Test-Driven Learning in Early Programming Courses

David S. Janzen
California Polytechnic State University
Computer Science
San Luis Obispo, California USA
djanzen@calpoly.edu

Hossein Saiedian
University of Kansas
Electrical Engineering and Computer Science
Lawrence, Kansas USA
saiedian@ku.edu

ABSTRACT

Coercing new programmers to adopt disciplined development practices such as thorough unit testing is a challenging endeavor. Test-driven development (TDD) has been proposed as a solution to improve both software design and testing. Test-driven learning (TDL) has been proposed as a pedagogical approach for teaching TDD without imposing significant additional instruction time.

This research evaluates the effects of students using a test-first (TDD) versus test-last approach in early programming courses, and considers the use of TDL on a limited basis in CS1 and CS2. Software testing, programmer productivity, programmer performance, and programmer opinions are compared between test-first and test-last programming groups. Results from this research indicate that a test-first approach can increase student testing and programmer performance, but that early programmers are very reluctant to adopt a test-first approach, even after having positive experiences using TDD. Further, this research demonstrates that TDL can be applied in CS1/2, but suggests that a more pervasive implementation of TDL may be necessary to motivate and establish disciplined testing practice among early programmers.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Evolutionary prototyping, object-oriented design methods*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Design, Verification

Keywords

Test-driven learning, test-driven development, pedagogy, CS1

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'08, March 12–15, 2008, Portland, Oregon, USA.
Copyright 2008 ACM 978-1-59593-947-0/08/0003 ...$5.00.

1. INTRODUCTION AND RELATED WORK

Test-driven development (TDD) [3] is a disciplined development approach that involves writing automated unit tests before writing the corresponding functional software units in short, rapid iterations. TDD gained popularity as a core practice in eXtreme Programming (XP) [2], but is emerging as a stand-alone practice that can be integrated into a variety of development processes.

Many industry practitioners have embraced the use of TDD in non-trivial situations to a level "beyond the visionary phase and into the early mainstream." [10] Interest among computing educators also appears to be growing. No less than twenty industry and academic (ten each) empirical studies involving TDD are summarized in [6, 10]. The empirical TDD studies in academia have predominately focused on more advanced students in software engineering, capstone, and graduate courses.

Edwards [5, 4], however, proposed the use of TDD throughout the undergraduate computing curriculum. His work focused on the use of automated grading systems as a mechanism to encourage and assess TDD use while providing rapid student feedback. Similarly, TDD support is provided in integrated development environments such as BlueJ [11] and DrJava [1] that target early programmers. Wellington et al. developed an Eclipse plug-in [13] to simplify the writing of test cases for early programmers. Despite the growing interest and tool support for TDD, relatively little has been written regarding how to teach TDD, and whether a test-first or test-last development approach is better in early programming courses.

Test-driven learning (TDL) [8] was proposed at SIGCSE'06 as a mechanism for teaching and motivating the use of automated testing as both a design and a verification activity. TDL proposes to use automated unit tests in lecture, lab, and exercise examples. TDL may be applied at any level of instruction from beginning programming through professional training courses. TDL may be used to teach either a test-first or test-last development approach.

We report here on experiences applying TDL on a limited basis in both a CS1 and CS2 course, and we compare student use of test-first and test-last approaches. The experiment design and context are presented in section 2. Results are presented and analyzed in sections 3 and 4, and conclusions are discussed in section 5.

2. EXPERIMENT DESIGN AND CONTEXT

This experiment was designed to examine the ability of beginning programmers to adopt the test-first and test-last

development approaches, and to determine if the approach used affects the quality of software produced at this level. Test-first refers to writing automated unit tests immediately *before* new functional units are written, and test-last means the tests are written immediately *after* new functional units are written. This experiment was conducted with 104 students in a CS1 course, and 36 students in a CS2 course at the University of Kansas during the Fall 2005 semester. None of the 140 students reported previous experiences with writing automated tests in either a test-first or test-last approach.

2.1 CS1 Experiment Design

Automated testing and test-first/test-last concepts were presented in week six of a sixteen week CS1 course. The first author presented a guest lecture after students had completed three C++ projects and covered topics such as basic syntax, iteration control structures, elementary functions, and simple data structures such as arrays. The lecture introduced automated unit testing of functions using C/C++ *assert* statements. Both a test-first and a test-last application of automated unit testing was presented. Students were asked to complete a pre-experiment survey and sign an informed consent agreement at the end of the guest lecture.

The lecture was followed by two labs that were taught by graduate teaching assistants. The first lab introduced automated unit testing in the context of writing simple functions. The second lab reinforced automated unit testing in the context of writing recursive functions, using reference parameters, and function overloading. The labs applied the TDL approach of introducing concepts and constructs through the use of automated unit tests. The labs introduced the difference between test-first and test-last programming, and gave students hands-on experience with both approaches.

After completing the two labs, students were asked to complete two programming projects. The projects were completed in C++ using a non-integrated development environment. The first project (Project 4) required students to create a data structure for representing a three-dimensional point, then create functions that operate on such points. Students had not been introduced to classes so they generally used an array-based data structure and global functions in their solutions.

Students with student IDs ending in an even number were asked to complete the first project with a test-first approach, and students with student IDs ending in an odd number were asked to complete the first project with a test-last approach. The test-first and test-last groups had forty and sixty-six students respectively.

The second project (Project 5) required students to create class-based data structures for representing points and polygons. A textual user interface was to allow users to specify a number of points in a polygon and the program was to calculate the perimeter and area of that polygon. Test-first/test-last assignments were switched on the second project so students with student IDs ending in an odd number were asked to complete the second project with a test-first approach, and students with student IDs ending in an even number were asked to complete the second project with a test-last approach.

At the beginning of the second project, students were provided a solution to the first project that included a full set of automated unit tests. The post-experiment survey was administered following completion of the second project.

2.2 CS2 Experiment Design

CS2 students were given a very brief introduction to test-first and test-last programming on the first day of the course, then asked to complete the pre-experiment survey and informed consent form. Students were introduced to automated unit testing using *assert* statements in a lab in week three of the sixteen week semester. The lab presented examples and required hands-on exercises with automated tests using classes, along with simple and recursive functions. The lab presented both test-first and test-last approaches.

Students were then required to complete two programming projects using either a test-first or test-last approach. At the request of the course professor, students were allowed to self-select which approach they used, but were encouraged to choose test-first if their student ID started with an even number, and test-last if their student ID started with an odd number. Six students elected to use the test-first approach, while thirty students elected to use the test-last approach. Although there was no statistically significant difference in the previous gpa or overall preparedness of the two groups, the student self-selection does diminish the validity of the CS2 experiment. The low test-first numbers also reveal early programmer reluctance to adopt the test-first approach.

Each programming project was to be completed in two and three weeks respectively. The first project required students to build an application that stored and manipulated a list of automobile drivers with traffic citations. The application had a textual user interface that allowed the user to insert, delete, find, and print driver and citation information. The public interface for the main list data structure class was prescribed in the project description. Students were expected to design at least two additional classes.

The second project extended and modified the first project. The prescribed class was to be modified internally to use a pointer-based linked list instead of an array-based list. The application was to allow multiple lists, and the class interface was modified slightly. Exceptions and some recursive functions were also added to the requirements. At the beginning of the second project, students were provided with a solution to the first project that included a full set of automated unit tests. A post-experiment survey was administered after the completion of the second project.

3. CS1 EXPERIMENT RESULTS

This section reports and discusses the testing, productivity, and subjective/evaluative results of the experiment in the CS1 course. Whenever statistical significance is discussed, a two-sample t-test was used with significance achieved at $p<.05$.

3.1 Test Results

CS1 students wrote automated unit tests as assert statements in a separate function as described in [8]. The number of assert statements written were counted and ratios were calculated for asserts per line-of-code and asserts per module (module=class if classes were used, or entire program if no classes were used). The assert counts were deemed to be a practical estimation of testing effort.

Table 1 reports the testing results for the CS1 projects. The test-first students wrote 52% more asserts in the first project. In the second project, the test-last programmers wrote 39% more asserts. Recall that the students were asked to switch test-first/test-last approaches between the

Metric	#Asserts	#Asserts/ LOC	#Asserts/ Module
	Project 4		
p-value	0.1109	0.2555	0.0870
Significant?	No	No	No
Higher Method	TF	TF	TF
TF Mean	5.85	0.03	5.85
Std Dev	6.68	0.03	6.68
TL Mean	3.85	0.02	3.72
Std Dev	5.28	0.03	5.06
%difference	52%	35%	57%
	Project 5		
p-value	0.1094	0.2489	0.1271
Significant?	No	No	No
Higher Method	TL	TL	TL
TF Mean	1.89	0.01	0.63
Std Dev	2.94	0.02	0.98
TL Mean	3.10	0.02	1.01
Std Dev	4.18	0.02	1.38
%difference	-39%	-28%	-38%

Table 1: CS1 Test Metrics

two projects. This data indicates that the same programmers wrote more tests in both projects regardless of the approach they used. Because there were no statistically significant differences in the academic background and programming experience of the two groups, one must question whether the first approach used somehow influenced the number of asserts written. Did this group write more asserts because they started out using the test-first approach? Did this approach somehow form a habit or appreciation for writing tests that persisted even when using a test-last approach? This explanation seems plausible given that studies with more mature developers [14, 7] indicated that test-first programmers consistently write more tests than test-last programmers.

3.2 Productivity Results

This section discusses the volume of code produced, and the amount of time students reported they spent on the projects. The test-first programmers on the first project reported spending 10% more time producing solutions that were 7% more lines of code than the test-last programmers. In the second project, the test-first programmers reported spending 11% more time producing solutions that were 11% smaller than the test-last solutions. The test lines of code are included in the total lines of code comparisons here.

Not surprisingly, it appears that the solution size corresponds to the number of tests written. Although none of the productivity differences were statistically significant, the development time data seems to indicate that beginning test-first programmers take slightly more time completing their solutions than the test-last programmers. However, it is interesting to note that the programmers who used the test-first approach in the first project, actually wrote more tests (and more code) in less time in the second project.

3.3 Subjective and Evaluative Results

This section presents results on student project grades. Table 2 reports results from an analysis of the grades assigned to the two CS1 projects. Mean and standard deviations are given, and p-values<.05 would indicate statistical significance. Graduate teaching assistants assigned the scores based on a rubric provided by the professor. Component scores such as for correctness, style, and error checking were not tracked. The data indicates virtually no differences between the test-first and test-last groups.

Project	p-value	TF Mean	TF SDev	TL Mean	TL SDev	%diff
4	0.8516	95.26	5.72	95.05	5.30	0%
5	0.6645	87.77	13.91	89.04	14.72	-1%

Table 2: CS1 Project Evaluations

	Quality	Changes	Reuse
p-value	0.0459	0.0242	0.0233
Significant?	Yes	Yes	Yes
Higher Method	TF	TF	TF
TF Mean	3.98	3.90	3.69
Std Dev	1.25	1.24	1.34
TL Mean	3.25	3.06	2.88
Std Dev	1.74	1.76	1.64
%difference	22%	27%	28%

Table 3: CS1 Programmer Opinions on Project 5

3.4 Programmer Perceptions

This section describes the results from the pre and post experiment surveys. Table 3 reports the statistically significant differences on project 5. The test-first programmers indicated that they were more confident that the code they wrote was correct (Quality), they were more confident that they could make changes to their code without breaking things (Changes), and they were more confident that they could reuse their code in a future project (Reuse). The differences on project 4 were not statistically significant.

Figure 1 illustrates student responses in the post experiment survey from the following questions:

- which approach they would choose in the future (Choice)
- which approach was the best for the project(s) they completed (BestApproach)
- which approach would cause them to more thoroughly test a program (ThoroughTesting)
- which approach produces a correct solution in less time (Correct)
- which approach produces code that is simpler, more reusable, and more maintainable (Simpler)
- which approach produces code with fewer defects (FewerDefects)

Despite higher opinions in other categories and what may have been positive experiences with the test-first approach, only 10% of the CS1 programmers indicated that they would choose to use the test-first approach.

3.5 Longitudinal Results

Twenty-eight students completed a longitudinal survey in late Spring 2006, about four months after participating in

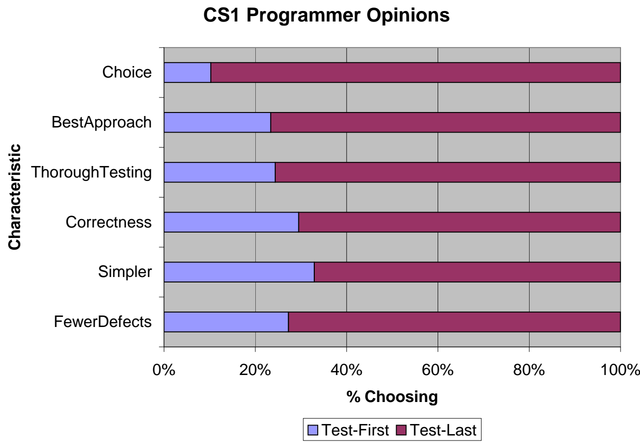

Figure 1: CS1 Programmer Opinions

Metric	#Asserts	#Asserts/ LOC	#Asserts/ Module
p-value	0.02456	0.0059	0.0263
Significant?	Yes	Yes	Yes
Higher Method	TF	TF	TF
TF Mean	34.00	0.06	4.72
Std Dev	43.14	0.06	6.15
TL Mean	11.61	0.02	1.58
Std Dev	17.80	0.03	2.48
%difference	193%	168%	200%

Table 4: CS2 Test Metrics

the original experiment. Ten of the twenty-eight (36%) reported using the test-first approach on a project where they had a choice. Twenty-one reported voluntarily using the test-last approach on a subsequent project. Only two students (7%) indicated that they would choose to use the test-first approach on future projects given the option. The data in this section has demonstrated that beginning programmers are clearly uneasy about adopting the test-first approach.

4. CS2 EXPERIMENT RESULTS

This section reports and discusses the testing, productivity, and subjective/evaluative results of the experiment in the CS2 course.

4.1 Test Results

Similar to the CS1 experiment, students wrote automated unit tests as assert statements in a separate function. Table 4 reports the aggregate testing results for the CS2 projects. Number of asserts per method and number of asserts per class were also calculated because the CS2 projects all involved solutions with classes. Individual results from each project are not reported to save space. These additional metrics and project comparisons resulted in similar statistically significant differences.

4.2 Productivity Results

The test-first programmers reported spending 13% less time producing solutions that were 12% larger in lines of code than the test-last programmers on all of the CS2 projects.

Metric	p-value	TF Mean	TF SDev	TL Mean	TL SDev	%diff
Project 1						
Score	0.0938	88.83	8.95	79.47	21.01	12%
Correct	0.4730	40.50	8.38	37.23	15.30	9%
Style	0.0259	28.33	2.34	24.67	6.63	15%
I/O	0.0246	10.00	0.00	8.63	3.16	16%
Robust	0.0136	10.00	0.00	9.00	2.08	11%
Project 2						
Score	0.0435	90.17	15.45	72.83	21.60	24%
Correct	0.0110	43.50	9.46	28.80	16.60	51%
Style	0.9037	28.33	2.34	28.47	2.61	0%
I/O	0.4592	8.33	4.08	6.90	4.20	21%
Robust	0.0434	10.00	0.00	8.67	3.46	15%

Table 5: CS2 Project Evaluations

Recall that the lines of code includes test lines of code and the previous section revealed that the test-first programmers wrote significantly more tests than the test-last programmers. The differences are not statistically significant.

4.3 Subjective and Evaluative Results

Table 5 reports results from an analysis of the grades assigned to the CS2 projects. Again, mean and standard deviations are given, and p-values$<.05$ indicate statistical significance. Graduate teaching assistants assigned the scores based on a mutually agreed upon rubric. The total score (Score) is presented along with component scores that account for proper working of the software (Correct), good internal design and programming style (Style), adherence to requirements in input/output (I/O), and robust detection and handling of error conditions (Robust).

The data indicates that the test-first projects were deemed superior to the test-last projects in several categories. Several of these differences are significant at $p<.05$.

4.4 Programmer Perceptions

Figure 2 illustrates programmer opinions from the post experiment survey. This chart coincides with the CS1 results and indicates that beginning programmers prefer the test-last approach. There were no statistically significant differences in programmer confidence between the students who used the test-first and test-last approaches.

4.5 Longitudinal Results

A longitudinal survey was administered in late Spring 2006 for the Fall 2005 CS2 experiment. Twelve students completed the survey. Fifty percent reported using a test-first approach on at least one subsequent project, and seventy percent reported using a test-last approach when given the choice. Three of the twelve (25%) indicated that they would choose the test-first approach if given the option on future projects.

5. CONCLUSIONS AND FUTURE WORK

This research compared the effects and acceptance of test-first and test-last approaches in early programming courses. We are unable to make broad conclusions due to the lack of randomized groups and small number of test-first programmers in the CS2 experiment. We have several confounding factors in the CS1 experiment including solutions with auto-

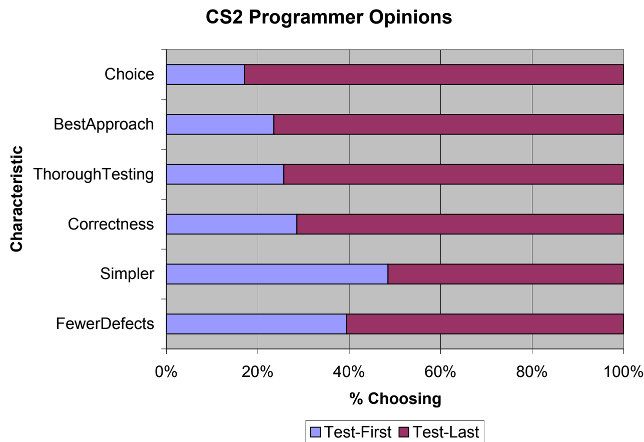

Figure 2: CS2 Programmer Opinions

mated tests following the first project, and the introduction of objects in the second project. All these decisions were deemed necessary for preserving the pedagogical integrity of the courses. However, we can compare our results with studies conducted with more mature programmers in order to determine some trends and future directions.

Similar to findings with more mature programmers, test-first programmers in CS2 wrote significantly more automated unit tests. In the CS1 experiment, programmers who first used a test-first approach wrote more tests on average, even when they later switched to a test-last approach. The authors observed a similar phenomenon in an experiment with professional programmers. This result causes one to question whether using test-first *first* causes some sort of residual effect.

Test-first programmers in CS2 scored higher on most component and overall project grades than their test-last counterparts. Plus they accomplished this with less effort (time). We encourage further studies with fully randomized samples to determine if this can be expected in general, or was the result of other factors.

Programmer opinions gathered at the beginning and end of this experiment indicate a strong reluctance on the part of early programmers to adopt a test-first approach. This result was noted in [9], and contrasts with experiments [7] with more mature developers using the Java Programming Language. It is possible that the use of the C++ language with a rudimentary assert mechanism for automated tests is partly to blame.

In addition, it seems likely that a pervasive TDL approach may be necessary to improve adoption motivation. It is very likely that students need to see examples using automated unit testing modeled throughout an early programming course. Perhaps ideally, such an approach might be combined with automated grading systems [5, 12] to enforce high test coverage and attribute grade value to testing activities.

The authors intend to complete additional studies applying TDL with Java and JUnit throughout an early programming course. We encourage replicated studies in similar and diverse environments. Resources from this experiment, including some lecture slides and lab materials are available at http://www.simexusa.com/tdl/.

6. REFERENCES

[1] E. Allen, R. Cartwright, and B. Stoler. Drjava: a lightweight pedagogic environment for java. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 137–141, New York, NY, USA, 2002. ACM Press.

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

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

[4] 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.

[5] S. H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 26–30, New York, NY, USA, 2004. ACM Press.

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

[7] 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.

[8] 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.

[9] D. Janzen and H. Saiedian. A leveled examination of test-driven development acceptance. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 719–722, Washington, DC, USA, 2007. IEEE Computer Society.

[10] R. Jeffries and G. Melnik. Tdd – the art of fearless programming. *IEEE Software*, 24(3):24–30, 2007.

[11] 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.

[12] J. Spacco, D. Hovemeyer, W. Pugh, J. Hollingsworth, N. Padua-Perez, and F. Emad. Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses. In *ITiCSE '06: Proceedings of the 11th annual conference on Innovation and technology in computer science education.* ACM Press, 2006.

[13] C. Wellington, T. Briggs, and C. D. Girard. Experiences using automated tests and test driven development in computer science i. In *Agile 2007*, pages 106–112, Washington, DC, USA, 2007. IEEE Computer Society.

[14] L. Williams, M. Maximillien, and M. Vouk. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337–342, 2003.