A Leveled Examination of Test-Driven Development Acceptance

David S. Janzen
Cal Poly at San Luis Obispo
djanzen@csc.calpoly.edu

Hossein Saiedian
University of Kansas
saiedian@eecs.ku.edu

Abstract

Test-driven development (TDD) has garnered considerable attention in professional settings and has made some inroads into software engineering and computer science education. A series of leveled experiments were conducted with students in beginning undergraduate programming courses through upper-level undergraduate, graduate, and professional training courses. This paper reports that mature programmers who try TDD are more likely to choose TDD over a similar test-last approach. Additionally this research reveals differences in programmer acceptance of TDD between beginning programmers who were reluctant to adopt TDD and more mature programmers who were more willing to adopt TDD. Attention is given to confounding factors, and future studies aimed at resolving these factors are identified. Finally proposals are made to improve early programmer acceptance of TDD.

1 Introduction

Test-driven development (TDD) [3] is a novel software development practice that has gained recent attention with the popularity of the Extreme Programming [2] software development methodology. Computer science and software engineering educators as well as professional software trainers are beginning to incorporate TDD into their courses. However little is known about the appropriate time and methods for introducing TDD into the curriculum. This research reports on differences in student acceptance of TDD based on programmer maturity.

2 Related Work

A handful of studies have investigated the use of TDD in academia. Some early research reports mixed results [7]regarding quality and productivity improvements from TDD particularly on small software projects. More recent research [6] conducted with advanced undergraduate students suggests that a test-first approach increases the number of tests written and improves productivity, increasing the likelihood of higher quality software with similar or lower effort.

Barriocanal [1] documented an experiment in which students were asked to develop automated unit tests in programming assignments. Christensen [4] proposes that software testing should be incorporated into all programming assignments in a course, but reports only on experiences in an upper-level course. Patterson [12] presents mechanisms incorporated into the BlueJ [10] environment to support automated unit testing in introductory programming courses.

Edwards [5] has suggested an approach to motivate students to apply TDD that incorporates testing into project grades, and he provides an example of an automated grading system that provides useful feedback. The authors have proposed a pedagogic approach called "Test-Driven Learning" (TDL) [9] that incorporates automated tests into programming courses. A minimal TDL approach was employed in the experiments reported here.

3 Experiments

Six experiments were conducted to compare the effects of a test-first (TDD) approach with a test-last approach. All programmers in the experiments were instructed in both approaches and tools for writing automated unit tests. The experiments were part of a larger series of studies investigating the effects of TDD on internal software quality [8].

Five experiments were conducted in academic settings at the University of Kansas and one experiment was conducted in a professional training course in a Fortune 500 company. Separate experiments were conducted in courses ranging from beginning programming (CS1) through graduate software engineering. The first experiment was conducted in an undergraduate software engineering course in Summer 2005. In Fall 2005, experiments were conducted in Programming 1 (CS1), Programming 2 (CS2), and the graduate software engineering course. The CS2 experiment was then repeated in Spring 2006.

Due to course and industry constraints, the experiment design was not consistent across all experiments although

Name	Null Hypothesis	Alternative Hypothesis
O1	$Op_{TF} = Op_{TL}$	$Op_{TF} > Op_{TL}$
O2	$Op\|TF_{TF} = Op\|TF_{TL}$	$Op\|TF_{TF} > Op\|TF_{TL}$

Table 1. Formalized Hypotheses

it was consistent within each experiment. For instance in the CS1 and industry experiment, students were randomly assigned to use a test-first or test-last approach, whereas in the CS2 experiment students self-selected between the two approaches. The early (CS1 and CS2) programmers used the C++ programming language with simple assert statements for automated unit tests, while all other programmers used the Java programming language and JUnit. Course enrollments varied from over one hundred in CS1 to about 30 in CS2 and twelve to fifteen in each of the software engineering and industry training courses.

3.1 Hypothesis

A formalization of the experiment hypotheses is presented in Table 1. Hypothesis **O1** examines whether all programmers, whether they have used the test-first approach or not, perceive test-first as a better approach. Hypothesis **O2** more specifically examines whether programmers who have attempted test-first prefer the test-first approach over a test-last approach.

3.2 Programmer Opinion Results

Programmer opinions of the test-first and test-last approaches were measured in each of the experiments. All programmers participating in the experiments were asked to complete surveys at three points: prior to the experiment (pre-experiment), shortly after the experiment (post-experiment), and several months after the experiment (longitudinal). The results were analyzed statistically using the two-sample *t*-test with significance at $p < .05$.

Figures 1 and 2 report programmer opinions of the test-first and test-last approaches from the post-experiment surveys. The results have been grouped by developer maturity. CS1 and CS2 programmers are in the "Beginning" group, and industry programmers and student programmers from the software engineering courses are in the "Mature" group. The corresponding questions ask programmers to choose:

1. which approach they would choose in the future (Choice)

2. which approach was the best for the project(s) they completed (BestApproach)

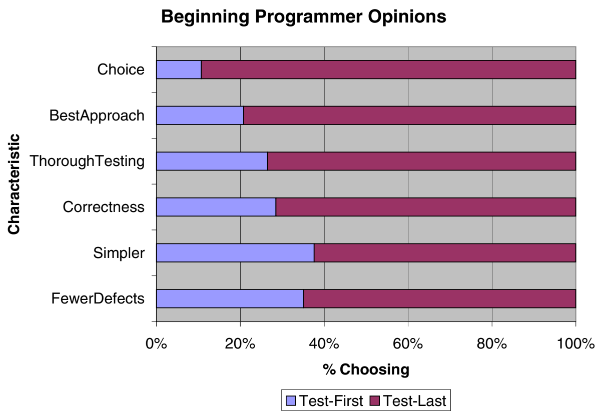

Figure 1. Early Programmer Opinions

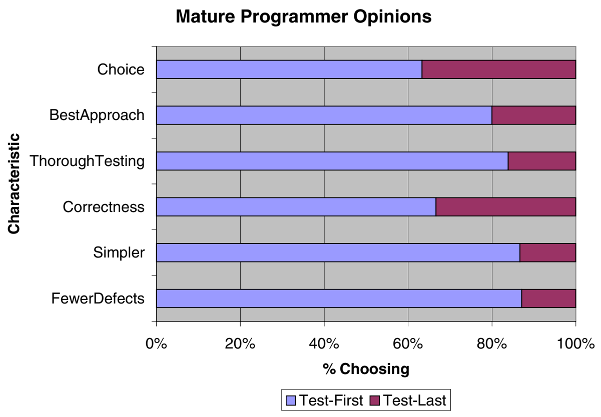

Figure 2. Mature Programmer Opinions

3. which approach would cause them to more thoroughly test a program (ThoroughTesting)

4. which approach produces a correct solution in less time (Correct)

5. which approach produces code that is simpler, more reusable, and more maintainable (Simpler)

6. which approach produces code with fewer defects (FewerDefects)

The charts illustrate that beginning programmers think the test-last approach is better and are more likely to choose it whereas more mature programmers think the test-first approach is better and are more likely to choose it. The longitudinal survey reported similar results with 86% of beginning programmers choosing the test-last approach and 87% of mature programmers choosing the test-first approach.

Interestingly, the percentage of programmers choosing the test-first method is always slightly less than the programmer opinions on other desirable characteristics. In other words, despite recognizing many valuable benefits of

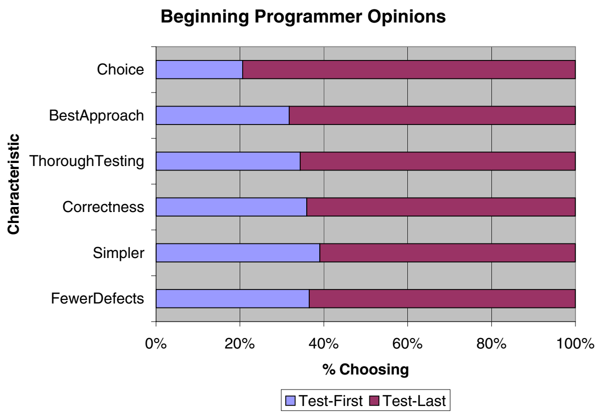

Figure 3. Early Programmers w/TF

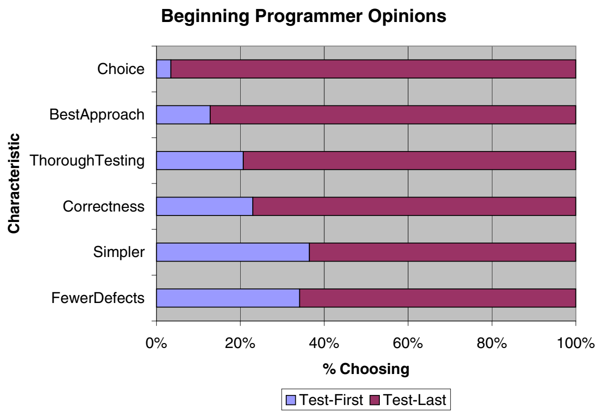

Figure 4. Early Programmers w/Only TL

the test-first approach, some programmers are still unwilling to choose it. A number of comments on the surveys corresponded with this trend. Several programmers noted that even though they thought the test-first approach was better, they perceived it as being more difficult or very different from what they were comfortable with.

The beginning programmer survey data from Figure 1 was divided into two groups: those who used the test-first approach on at least one project and those who only used the test-last approach on all projects. The former group contained a total of 65 programmers and the latter group had 88 programmers. Figure 3 reports the percent of programmers preferring the test-first and test-last approaches on the six characteristics out of programmers who used the test-first approach on at least one project. Figure 4 reports the same information for the programmers who used the test-last approach on all projects.

Likewise the mature programmer survey data from Figure 2 was divided into two groups: those who used the test-first approach on at least one project and those who only used the test-last approach on all projects. The former group contained a total of 16 programmers and the latter group had 15 programmers. Figure 5 reports the percent of programmers preferring the test-first and test-last approaches on the six characteristics out of programmers who used the test-first approach on at least one project. Figure 6 reports the same information for the programmers who used the test-last approach on all projects.

These charts demonstrate that mature programmers who try the test-first approach almost unanimously like and choose the test-first approach. Beginning programmers clearly have a preference for the test-last approach. However, the charts illustrate that trying the test-first approach significantly increases the likelihood that programmers will see benefits with and may choose the test-first approach.

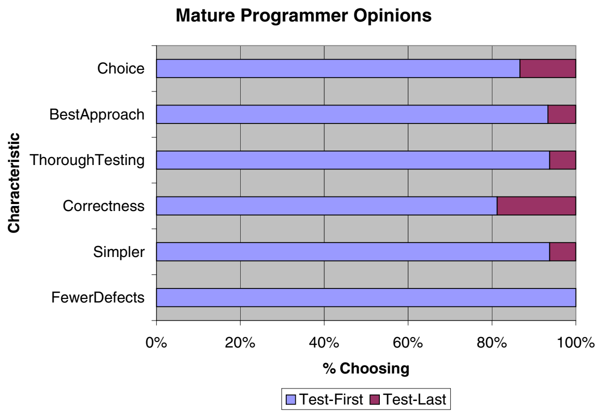

Figure 5. Mature Programmers w/TF

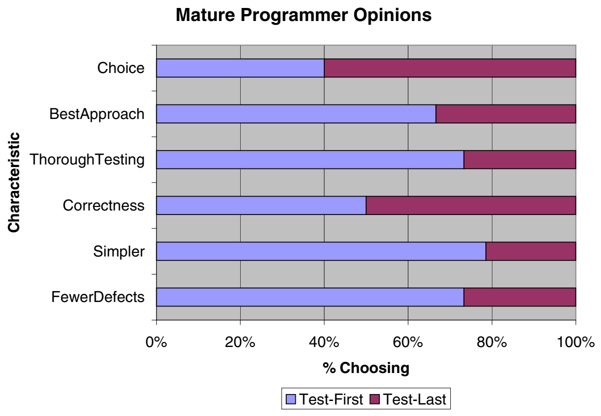

Figure 6. Mature Programmers w/Only TL

Hypothesis O1. Hypothesis **O1** examines whether programmers prefer the test-first or test-last approach. In the pre-experiment survey, beginning programmers had a statistically significant higher opinion of the test-last approach over the test-first approach. Additionally 76% indicated that they would choose the test-last approach. Mature programmers had a slightly (not statistically significant) higher opinion of the test-first approach and 62% indicated that they would use the test-first approach if given the chance. As a result, we must keep the **O1** null hypothesis and assume that programmers in general do not prefer the test-first approach.

Hypothesis O2. Hypothesis **O2** examines programmer opinions after trying the test-first approach. The differences in choice as reported in Figures 3, 4, 5, and 6 are statistically significant for both the beginning and mature developers. Therefore we can claim that developers (both beginning and mature) who try the test-first approach are significantly more likely to choose the test-first approach over the test-last approach. Despite this significant difference, a majority of beginning developers still would choose the test-last approach, while a majority of mature developers would choose the test-first approach. These results allow us to reject the **O2** null hypotheses for mature developers and claim that mature programmers prefer the test-first approach. Although the improvement is significant for beginning developers, we cannot say that they prefer the test-first approach.

3.3 Confounding Factors and Future Work

Three confounding factors were identified in this research. First the early programmers used C++ and assert statements for automated unit tests, whereas the more mature developers used Java and JUnit. A future study is planned with Java and JUnit in early programming courses to determine if the language and testing framework make a difference in programmer acceptance of TDD.

Second the early programmers worked individually whereas the more mature developers worked in teams (SE courses) or in pairs (industry training). Evidence [11] suggests that early programmers have better experiences when pairing. A study could easily examine the effects of adding pair-programming to TDD on TDD acceptance.

Finally the early programmers worked on relatively small (two to three week) projects whereas most of the mature programmers worked on semester-long projects. Early programming courses traditionally use small projects so we propose to apply the test-driven learning approach throughout an early programming course to examine the effects of extended TDD exposure on programmer acceptance.

4 Conclusions

This research has reported on significant differences in programmer willingness to adopt TDD based on TDD experience and developer maturity. First this research has demonstrated that developers are more likely to choose TDD after having tried it. Second this research has revealed that mature developers are much more willing to accept TDD than early programmers. Confounding factors such as programming language, independence, project size and TDD exposure time were identified. Future studies were proposed to address such factors.

References

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

[2] K. Beck. *Extreme Programming Explained.* Addison-Wesley Longman, Inc., 2000.

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

[4] H. B. Christensen. Systematic testing should not be a topic in the computer science curriculum! In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, pages 7–10. ACM Press, 2003.

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

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

[7] D. Janzen and H. Saiedian. Test-driven development: concepts, taxonomy and future directions. *IEEE Computer*, 38(9):43–50, Sept 2005.

[8] D. Janzen and H. Saiedian. On the influence of test-driven development on software design. In *Nineteenth Conference on Software Engineering Education & Training*, pages 141–148. IEEE-CS, 2006.

[9] D. Janzen and H. Saiedian. Test-driven learning: Intrinsic integration of testing into the CS/SE curriculum. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pages 254–258. ACM Press, 2006.

[10] M. Kolling and J. Rosenberg. Guidelines for teaching object orientation with java. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, pages 33–36. ACM Press, 2001.

[11] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald. Pair programming improves student retention, confidence, and program quality. *Commun. ACM*, 49(8):90–95, 2006.

[12] A. Patterson, M. Kolling, and J. Rosenberg. Introducing unit testing with BlueJ. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, pages 11–15. ACM Press, 2003.