td>Senior</td>
<td>34 (19 TDD / 15 Control)</td>
<td>2.5% slower</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yenduri [22]</td>
<td>Cont. Exp.</td>
<td>Senior</td>
<td>18 (9 TDD / 9 Control)</td>
<td>25.4% faster</td>
<td>34.8% fewer defects</td>
<td></td>
</tr>
<tr>
<td>Barriocanal [2]</td>
<td>Exp. Report</td>
<td>Freshman</td>
<td>100</td>
<td></td>
<td></td>
<td>Only 10% students wrote test cases by choice</td>
</tr>
<tr>
<td>Melnik [17]</td>
<td>Exp. Report</td>
<td>Freshman – Graduate</td>
<td>240</td>
<td>78% agreed on improvement</td>
<td>76% agreed on improvement</td>
<td>Correlation between age and attitude towards TDD.</td>
</tr>
<tr>
<td>Janzen [9]</td>
<td>Field Study</td>
<td>Freshman – Graduate</td>
<td>160 (130 beginners / 30 mature)</td>
<td>Increased or no effect</td>
<td>Improved or no effect</td>
<td></td>
</tr>
<tr>
<td>Janzen [7]</td>
<td>Survey</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>87% mature programmers prefer TDD, 86% beginner programmers prefer test-last.</td>
</tr>
<tr>
<td>Jones [12]</td>
<td>Survey</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**

- NLOC: non-commented lines of code.
- Cont. Exp.: controlled experiment.
- CCCC: C and C++ Code Counter.
- Stat. Sig.: statistically significant.
- w.r.t.: with respect to TDL: test-driven learning.

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 [17, 9]. As seen in Table 1, most of the success stories come from experiments conducted between junior undergraduate and graduate levels of education. Does this mean that TDD should not be used in introductory programming courses? No. It means that a lot more work needs to go into course plans.

### 3. INTRODUCING TDD

Determining when and how to introduce TDD practices into a curriculum can be difficult. Most of the experiments introduced TDD at the beginning of the semesters. Introductions usually consisted of:
• Explaining automated unit testing
• Describing TDD
• Providing documentation
• Supplying examples of how to write test cases, execute test cases, and interpret results

Introduction lengths varied from a thirty-minute lecture[4] to a three-week topic[19]. Looking back on Table 1, promising results came from Edwards[5], 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 a 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[14] and only 10% of the students wrote test cases[2].

4. 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)[8] proposes teaching by example, presenting examples with automated tests, and starting with tests. TDL was proposed in SIGCSE 06 as a pedagogical tool for incorporating automated unit testing in computer programming courses. 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 [8], 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 [10] 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.

5. CONCLUSIONS AND FUTURE STUDIES

TDD in academia has moved on from its conception stage, and many studies have tried to prove correlations and effects. Studies show that TDD exposes students to analytical and comprehension skills needed in software testing. As a programming technique more than a testing process, TDD tends to help students with the design of complex projects, and increases student confidence. Controlled experiments can help determine when to introduce TDD in education, by using the independent variable of student class-level. Once an appropriate class-level is determined, more experiments should be conducted which identify optimal teaching plans, feedback mechanisms, and test harnesses for that level. Test-driven development reveals valuable software testing skills to fledgling programmers; the next step is figuring out how and when to introduce it into a curriculum.

REFERENCES