2009 Sixth International Conference on Information Technology: New Generations

Evaluating Test-Driven Development in an Industry-sponsored Capstone Project

John Huan Vu, Niklas Frojd, Clay Shenkel-Therolf, and David S. Janzen
California Polytechnic State University, San Luis Obispo, California

Abstract

Test-Driven Development (TDD) is an agile development process wherein automated tests are created before production code is designed or constructed in short, rapid iterations. This paper discusses an experiment conducted with undergraduate students in a year-long software engineering capstone course. In this course the students designed, implemented, deployed, and maintained a software system to meet the requirements of an industry sponsor who served as the customer. The course followed an incremental process in which features were added incrementally under the direction of the industry sponsor and the professor. The fourteen students observed in the study were divided into three teams. Among the three teams were two experimental groups. One group consisted of two teams that applied a Test-First (TDD) methodology, while a control group applied a traditional Test-Last methodology. Unlike Test-First, the tests in Test-Last are written after the design and construction of the production code being tested. Results from this experiment differ from many previous studies. In particular, the Test-Last team was actually more productive and wrote more tests than their Test-First counterparts. Anecdotal evidence suggests that factors other than development approach such as individual ambition and team motivation may have more affect than the development approach applied. Although more students indicated a preference for the Test-First approach, concerns regarding learning and applying TDD with unfamiliar technologies are noted.

Keywords: Test-Driven Development (TDD), Test-First methodology, Test-Last methodology, software engineering, capstone project.

1. Introduction

The software engineering industry has a constant desire to improve the overall quality of the software produced. One method to improve software quality is the development of unit tests throughout implementation. Test-driven development (TDD) takes this a step further by using the development of automated unit tests to drive the design of software, focusing the developer on testable interfaces prior to implementation concerns. Although improving the quality of a software can mean different things to different people; this paper aims to help examine two different testing methodologies—Test-First and Test-Last—and their affects on both internal and external quality of the software. This paper outlines the experiment design along with an analysis and formalized conclusions based on experimental results.

2. Related work

This section provides information about previous studies of TDD. This includes related work and experimental design completed at other institutions.

2.1. Previous studies

TDD has been studied in a number of prior experiments. An early study explored the effects of TDD versus a waterfall-like approach on code quality and test coverage [5]. It was suggested that TDD encouraged the implementation of unit tests. A study similar to the one presented in this article found that undergraduate students who used a TDD methodology as opposed to a more traditional development process tended to write more tests and were more productive [1]. Another study compared Test-First methodology to Test-Last in early computer science courses. This study reported that students who followed Test-First wrote more tests than their Test-Last counterparts [3]. Other experiments examined whether or not TDD improves software design quality [4,6]. These studies demonstrated that TDD led to smaller and simpler methods and classes, and higher cohesion.

2.2. Test-First versus Test-Last methodologies

According to *Test Driven*, TDD can be described as to "only ever write code to fix a failing test" [2]. Before any production code is ever written, the programmer must first write a test that will define the new functionality being coded. That is why TDD is referred to in the industry and throughout this paper as Test-First.

The traditional software development process is referred to in the industry and throughout this paper as Test-Last. In this case, the programmer writes test after the production code is written.

978-0-7695-3596-8/09 $25.00 © 2009 IEEE
DOI 10.1109/ITNG.2009.11

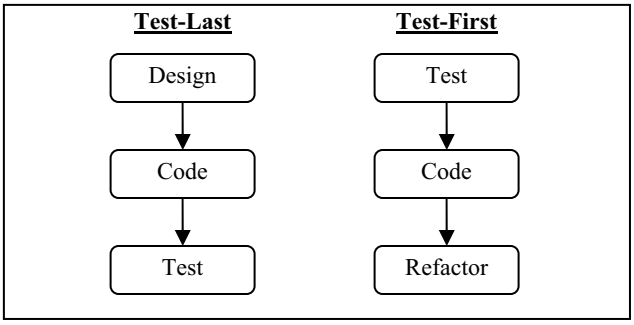

Figure 2.2.1. Comparison of Test-First and Test-Last methodologies [2].

The Test-First and Test-Last methodologies can be summarized in Figure 2.2.1. These figures only describe the detailed design, code, and unit test phases of the software development lifecycle. Both Test-First and Test-Last presume that requirements and high-level architecture phases precede them, and that they are followed by a quality assurance phase.

A key note to mention is that Test-First methodology sequence uses the word "refactor" in Figure 2.2.1. According to Koskela, "The final step of the Test-Driven Development cycle of test-code-refactor is when we take a step back, look at our design, and figure out ways of making it better [2]." Although none of the steps in the Test-Last methodology sequence contain the word "refactor", this does not imply that this activity is omitted. Refactoring occasionally occurs during the test phase of the Test-Last methodology when programmers are addressing known software defects.

The following are the steps of Test-First methodology [1] and are summarized in Table 2.2.1:

1. Pick a feature or a user requirement.
2. Write a test that fulfills a small task or piece of the feature or user requirement (e.g. one method) and have the test fail.
3. Write the production code that implements the task and will pass the test.
4. Run all of the tests.
5. Refactor the production and test code to make them as simple as possible, ensuring all tests pass.
6. Repeat steps 2 to 5 until the feature or user requirement is implemented.

The following are the steps of Test-Last methodology [1] and are summarized in Table 2.2.1:

1. Pick a feature or a user requirement.
2. Write the production code that implements the feature or user requirement.
3. Write the tests to validate the feature or user requirement.
4. Run all the tests.
5. Refactor if necessary.

Table 2.2.1. Comparison of Test-First and Test-Last methodologies [1]

	Test-First	**Test-Last**
When are tests written?	Written *before* production code	Written *after* production code
When are tests run?	*Alongside* production code and frequently	*After* production code and less frequently

In summary, the Test-First methodology requires the creation of tests which incrementally develops small pieces of functionality until a feature is fully implemented. In contrast, the Test-Last methodology first develops the production code implementing a feature or user requirement and then writes the tests afterward.

3. Experimental design

This section outlines the initial goals of the study, describing the experiment design, proposing hypotheses, analyzing the study subjects, laying out the experimental procedure, and identifying the experiment variables and formalized hypotheses.

3.1. Goals

The goal of this experiment is to compare the Test-First methodology with the Test-Last methodology within an undergraduate software engineering capstone course. This experiment will evaluate the programmers' productivity, internal and external quality of the product, and the programmers' perception of the methodology.

3.2. Experiment variables and formalized hypotheses

The experiment examines a number of hypotheses that are summarized in Table 3.2.1. There were also some experiment variables to note and consider:

1. Two of the three teams that utilized a Test-First methodology used different application frameworks—Google Web Toolkit and Adobe Flex. The other team that utilized a Test-Last methodology used Google Web Toolkit.
2. One of the three teams used Adobe Flex Builder, an Eclipse based development environment, that offered a drag-and-drop interface.
3. The group that utilized a Test-First methodology using Google Web Toolkit underwent a personnel change (one person replaced) between the requirements elaboration and construction phase requiring some training in the new technology.

Table 3.2.1. Summary of hypotheses

Name	Null Hypothesis	Alternative Hypothesis
P1	$Prod_{TF} = Prod_{TL}$	$Prod_{TF} > Prod_{TL}$
C1	$\#Lines_{TF} = \#Lines_{TL}$	$\#Lines_{TF} > \#Lines_{TL}$
T1	$\#Tests_{TF} = \#Tests_{TL}$	$\#Tests_{TF} > \#Tests_{TL}$
T2	$\#TestCov_{TF} = \#TestCov_{TL}$	$\#TestCov_{TF} > \#TestCov_{TL}$
Q1	$IQltyCC_{TF} = IQltyCC_{TL}$	$IQltyCC_{TF} < IQltyCC_{TL}$
Q2	$IQltyWM_{TF} = IQltyWM_{TL}$	$IQltyWM_{TF} < IQltyWM_{TL}$
Q3	$EQlty_{TF} = EQlty_{TL}$	$EQlty_{TF} < EQlty_{TL}$
S1	$Stu_{TF} = Stu_{TL}$	$Stu_{TF} > Stu_{TL}$
S2	$Stu\|TF_{TF} = Stu\|TF_{TL}$	$Stu\|TF_{TF} > Stu\|TF_{TL}$

Hypothesis **P1** will examine whether the productivity, measured by the number of hours per number of features implemented, of Test-First programmers is higher than their Test-Last counterpart. The measurements are gathered through time logs and through the teams' input of what features were implemented. The analysis will be covered in **Section 4.2**.

Hypothesis **C1** will examine whether Test-First programmers produced more production code than their Test-Last counterpart. We will examine the number of lines of code written during production and not part of tests. The analysis will be covered in **Section 4.3**.

Hypothesis **T1** will examine whether Test-First programmers produced more tests than their Test-Last counterpart. The measurements are the number of lines of code written in the tests. The analysis will be covered in **Section 4.3**.

Hypothesis **T2** will examine whether Test-First programmers produced tests that covered more lines of production code than their Test-Last counterpart. Test coverage will be measured as the percent of number of production lines of code executed by the tests divided by the total number of production lines of code. The analysis will be covered in **Section 4.4**.

Hypothesis **Q1** will examine whether the internal quality, measured by cyclomatic complexity, of the production code by Test-First programmers is lower than their Test-Last counterpart. Cyclomatic complexity is "the number of branches in the module" [7]. The analysis will be covered in **Section 4.5**.

Hypothesis **Q2** will examine whether the internal quality, measured by weighted methods per class, of the production code by Test-First programmers is lower than their Test-Last counterpart. Weighted methods per class are the sum of the complexities of methods [8]. The measurements are gathered through automated metrics described in Section 3.3. The analysis will be covered in **Section 4.5**.

Hypothesis **Q3** will examine whether the external quality as measured by the total number of recorded defects of the production code by Test-First programmers is lower than their Test-Last counterpart. The measurements are gathered through integration tests to show the number of defects when modules are all compiled together. The analysis will be covered in **Section 4.6**.

Hypothesis **S1** will examine whether the programmers hold a higher opinion of the Test-First methodology than Test-Last. Hypothesis **S2** will examine whether the Test-First programmers favor the Test-First methodology more than Test-Last. The results of the programmers' opinions are gathered through a survey given out by the professor of the course. The analysis will be covered in **Section 4.7**.

3.3. Experiment Design

Three teams participated in this experiment, consisting of a total of fourteen students. Two of the three teams utilized a Test-First methodology while the remaining team utilized a Test-Last methodology. Although the students were part of a year-long capstone project, this experiment focused on the work done during the construction phase.

During the construction phase, the three teams worked from a common Software Requirement Specification (SRS) document approved by the representatives of the industry sponsor. The SRS described the functional requirements, quality attributes, and a number of use cases to be implemented.

Even though the three teams shared an SRS document, the technologies and third-party software packages used were not common. For a web application framework, two of the three teams used Google Web Toolkit (GWT) while the remaining team chose Adobe Flex. To control the variability of the two web application frameworks, the two teams utilizing the GWT were randomly split between the Test-First and Test-Last methodologies.

All of the participants were notified that they were part of a study on TDD for which they signed an agreement as required by the Cal Poly Human Subjects Committee.

3.4. Subjects

The study participants were all upper-level undergraduate and graduate students in the Computer Science and Software Engineering programs. The group sizes were kept between four and five people among a total of fourteen students. Although the students range in experience levels, the students were required to fulfill a number of course prerequisites including a two-quarter software engineering sequence and an intermediate individual design and development course. In addition, nearly all had hands-on work experience. All of the students were educated with the Test-First and Test-Last methodologies through lectures and student presentations.

3.5. Apparatus and Experiment Task

Three software packages were used during the experiment to collect metrics on each group's code base:

1. **EMMA**: a code coverage tool for Java.
2. **Chidamber and Kemerer Java Metrics (CKJM)**: a software metrics tool for Java.
3. **Metrics 1.3.6**: a software metrics tool for Eclipse.

A representative from each of the three groups was asked to collect the desired metrics using the above tools.

3.6. Procedure

Each group maintained a subversion code repository, recorded their time logs, and completed a survey on their perception of the Test-First and Test-Last methodology before and after the experiment.

After the construction phase, a committee was formed to collect the data, formalize hypotheses as outlined in **Section 3.2**, and conduct an analysis of the results. The committee consisted of representatives from each of the teams along with project leader.

The software metrics were gathered on each of the group's code repository using the software packages stated in **Section 3.5**. The time logs were collected through a spreadsheet collected by the professor on a weekly basis. The surveys were collected online and given to students before and after the experiment.

After all the data was gathered, statistical analysis programs were used to test some of the hypotheses with a resulting an analysis outlined in **Section 4**. The experiment was facilitated by a professor who oversees the year-long capstone project.

4. Data Analysis

The following sections provide an analysis of the data collected along with the interpretations of the various metrics obtained, including characterization of groups, productivity, code size and test density, line coverage from test density, internal and external quality, and programmer perceptions. Some of the groups were omitted with an "n/a" for not applicable because the data was unable to be obtained. A summary of the analysis and the hypotheses tests can be found in **Section 3.2**.

4.1. Characterization of Groups

The two experimental groups consisted nine subjects utilizing the Test-First methodology and five students utilizing the Test-Last methodology. To distinguish between the two teams utilizing the Test-First methodology, Test-First A was the team that utilized Adobe Flex and Test-First B was the team that utilized GWT.

4.2. Productivity

A hypothesis test, labeled **P1** from **Section 3.2**, examined whether the productivity, measured by the number of hours per number of features implemented, of Test-First programmers is higher than their Test-Last counterpart. Table 4.2.1 reports effort in terms of total hours spent in software construction, and the number of features implemented by team.

The Test-Last team was clearly more productive in terms of hours per feature. A two-sample t-test produced a p-value of 0.998, indicating that the P1 null hypothesis could not be rejected. Therefore, the productivity of the Test-Last programmers was not less than their Test-First counterparts, and in fact the opposite appears to be true.

Table 4.2.1. Team Productivity

	Test-Last	Test-First A	Test-First B
Number of Group Members	5	5	4
Number of Total Hours	169.05	140.25	133.8
Number of Features Implemented	12	6	6
Hours per feature	14.09	23.38	22.30

4.3. Code Size and Test Density

A hypothesis test, labeled **C1** from **Section 3.2**, examined whether Test-First programmers produced more production code than their Test-Last counterparts. A hypothesis test, labeled **T1** from **Section 3.2**, examined whether Test-First programmers produced more tests, measured by test code lines per source (production) code lines, than their Test-Last counterpart. Table 4.3.1 reports results on source and test code size. Although source size is very similar, the Test-Last team actually wrote four times as many lines of test code as both the Test-First teams combined. A two-sample t-test gives a p-value of 0.803 for hypothesis **C1 and a** p-value close to 1 for hypothesis **T1**. This indicated that neither null hypotheses can be rejected.

Table 4.3.1. Production and Test Code Size

Code and Tests	Test-Last	Test-First A	Test-First B
Source lines of code	3393	3358	2468
Test lines of code	4140	560	423
Test lines per source line	1.220	0.1668	0.1714

4.4 Line Coverage from Tests

A hypothesis test, labeled **T2** from **Section 3.2**, examined whether Test-First programmers produced tests that covered more lines of production code than their Test-Last counterpart.

Table 4.4.1 summarizes the data showing the line coverage of the production code by the student-written tests. The data was separated into two parts—production code that includes the graphical user interface (GUI) code

and another without the GUI code focusing on the process and system.

Because of technological differences, we were unable to collect data for the Test-First A team resulting in insufficient data to perform any statistical tests. Based on simple observation, the Test-Last group covered more lines of code for both the production with and without GUI.

Table 4.4.1 Line Coverage

Code and Tests	Test-Last	Test-First A	Test-First B
Line coverage incl. GUI	34%	n/a	10%
Line coverage excl. GUI	61%		23%

4.5 Internal Quality

A hypothesis test, labeled **Q1** from **Section 3.2**, examined whether the internal quality, measured by cyclomatic complexity, of the production code by Test-First programmers is lower than their Test-Last counterpart. A two-sample t-test was conducted with the summary of data in Table 4.5.1.

Since the cyclomatic complexity shows the number of paths through a source code, the analysis is composed of the classes that did not implement the graphical user interfaces. The consensus for this analysis was to focus on the classes containing the algorithm and logic because the classes to implement the graphical user interface were geared towards the cosmetics of the program.

The test gave a resulting p-value of 0.732. This indicated that the null hypothesis could not be rejected. Therefore, the cyclomatic complexity of the Test-Last groups did not differ from their Test-First counterpart.

Table 4.5.1 Cyclomatic Complexity

Cyclomatic Complexity	Test-Last	Test-First
Sample Size	76	69
Mean	3.00	3.64
Standard Deviation	2.70	8.11
P-value	0.732 (not rejected)	

A hypothesis test, labeled **Q2** from **Section 3.2**, examined whether the internal quality, measured by weighted methods per class, of the production code by Test-First programmers is lower than their Test-Last counterpart. A two-sample t-test was conducted with the summary of data in Table 4.5.2.

Since the weighted methods per class determines where a class should be refactored into more classes, the analysis is composed of all of the classes including the classes to implement the graphical user interfaces.

The test gave a resulting p-value near 0. This indicated that the null hypothesis is rejected. Therefore, the weighted methods per class for the Test-Last groups is higher than their Test-First counterpart.

Table 4.5.2 Weighted Methods per Class

Number	Test-Last	Test-First
Sample Size	128	49
Mean	7.66	4.04
Standard Deviation	6.75	2.63
P-value	0.000 (rejected)	

4.6 External Quality

A hypothesis test, labeled **Q3** from **Section 3.2**, examined whether the external quality, measured by the total number of recorded defects, of the production code by Test-First programmers is lower than their Test-Last counterpart. Table 4.6.1 summarizes the data showing the number of defects for each group.

Defect data was collected during the third quarter of the capstone project. At the beginning of this third quarter, the three groups reduced down to two groups—Test-Last and Test-First A. As a result, no defect data is available for Test-First B. The data does indicate that the Test-Last group had 39% more defects than their Test-First counterpart.

Table 4.6.1 Number of Defects

Code and Tests	Test-Last	Test-First A	Test-First B
Number of Defects	78	56	n/a

4.7 Programmer Perceptions

A hypothesis test, labeled **S1** from **Section 3.2**, examined whether the programmers hold a higher opinion of the Test-First methodology than Test-Last. A hypothesis test, labeled **S2** from **Section 3.2**, examined whether the Test-First programmers favor the Test-First methodology more than Test-Last. The results from the questionnaire can be seen in Table 4.7.1.

Ten out of the fourteen students preferred Test-First over Test-Last. Interestingly, for students who utilized the Test-First methodology, five out of nine preferred Test-First. Overall, students prefer Test-First over Test-Last.

Table 4.7.1 Student Opinions

Preference	Test-Last	Test-First
All students	4	10
Test-First students	4	5

5. Threats to Validity

The most obvious threat to validity was the small sample size of fourteen students. In addition, the differences in implementing the project were significant since two of the three teams implemented the project using GWT while the other team implemented the project using Adobe Flex. While GWT provided documentation to help program the widgets and set up the framework,

Adobe Flex provided a capability to easily drag and drop the widgets with the ability to export the program as a SWF file that is compatible with any Adobe Flash player.

Furthermore, TDD was also a relatively new concept to many of the students who were used to the traditional approach. Given this, some of the students reported challenges with applying the TDD process properly.

6. Conclusions and Future Work

The experiment evaluated effects of TDD conducted with undergraduate and graduate students in a year-long software engineering capstone course working alongside an industry sponsor and a professor. The study compared the Test-First methodology with Test-Last through the programmer's productivity, internal and external quality of the product, and the programmer's perceptions. The metrics were analyzed with a number of results through a statistical hypothesis testing.

In contrast to several previous studies, our data analysis indicates that the Test-First methodology did not outperform Test-Last in many of the measures. In fact, the Test-Last group appeared to be more productive than their Test-First counterpart in terms of hours per implemented feature and total features completed. All three teams wrote about the same amount of lines of production code but the group utilizing Test-Last outperformed the groups utilizing Test-First with more than seven times the amount of test code. The group utilizing Test-Last had higher line coverage than their Test-First counterpart. The only area where the group utilizing Test-Last didn't differ significantly from the groups utilizing Test-Last was in the cyclomatic complexity. The group utilizing Test-Last produced higher weighted methods per class than the Test-First groups indicating that they wrote larger, more complex classes. Concerning the programmer's perception, the results indicated a preference for Test-First, but they were not significant enough to state that the class, as a whole, preferred the Test-First methodology over Test-Last. This was the same perception for those who programmed with the Test-First methodology throughout the study.

The professor observed that the Test-First teams struggled to consistently and properly apply TDD. Students reported that the use of unfamiliar technologies (Flex and GWT) made learning and applying TDD particularly difficult.

Regarding the volume of test data reported in Table 4.3.1, although the Test-First and Test-Last teams had similar ratios of test lines of code to production lines of code through much of the software construction phase, the Test-Last team made a significant late effort to improve test-coverage percentages. This difference is attributed more to team dynamics and individual ambition than to the development approach applied.

In conclusion, the study does not imply a generalization to other contexts since multiple factors could have biased the results. For example, further studies are needed with a larger sample size and with differing programming experiences from students to professionals. In addition, better metric tools are able to provide better accounts of the programming experience. This would increase the validity of the TDD approach versus a traditional development approach to motivate others whether or not to adopt this different approach.

7. Acknowledgements

We would like to thank Kevin Carr and Ross Wampler for contributing to this research. We would like to acknowledge Cyril Aspuria, Victor Fehlberg, and Bryan Yee for their role as industry customer in this project. We would also like to thank all of the participants of the experiment whom we were fortunate enough to work with throughout the capstone project.

8. References

[1] H. Erdogmus, M. Morisio, and M. Torchiano, "On the Effectiveness of the Test-First Approach to Programming", *IEEE Transactions on Software Engineering*, vol. 31, no. 3, IEEE Press, Piscataway, New Jersey, USA, March 2005, pp. 226-237.

[2] L. Koskela, *Test Driven*, Manning Publications, Greenwich, Connecticut, USA, 2008.

[3] D. Janzen, and H. Saiedian, "Test-Driven learning in early programming courses", *ACM SIGCSE Bulletin*, vol. 40, no. 2, ACM, New York, New York, USA, 2008, pp. 532-536.

[4] D. Janzen, and Hossein S., "Does Test-Driven Development Really Improve Software Design Quality?", *IEEE Software*, vol. 25, no. 2, IEEE Press, Piscataway, New Jersey, USA, March/April 2008, pp.77-84.

[5] B. George, and L. Williams, "An initial investigation of test driven development in industry", *Symposium on Applied Computing*, ACM, New York, New York, USA, 2003, pp. 1135-1139.

[6] D. Janzen, "Software architecture improvement through test-driven development", *Conference on Object Oriented Programming Systems Languages and Applications*, ACM, New York, New York, USA, 2005, pp. 240-241.

[7] S. R. Schach, *Classical and Object-Oriented Software Engineering*, third edition, Richard D. Irwin, a Times Mirror Higher Education Group, Inc. company, 1996.

[8] J. Michura and M.A.M. Capretz, "Metrics suite for class complexity", Information Technology: Coding and Computing, vol. 2, no. 3, IEEE Press, Piscataway, New Jersey, USA, 2005, pp. 404-409.