

# Software Architecture Improvement through Test-Driven Development

David S. Janzen  
University of Kansas  
Electrical Engineering and Computer Science  
Lawrence, Kansas USA  
djanzen@ku.edu

## 1. PROBLEM & MOTIVATION

Despite a half century of advances, the software construction industry still shows signs of immaturity [1]. Professional software development organizations continue to struggle to produce reliable software in a predictable and repeatable manner. While a variety of development practices are advocated that might improve the situation, developers are often reluctant to adopt new, potentially better practices based on anecdotal evidence alone. Empirical evidence of a practice's efficacy are rarely available or conclusive and adopting new practices is time-consuming, expensive, and risky.

Test-driven development (TDD) is a new approach that offers the potential to significantly improve the state of software construction. TDD is a disciplined software development practice that focuses on software design by first writing automated unit-tests followed by production code in short, frequent iterations [3]. TDD focuses the developer's attention on a software's interface and behavior while growing the software architecture organically.

This research applies empirical software engineering techniques to examine the ability of TDD to produce better software designs than more traditional Test-Last approaches produce in terms of reusability, extensibility, and maintainability. In addition, this research examines the pedagogical implications of the approach and results.

If TDD proves to improve software quality at minimal cost, and if students can learn and benefit from TDD from early on, then this research can have a significant impact on the state of software construction. Software development organizations will recognize the benefits of TDD as both a design and testing approach, and they will be convinced to adopt TDD in appropriate situations. New textbooks and teaching materials can be written applying the test-driven learning approach developed in this research. As students learn to take a more disciplined approach to software development with TDD, they will carry this into professional software organizations and improve the overall state of software construction.

## 2. BACKGROUND & RELATED WORK

TDD has gained recent attention with the popularity of the Extreme Programming (XP) [3] agile software development methodology. Although TDD has been applied sporadically in various forms for several decades [12], possible

definitions have only recently been proposed. While some XP practices like pair programming have enjoyed significant research, advocates of TDD rely primarily on anecdotal evidence of TDD's benefits.

### 2.1 TDD Studies in Industry

A few evaluative research studies have looked at TDD as a testing practice to remove defects [8]. Three empirical studies on TDD were conducted in industry settings involving fairly small groups in at least four different companies [13, 7, 16]. These studies primarily examined defect density as a measure of software quality, although some survey data indicated that programmers thought TDD promoted simpler designs. In the George study, programmer experience with TDD varied from novice to expert, while the other studies involved programmers new to TDD.

These studies revealed that programmers using TDD produced code which passed between 18% and 50% more external test cases than code produced by the corresponding control groups. The studies also reported less time spent debugging code developed with TDD. Further they reported that applying TDD had from minimal impact to a 16% decrease in programmer productivity. In other words, applying TDD sometimes took longer than not using TDD. In the case that took 16% more time, it was noted that the control group also wrote far fewer tests than the TDD group.

### 2.2 TDD Studies in Academia

Five studies were conducted in academic settings that report empirical results. When referring to software quality, all but one [11] of the empirical studies focused on the ability of TDD to detect defects early. Two [5, 11] of the five studies reported significant improvements in external software quality and programmer productivity. One [6] reported a correlation between number of tests written and productivity. In this study, students using Test-First wrote more tests and were significantly more productive. The remaining two [14, 15] studies reported no significant improvements in either defect density or productivity. All five studies were relatively small and involved only a single semester or less. In all studies, programmers had little or no previous experience with TDD.

### 2.3 TDD and Design

As a member of the Extreme Programming 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 on 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).

According to XP and TDD pioneer Ward Cunningham, "Test-first coding is not a testing technique." In fact TDD goes by various names including test-first programming, test-driven design, and test-first design. The *driven* in TDD focuses on how TDD informs and leads analysis, design and programming decisions. TDD assumes that the software design is either incomplete, or at least very pliable and open to evolutionary changes.

The empirical studies to date have focused on TDD as a testing technique for improving external quality by discovering and eliminating defects. However, there is no research on the broader efficacy of TDD, nor on its effects on internal design quality outside a pilot study for this work [11]. Further, no empirical research has examined the appropriate place or teaching techniques for introducing TDD in the undergraduate curriculum.

### 3. UNIQUENESS OF THE APPROACH

This research is the first comprehensive evaluation of how TDD affects overall software architecture quality beyond just defect density. Empirical software engineering techniques were applied to evaluate the ability of TDD to produce better software designs than more traditional Test-Last approaches produce in terms of reusability, extensibility, and maintainability.

In addition, this research makes important pedagogical contributions. A new teaching approach called "test-driven learning" [10] is introduced that incorporates teaching with automated tests. The research demonstrates that undergraduate computer science students can learn to apply TDD, and it examines at what point in the curriculum TDD is best introduced.

#### 3.1 Experiment Design

The goal of this experiment is to compare iterative Test-First programming (TDD) with Test-Last programming for the purpose of evaluating internal quality, programmer productivity, and programmer perceptions. A series of simultaneous experiments was conducted in three undergraduate computer science courses, one graduate software engineering course, and one Fortune 500 company. The studies in undergraduate courses were conducted simultaneously in CS1/CS101 (Computer Programming 1), CS2/CS102 (Computer Programming 2 /Data Structures), and SE/CS391 (Software Engineering). In addition, a case study analyzing previous projects was performed with the company.

Programmers in each study completed a pre-experiment survey on their programming experience and attitudes toward software testing and design. Programmers were divided into balanced control and study groups within each experiment, subject to the constraints of each course.

Undergraduate programmers were taught to write auto-

mated unit-tests integrated with course topics using a new approach called test-driven learning (TDL) [10]. The TDL approach involves modeling regular unit-testing in lecture and lab instruction through examples with automated tests. Most commonly, output statements are replaced with automated unit tests to demonstrate both the interface and the behavior of the code under investigation. Graduate and professional programmers were given more concentrated instruction on TDD and the use of automated unit-test frameworks. Both the control and the study groups completed the same set of labs and training on testing, writing unit tests, and using automated test frameworks. Instruction was given on both Test-First and Test-Last development approaches.

Students were then required to complete one or two programming assignments. The study group was asked to use test-driven development techniques while the control group was asked to perform iterative Test-Last development. In the early courses where two smaller assignments were required, the second assignment built on or reused significant parts of the first.

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. In the second project, students could choose to build on either their own solution, or the solution provided.

Students were required to track the amount of time they spent on programming projects. At the end of each experiment, programmers were asked to complete a survey indicating their attitudes toward testing and test-driven development. Student exam and course grades were compared to determine if any correlation exists between test-driven development and academic performance.

The following semester, a sample of students from both the control and study groups will again be examined to determine the voluntary use of test-driven development in course programming assignments. Students from CS1 will be examined in CS2. Students from CS2 will be examined in the SE course. Students from the SE course will be examined in a subsequent course if a significant enough number of them enroll in a common programming-based course.

#### 3.2 Formal Hypotheses

Several hypotheses are examined. A formalization of the hypotheses is presented in Table 1. Each of these hypotheses is discussed in turn here.

Hypothesis **P1** considers whether Test-First (TDD) programmers are more productive than Test-Last programmers. Development time, effort per feature, and effort per lines of code will be examined.

Some sources[2, 4] claim that Test-First programmers consistently write a significant amount of test code. Our first 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 man-

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	$\text{IntQty}_{TF} = \text{IntQty}_{TL}$	$\text{IntQty}_{TF} > \text{IntQty}_{TL}$
Q2	$\text{IntQty} \text{Tested}_{TF} = \text{IntQty} \text{Not-Tested}_{TF}$	$\text{IntQty} \text{Tested}_{TF} > \text{IntQty} \text{Not-Tested}_{TF}$
O1	$\text{Op}_{TF} = \text{Op}_{TL}$	$\text{Op}_{TF} > \text{Op}_{TL}$
O2	$\text{Op} \text{TF}_{TF} = \text{Op} \text{TF}_{TL}$	$\text{Op} \text{TF}_{TF} > \text{Op} \text{TF}_{TL}$

**Table 1: Formalized Hypotheses**

ner 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 programmers will rarely achieve such high test-coverage.

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

## 4. RESULTS & CONTRIBUTIONS

An extensive analysis of all software produced in the experiments is currently being conducted. Internal quality is somewhat subjective and measures are 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. Over fifty structural and object-oriented metrics have been collected and are currently being analyzed statistically. Analysis of variance and paired t-tests are being calculated to determine differences between the Test-First and Test-Last groups in each experiment. Subjective measures of design quality are also being evaluated. Although the analysis is not complete, a number of early results can be reported here.

### 4.1 Initial Industry Results

The first of two result sets comes from the case study being conducted in a large corporation. A set of Test-First and Test-Last projects were developed by the same group of programmers. The data presented here represents recent Test-First and Test-Last software projects as well as some Test-Last projects produced over a span of several years. Table 2 and 3 summarizes results of method-level metrics performed on approximately 4,600 methods representing over 37,000 lines of code in fourteen primarily web-based Java software projects.

The data indicates that software developed in a Test-First manner is likely to have fewer statements (NOS and MLOC), more exceptions (NOE), and a lower computational complexity (V(G)) and nested block depth (NBD). Test-First code is also likely to be simpler as measured with Halstead’s length and level metrics (AHL and LVL), and take less effort (EFF) with fewer expected defects (BUG). Finally, Test-First methods are likely to have more parameters (PAR).

Additional analysis is being completed on class and project level metrics, as well as to only examine recent Test-First and Test-Last projects, thus eliminating the validity threat

that programmers produce better code now than they did a couple of years ago. However, the results presented indicate that this research may have a significant impact on professional practice once published.

### 4.2 Initial Academic Results

The second set of results presented here come from the first academic study conducted in Summer 2005 with undergraduate students in an upper-level software engineering course. These results will be presented in April at the Conference on Software Engineering Education and Training [9].

The experiment consisted of three student groups. All three developed the same HTML pretty-print system. 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 reasonable previous Java experience. Test-First/Test-Last team assignments were made after analyzing the pre-experiment questionnaire to ensure the teams were reasonably balanced. One team applied Test-First programming. A second team applied Test-Last programming. Despite being instructed to write automated unit tests, the final Test-Last team reported that they “ran out of time” and performed only manual testing. This team will be labeled the “No-Tests” team.

#### 4.2.1 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.

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. This data indicates a possible trend toward rejecting the **P1** null hypothesis, making it likely that Test-First programmers are more productive than Test-Last programmers. 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, the pre-experiment questionnaire indicates that there was no statistically significant difference in the academic or practical background of the teams.

#### 4.2.2 Code Size and Test Density

Table 4 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 application

Metric	NOS	NOE	V(G)	PL	AHL	VOC	VOL
Analysis of Variance (ANOVA) Results							
F-ratio	1.6012	176.93	7.4733	0.34842	2.9382	0.13857	2.7015
p-value	0.2058	0.0001	0.0063	0.555	0.0866	0.7097	0.1003
Significant	No	Yes	Yes	No	No	No	No
Paired t-test							
Diff b/w Means	1.26	-0.36	0.50	4.64	12.64	0.48	82.77
t-Statistic	2.398	-9.792	6.413	0.8795	3.531	0.4784	3.779
df	320	229	407	273	346	257	396
p-value	0.0171	0.0001	0.0001	0.3799	0.0005	0.6328	0.0002
Significant	Yes	Yes	Yes	No	Yes	No	Yes
Higher Method	TL	TF	TL	TL	TL	TL	TL

**Table 2: Analysis of Method Metrics on Industry Data, Part 1**

Metric	LVL	PD	EFF	BUG	MLOC	NBD	PAR
Analysis of Variance (ANOVA) Results							
F-ratio	40.905	1.4308	2.0563	2.7029	8.4694	16.051	70.739
p-value	0.0001	0.2317	0.1516	0.1002	0.0036	0.0001	0.0001
Significant	Yes	No	No	No	Yes	Yes	Yes
Paired t-test							
Diff b/w Means	0.16	0.46	2.70	0.028	4.11	0.24	-0.46
t-Statistic	7.509	1.842	5.598	3.78	7.477	4.384	-5.621
df	249	278	2474	396	470	244	226
p-value	0.0001	0.0665	0.0001	0.0002	0.0001	0.0001	0.0001
Significant	Yes	No	Yes	Yes	Yes	Yes	Yes
Higher Method	TL	TL	TL	TL	TL	TL	TF

**Table 3: Analysis of Method Metrics on Industry Data, Part 2**

implemented more additional features besides the graphical user interface, the GUI was a significant feature and this allows a more consistent comparison with the two teams that only implemented a text user interface.

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.

Test density and coverage measurements revealed that 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 the tests written by the Test-Last team.

### 4.2.3 Internal Quality

Structural and object-oriented metrics were calculated for all software to gauge the internal quality of each. While most metrics produced comparable and acceptable values for all three projects, some warnings were noted regarding complexity. In particular, the Test-First and No-Tests code contained some methods with out-of-range Nested Block Depth and Cyclomatic Complexity measures.

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 5 reports differences with Weighted Methods per Class (WMC), Coupling Between Objects (CBO), Nested Block Depth (NBD), 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. This difference is approaching statistical significance at  $p=.08$ . In addition, tested classes had 104% lower coupling measures than untested classes.

### 4.2.4 Programmer Perceptions

Pre and post-experiment surveys were administered to all programmers. Comparisons between the two surveys revealed that all three teams perceived the Test-First approach more positively after the experiment (up to 39% more) and inversely perceived the Test-Last approach more negatively (up to 30% more). 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 use it again, supporting the rejection of the **O2** null hypothesis. All programmers from the No-Tests team indicated they would prefer to use Test-Last, causing us to keep the **O1** null hypothesis. 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

Team	# of classes	LOC	Test LOC	LOC/method	LOC/feature
Test-First	13	1053	168	12.10	87.75
Test-First(less 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 4: Code Size Metrics

Code	WMC		CBO		NBD		Complexity		# Parameters	
	average	max	average	max	average	max	average	max	average	max
Tested	7.80	21	2.2	3	1.50	3	1.77	5	1.00	3
Non-Tested	13.55	53	<b>4.5</b>	<b>20</b>	<b>2.20</b>	<b>6</b>	<b>2.53</b>	<b>13</b>	<b>0.48</b>	<b>6</b>

Table 5: Metrics on Tested and Untested code of Test-First Project

future changes to their software. This difference with the Test-Last team was nearly statistically significant ( $p=.059$ ).

### 4.3 Conclusions

The two sets of results provide a taste of the analysis to be completed in this research. Early results from analyzing the remaining experiments indicates that while many aspects are unaffected by the Test-First/Test-Last choice, some important measures have significant differences. For instance, in general computational complexity is significantly lower in Test-First projects, coupling is significantly higher in Test-First projects, and test volume and coverage is significantly higher in Test-First projects. Experiments in the early programming courses indicates that the test-driven learning approach is effective in teaching testing intrinsically with core topics, but that additional motivation must be provided to keep programmers writing automated tests.

Professional programmers will benefit from the results of this study in two primary ways. First, the focus on the effects of TDD on internal design quality will emphasize the essence of TDD as a design mechanism and will help dispel misconceptions of TDD as a testing approach. Second, the significant effects of TDD on development aspects such as complexity, productivity, and coupling will inform adoption decisions.

Likewise, academics will benefit by adopting the TDL teaching approach to teach testing for free. It is expected that textbooks and instructional materials will emerge incorporating this approach. In addition, for the first time, educators will have empirical evidence enabling decisions regarding the appropriate placement of TDD in the curriculum.

## 5. REFERENCES

- [1] 2004 third quarter research report. Technical report, Standish Group International, Inc., 2004.
- [2] D. Astels. *Test Driven Development: A Practical Guide*. Prentice hall PTR, 2003.
- [3] K. Beck. Aim, fire. *Software*, 18(5):87–89, Sept.-Oct. 2001.
- [4] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.
- [5] S. Edwards. Using test-driven development in the classroom: providing students with automatic, concrete feedback on performance. In *Proceedings of the International Conference on Education and Information Systems: Technologies and Applications EISTA '03*, August 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] B. George and L. Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337–342, 2004.
- [8] D. Janzen and H. Saiedian. Test-driven development: concepts, taxonomy and future directions. *IEEE Computer*, 38(9):43–50, Sept 2005.
- [9] D. Janzen and H. Saiedian. On the influence of test-driven development on software design. In *CSE&T*, pages –, 2006.
- [10] 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 –. ACM Press, 2006.
- [11] R. Kaufmann and D. Janzen. Implications of test-driven development: a pilot study. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 298–299. ACM Press, 2003.
- [12] C. Larman and V. R. Basili. Iterative and incremental development: a brief history. *IEEE Computer*, 36(6):47–56, June 2003.
- [13] E. M. Maximilien and L. Williams. Assessing test-driven development at IBM. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 564–569, Piscataway, NJ, May 3–10 2003. IEEE Computer Society.
- [14] M. Müller and O. Hagner. Experiment about test-first programming. *IEEE Proceedings-Software*, 149(5):131–136, 2002.
- [15] M. Pančur, M. Ciglarič, M. Trampuš, and T. Vidmar. Towards empirical evaluation of test-driven development in a university environment. In *Proceedings of EUROCON 2003. Computer as a Tool. The IEEE Region 8*, volume 2, pages 83–86, 2003.
- [16] L. Williams, E. 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.