On the Influence of Test-Driven Development on Software Design

David S. Janzen **Hossein Saiedian**
Electrical Engineering and Computer Science
University of Kansas, Lawrence, KS USA
{djanzen,saiedian}@eecs.ku.edu

Abstract

Test-driven development (TDD) is an agile software development strategy that addresses both design and testing. This paper describes a controlled experiment that examines the effects of TDD on internal software design quality. The experiment was conducted with undergraduate students in a software engineering course. Students in three groups completed semester-long programming projects using either an iterative Test-First (TDD), iterative Test-Last, or linear Test-Last approach. Results from this study indicate that TDD can be an effective software design approach improving both code-centric aspects such as object decomposition, test coverage, and external quality, and developer-centric aspects including productivity and confidence. In addition, iterative development approaches that include automated testing demonstrated benefits over a more traditional linear approach with manual tests. This study demonstrates the viability of teaching TDD with minimal effort in the context of a relatively traditional development approach. Potential dangers with TDD are identified regarding programmer motivation and discipline. Pedagogical implications and instructional techniques which may foster TDD adoption will also be referenced.

1: Introduction

Test-driven development [3] (TDD) has emerged as a novel software development approach that involves writing automated unit tests in an iterative Test-First manner. When applying TDD, a software developer writes one small automated unit test. The developer then writes just enough code to make the test pass. After possible refactoring, the cycle then quickly repeats with the developer writing another test and code to satisfy the test.

As a member of the Extreme Programming (XP) [2] best practices, TDD is most often associated with agile software development processes. Many agile processes reject a comprehensive design step preceding significant programming in favor of a small architectural sketch followed quickly by programming. In such a process, the software design and perhaps architecture are allowed to *emerge* as the software grows. Programmers make decentralized design decisions as they are coding.

TDD is considered an essential strategy in such an emergent design because when writing a test prior to code, the programmer contemplates and decides not only the software's interface (e.g. class/method names, parameters, return types, and exceptions thrown), but also on the software's behavior (e.g. expected results given certain inputs). For instance the following simple test is written using JUnit [6].

```
public void testCreateEmptyBank() {
    Bank b = new Bank();
    assertEquals(b.getNumAccounts(),0);
}
```

Despite being such a simple test, it involves several design decisions including the class name (Bank), the expectation of a default constructor, a method named `getNumAccounts()` that returns an int, and the behavior that a default Bank has no accounts.

This paper describes a formal experiment that examines how the TDD approach involving Test-First programming with minimal up-front design affects internal software design quality. The experiment was conducted during the summer of 2005 with upper-level undergraduate students in a software engineering course. Students in three groups completed semester-long programming projects using either an iterative Test-First (TDD), iterative Test-Last, or linear Test-Last approach. Iterative approaches involve writing automated unit tests either just before (Test-First) or just after (Test-Last) a small portion of code is written. Manual or automated unit tests are written much later with the linear Test-Last approach. This experiment demonstrates the feasibility of using TDD in the context of a more traditional development method (i.e. no XP), and reveals potential quality improvements with minimal instruction cost.

2: Related Work

While some practitioners have applied some form of TDD for several decades [11], academic and industry studies have only more recently emerged [9]. These studies have examined the effects of TDD on external quality and programmer productivity with somewhat mixed results.

Two industry case studies [8, 15] report reductions in defect density with minimal impact on programmer productivity with TDD. Erdogmus [5] on the other hand identified productivity improvements with no significant change in external quality based on a controlled academic experiment.

Edwards [4] conducted studies with beginning programmers and found a significant reduction in defects. This study utilized a web-based program submission system that factored student-written tests and test coverage into an automated grading process that apparently provided significant motivation for early programmers to write tests.

While these results are mixed, there are no alarming results that might make TDD adoption risky. Interestingly, despite TDD being primarily a design mechanism, none of the studies to date examined the internal quality of software developed with TDD. Perhaps this is because TDD also produces tests which are generally associated with external quality, or simply because external quality is easier to measure by counting external test pass rates.

Internal quality is somewhat more subjective and prone to much debate. However, many internal metrics do exist that can provide insight on the quality of a software design or at least lack of quality in software. Also, if we agree that an important attribute of high internal quality software is that it is easier to modify, enhance, and reuse, then productivity and reuse can serve as indirect measures of internal quality as well.

3: Experimental Design

The goal of this experiment is to compare iterative Test-First programming with Test-Last programming for the purpose of evaluating internal quality, programmer productivity, and programmer perceptions. The experiment was conducted in the context of an undergraduate software engineering course consisting of junior and senior computer science and computer engineering students with at least two previous programming courses.

Students were asked to design and build an HTML pretty print system. This system was to take an HTML file as input and transform the file into a more human readable format by performing operations such as deleting redundant tags and adding appropriate indentation.

Students were taught a simplified form of the Unified Process including inception, elaboration, construction, and transition stages. The project schedule was divided into two iterations with the first focusing on a text-based user interface and a partial set of features. The second iteration added a graphical user interface and additional features.

Students were taught how to write automated unit tests with the JUnit framework. All students were instructed in how to write software in a Test-First and Test-Last manner. The total time spent on JUnit and Test-First/Test-Last programming was less than one and a half hours. Students were then divided into three groups: two groups were to complete the project with a Test-First approach and the third group was to use a Test-Last approach. Students were allowed to self-select their teams, but Java programming experience was established as a blocking variable to ensure that each team had at least one member with previous Java experience. Test-First/Test-Last team assignments were made after analyzing the pre-experiment questionnaire to ensure the teams were reasonably balanced.

3.1: Hypotheses

Several hypotheses are examined. A formalization of the hypotheses is given in Table 1. Each of these hypotheses is discussed in turn here. Hypothesis **P1** considers whether Test-First programmers are more productive than Test-Last programmers. We will examine development time, effort per feature, and effort per lines of code.

Some sources[1, 3] claim that Test-First programmers consistently write a significant amount of test code. Hypothesis **T1** examines whether Test-First programmers write more tests than Test-Last programmers. **T2** augments **T1** by examining whether the tests written by Test-First programmers actually exercise more production code (test-coverage) than the tests written by Test-Last programmers. The rationale for **T2** is that more tests may only be better if the tests actually exercise more lines or branches in the production code.

Hypothesis **Q1** tests if Test-First code has higher internal quality than Test-Last code. Recognizing that not all code may be covered by automated unit-tests, hypothesis **Q2** considers whether code developed in a Test-First manner and covered by tests has higher internal quality than code also developed in a Test-First manner, but not covered by tests. In an ideal situation, this hypothesis could not be examined because all Test-First code would be covered by unit tests. However, the reality is that students first learning to use TDD will rarely achieve such high test-coverage.

Finally hypothesis **O1** and **O2** address programmer opinions of the Test-First approach. Hypothesis **O1** examines whether programmers 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.

Name	Null Hypothesis	Alternative Hypothesis
P1	$\text{Prod}_{TF} = \text{Prod}_{TL}$	$\text{Prod}_{TF} > \text{Prod}_{TL}$
T1	$\#\text{Tests}_{TF} = \#\text{Tests}_{TL}$	$\#\text{Tests}_{TF} > \#\text{Tests}_{TL}$
T2	$\#\text{TestCov}_{TF} = \#\text{TestCov}_{TL}$	$\#\text{TestCov}_{TF} > \#\text{TestCov}_{TL}$
Q1	$\text{IntQlty}_{TF} = \text{IntQlty}_{TL}$	$\text{IntQlty}_{TF} > \text{IntQlty}_{TL}$
Q2	$\text{IntQlty}\vert\text{Tested}_{TF} = \text{IntQlty}\vert\text{Untested}_{TF}$	$\text{IntQlty}\vert\text{Tested}_{TF} > \text{IntQlty}\vert\text{Untested}_{TF}$
O1	$\text{Op}_{TF} = \text{Op}_{TL}$	$\text{Op}_{TF} > \text{Op}_{TL}$
O2	$\text{Op}\vert\text{TF}_{TF} = \text{Op}\vert\text{TF}_{TL}$	$\text{Op}\vert\text{TF}_{TF} > \text{Op}\vert\text{TF}_{TL}$

Table 1. Formalized Hypotheses

4: Data Analysis

Originally two student teams were instructed to use a Test-First approach and one team was instructed to use a Test-Last approach. Only one team actually used the Test-First approach. This team will be labeled the "Test-First" team. The other Test-First team did write automated unit tests, but the tests were not written by the same developer who wrote the production code, and they were written after implementation of the production code. This team will be labeled the "Test-Last" team. Despite being instructed to write automated unit tests, the remaining Test-Last team reported that they "ran out of time" and performed only manual testing. This team will be labeled the "No-Tests" team. While this reclassification of the groups calls into question the level of control in this controlled experiment, the existence of a Test-First and Test-Last group satisfies the experiment goals, and the creation of a "No-Tests" group adds an interesting alternative for comparison.

4.1: Productivity

The Test-First team implemented about twice as many features (12) as the No-Tests and Test-Last teams (5 and 6), with similar numbers of defects. In addition, the Test-First team was the only one to complete the graphical user interface. Despite implementing more features, the Test-First team did not invest the most time of all the teams. Table 2 reports the amount of time each team spent on the project. Total effort includes time spent on all project activities including general meetings and research. Dev(elopment) Effort includes only time spent directly on the project including analysis, design, code, test, fix, and review.

The Test-First team spent less effort per line-of-code and they spent 88% less effort per feature than the No-Tests team, and 57% less effort per feature than the Test-Last team. Individual productivity is known to vary widely among programmers so it is certainly possible that the Test-First team was blessed with one or more highly productive programmers. However, analysis of the pre-experiment questionnaire indicates that there was no statistically significant difference in the academic or practical background of the teams. This data indicates that Test-First programmers may be more productive than Test-Last programmers, however a larger sample size is necessary before rejecting the **P1** null hypothesis.

Team	Total Effort	Dev Effort	Dev Effort/LOC	Dev Effort/Feature
Test-First	6504	2239	2.13	186.58
No-Tests	11385	7340	7.38	1468.00
Test-Last	4450	2575	9.94	429.17

Table 2. Effort in minutes

Team	Classes	LOC	Test LOC	LOC/method	LOC/feature
Test-First	13	1053	168	12.10	87.75
Test-First(no GUI)	11	670	168	11.75	55.83
No-Tests	7	995	0	27.64	199.00
Test-Last	4	259	38	7.40	43.17

Table 3. Code Size Metrics

4.2: Code Size and Test Density

Table 3 reports the size of the code implemented in terms of number of classes and lines of code. For comparison, we also give the code size of the Test-First application with only the text user interface. While the Test-First team implemented additional features besides the graphical user interface, the GUI was a significant feature and removing it allows a more consistent comparison with the two teams that only implemented a text user interface.

As might be expected, the Test-First team implemented more code than the other two teams. We note that both the Test-First and Test-Last teams have a reasonable average method size and lines-of-code per feature, but the No-Tests team apparently wrote long methods and implemented an excessive amount of code for the provided functionality.

Table 4 reports test size and test coverage metrics as calculated with the STREW [14] Eclipse plug-in. The Test-First team wrote almost twice as many assertions per source-line-of-code as the Test-Last team. While the tests did not cover a significantly higher number of lines, they did cover 86% more branches than those written by the Test-Last team. This data indicates a statistically insignificant trend against **T1** and **T2** that merits further investigation.

4.3: Internal Quality

Over twenty-five structural and object-oriented metrics were calculated for all software to gauge internal quality. The metrics were gathered using freely available tools (see `http://metrics.sourceforge.net` and `http://cccc.sourceforge.net`). While most metrics had comparable and acceptable values for all three projects, some warnings were

Team	Assertions/SLOC	Test Coverage (lines)	Test Coverage (branches)
Test-First	0.077	19.00%	39.00%
Test-First(less GUI)	0.086	31.00%	43.00%
No-Tests	0.000	0.00%	0.00%
Test-Last	0.045	29.00%	23.00%

Table 4. Test Density and Coverage Metrics

Team	Nested Block Depth		Cyclomatic Complexity		Parameters		CBO		IF
	avg	max	avg	max	avg	max	avg	max	avg
Test-First	**2.02**	**6**	**2.33**	**13**	**0.62**	**6**	4.58	**20**	2.56
Test-First(no GUI)	**1.85**	**6**	**2.59**	**13**	0.89	6	3.27	5	1.47
No-Tests	**3.00**	**6**	**6.53**	**27**	1.08	5	2.57	6	0.00
Test-Last	1.20	3	1.46	4	0.57	3	1.25	2	0.00

Table 5. Internal Quality Metrics with Warnings

noted regarding complexity and coupling. Table 5 identifies concerns in the Test-First and No-Tests code with Nested Block Depth, Cyclomatic Complexity, Number of Parameters, Coupling Between Objects, and Information Flow.

A manual inspection of the projects reveals that the No-Tests and Test-Last systems, while organized into classes, are quite procedural in nature. The No-Tests code contains classes with verb names such as "AlignTags" and "DeleteTags" as well as many long, complex loops. The Test-Last code defines only three classes besides a holder class for `main()`, and `main()` contains the primary control logic of the system. The Test-Last code achieves more functionality with less code by relying heavily on the `java.util.regex.*` library from Java 1.5.

The Test-First code on the other hand is decomposed in a very object-oriented way with responsibilities being distributed between thirteen classes. There are concerns that coupling is too high in the Test-First code, particularly in the one class that has an Coupling Between Objects (CBO) of 20. It turns out that the GUI is created in one large class that is tightly coupled with many other parts of the system. GUI's are traditionally hard to test, and as noted above, the GUI code was not covered by any automated unit tests. Various approaches such as Dependency Inversion [13] and Command [7] objects might be used to reduce the coupling and allow automated testing of GUI code. Without knowledge of these patterns, it seems that the inability to design tests may have contributed to the high coupling in these modules.

An additional micro-evaluation was performed on the Test-First code. Code that was covered by automated unit tests was separated from code not covered by any tests. Table 6 reports differences with Weighted Methods per Class, Coupling Between Objects, Nested Block Depth, Computational Complexity, and Number of Parameters. All values for the 28% of methods that were tested directly are within normal acceptable levels, but values for NBD, Complexity, and Parameters are flagged with warnings in the untested code. The tested methods had a complexity average 43% lower than their untested counterparts. A two-sample t-test comparing the complexity means produces a p-value of .08. In addition, tested classes had 104% lower coupling measures than untested classes. Although this is insufficient to reject the **Q2** null hypothesis, it draws attention to complexity and coupling as effects that should be investigated further. In addition, the relatively low test coverage calls us to question whether the untested code would have a higher internal quality if it were written with tests, or if the lack of tests and corresponding internal quality issues are the result of other factors such as basic programmer laziness. In either case, it might be said that lack of test coverage could be an indicator of potential internal quality issues.

The balance of concerns with coupling/complexity along with manual observations on software design keep us from rejecting the **Q1** null hypothesis. While the micro-evaluation

	WMC		CBO		NBD		Complexity		Parameters	
Code	**avg**	**max**	**avg**	**max**	**avg**	**max**	**avg**	**max**	**avg**	**max**
Tested	7.80	21	2.2	3	1.50	3	1.77	5	1.00	3
Untested	13.55	53	**4.5**	**20**	**2.20**	**6**	**2.53**	**13**	**0.48**	**6**

Table 6. Metrics on Tested and Untested code of Test-First Project

	Test-First			Test-Last		
Team	**Pre**	**Post**	**% Change**	**Pre**	Post	**% Change**
Test-First	3.67	4	9%	3.33	2.33	-30%
No-Tests	1.5	2	33%	3.67	3.33	-9%
Test-Last	2.33	3.25	39%	4	3.25	-19%

Table 7. Programmer Perceptions of Test-First and Test-Last (0 to 4 scale)

of the Test-First software did shed some light on the effects of testing on internal quality, additional study is needed before making any widespread claims.

4.4: Programmer Perceptions

Pre and post-experiment surveys were administered to all programmers. Comparisons between the two surveys are reported in Table 7 and reveal that all three teams perceived the Test-First approach more positively after the experiment and inversely perceived the Test-Last approach more negatively. Additionally, 89% of programmers thought Test-First produced simpler designs, 70% thought Test-First produced code with fewer defects, and 75% thought Test-First was the best approach for this project.

In the post-experiment survey, all programmers who tried Test-First indicated they would prefer to use Test-First over Test-Last in future projects. All programmers from the No-Tests team indicated they would prefer to use Test-Last again on future projects. Comments on their surveys indicated that the No-Tests programmers are more comfortable with an approach that they already know. Programmers from the Test-Last team were split with half preferring to use Test-First on future projects and half choosing Test-Last.

Programmers were also asked in the post-experiment survey to evaluate their confidence in the software they developed. Although most responses were similar, the Test-First team did report higher confidence in the ability to make future changes to their software. A two-sample t-test comparing this difference with the Test-Last team was not statistically significant (p=.059).

4.5: Threats to Validity

The primary threat to validity was the small sample size. Only ten programmers participated in this study which is too few to draw any broad conclusions. Furthermore, The post-experiment survey revealed that a single programmer on each of the three teams implemented a majority of the core functionality. While all team members participated in development on each project, it is possible that differences in quality and productivity could be attributed to the individual skill levels of only three programmers.

5: Conclusions

This study evaluated the influence of TDD on programmer productivity and internal quality, with additional information regarding effects on test coverage and programmer perceptions. Results indicate that the Test-First approach may have a positive correlation with programmer productivity. While internal quality was not shown to be better with Test-First code, concerns were raised about internal quality issues when the Test-First process breaks down and tests are not written.

The study also demonstrated that programmers perceive TDD more positively after exposure to it, and particularly they are much more likely to adopt TDD after having tried it. The issue of motivating programmers to adopt TDD is raised. While a variety of techniques [12] have been identified for introducing new ideas like TDD into organizations, faculty have the luxury of setting course and grading requirements that can include the use of TDD. Academic efforts such as Test-Driven Learning [10] which incorporate automated tests through all levels of the curriculum may hold some promise for incorporating automated testing into the curriculum. Further studies with larger populations and a broader base of programmers including students and professional practitioners will reveal the validity of the observations in this work and may provide the motivation for broader adoption.

References

[1] David Astels. *Test Driven Development: A Practical Guide*. Prentice hall PTR, 2003.

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

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

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

[6] E. Gamma and K. Beck. http://www.junit.org.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[8] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337–342, 2004.

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

[10] D. Janzen and H. Saiedian. Test-driven learning: intrinsic incorporation of testing into the cs/se curriculum. In *SIGCSE '06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, 2006.

[11] Craig Larman and Victor R. Basili. Iterative and incremental development: a brief history. *IEEE Computer*, 36(6):47–56, June 2003.

[12] Mary Lynn Manns and Linda Rising. *Fearless Change*. Addison-Wesley Professional, 2004.

[13] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, Inc., 2003.

[14] N. Nagappan, L. Williams, M. Vouk, and J. Osborne. Early estimation of software quality using in-process testing metrics: a controlled case study. In *Third Software Quality Workshop, co-located with the International Conference on Software Engineering (ICSE 2005)*, pages 46–52, May 2005.

[15] L. Williams, E.M. Maximilien, and M. Vouk. Test-driven development as a defect-reduction practice. In *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering*, pages 34–45, Nov. 2003.