A Structured Experiment of Test-Driven Development

Boby George
Department of Computer Science
Virginia Polytechnic Institute and State University
Falls Church, VA 22043 USA
(+1) 703 893 0180
boby@vt.edu

Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8207 USA
(+1) 919 513 4151
williams@csc.ncsu.edu

ABSTRACT

Test Driven Development (TDD) is a software development practice in which unit test cases are incrementally written prior to code implementation. We ran a set of structured experiments with 24 professional pair programmers. One group developed a small Java program using TDD while the other (control group), used a waterfall-like approach. Experimental results, subject to external validity concerns, tend to indicate that TDD programmers produce higher quality code because they passed 18% more functional black-box test cases. However, the TDD programmers took 16% more time. Statistical analysis of the results showed that a moderate statistical correlation existed between time spent and the resulting quality. Lastly, the programmers in the control group often did not write the required automated test cases after completing their code. Hence it could be perceived that waterfall-like approaches do not encourage adequate testing. This intuitive observation supports that perception that TDD has the potential for increasing the level of unit testing in the software industry.

Keywords

Software Engineering, Test Driven Development, Extreme Programming, Agile Methodologies.

1. INTRODUCTION

Test Driven Development (TDD) [2], a software development practice used sporadically for decades [10, 14], has gained added visibility recently as a practice of Extreme Programming (XP) [1]. The practice involves the implementation of a system starting from the unit test cases of an object. Writing test cases and implementing that object or object methods, then triggers the need for other objects/methods. An important rule in TDD is: "If you can't write a test for what you are about to code, then you shouldn't even be thinking about coding." [6]

An object is the basic building block of Object-Oriented Programming. Unless objects are designed judiciously, dependency problems, such as tight coupling of objects and fragile super classes (inadequate encapsulation) can creep in. These problems could result in a large complex code base that compiles and runs slowly. XP originator Kent Beck asserts, "Test-first code tends to be more cohesive and less coupled than code in which testing isn't a part of the intimate coding cycle." [3] TDD proponents argue that reduced coupling occurs because the practice guides them to the building of objects that are actually needed (to pass test cases based on the requirements) rather than building objects that are thought to be needed (due to possible improper understanding of requirements). Moreover, TDD enables continuous regression testing, which improves code quality [2].

Software practitioners can be concerned about the lack of upfront design in TDD and the need to make design decisions at every stage. This concern necessitates the need to empirically analyze and quantify the effectiveness of this practice.

The research outlined in this paper empirically examines the following two hypotheses:

1. The TDD practice will yield code with superior external code quality when compared with code developed with a waterfall-like practice. External code quality will be assessed based on the number of functional, black-box test cases passed.
2. Programmers who practice TDD will develop code faster than programmers who develop code with a more traditional waterfall-like practice. Programmers' productivity will be measured by the time (hours) to complete the development.

To investigate these hypotheses, research data were collected from three sets of structured experiments conducted with professional programmers.

2. BACKGROUND AND RELATED WORK

In this section, we first describe the TDD practice. Then, we describe two empirical studies of TDD.

2.1 Test-Driven Development

The TDD practice starts with thoughts on how to test the required functionality. After writing automated test cases that generally will not even compile, the programmers write implementation code to pass these test cases. The work is kept within programmer's intellectual control; as the programmer is continuously making small implementation decisions and increasing functionality at a relatively consistent rate. All of the test cases that exist for the entire program must successfully pass before new code is considered fully implemented. Hence it is perceived,

with a degree of confidence, that the new code will not introduce a fault or mask a fault in the current code base. Another thumb rule in TDD is that whenever a software defect is found, unit test cases are added to the test suite prior to fixing the code.

The following is a theoretical analysis on the professed shortcomings and benefits of TDD.

2.1.1 Shortcomings

Lack of Design. Sometimes, practitioners who utilize TDD begin development with some design activities. However, TDD often does not include any upfront design. Hence the applicability of the later approach is limited by the comprehension capacity of programmers' minds. Further, practitioner van Deursen asserts that the TDD philosophy of having zero to very little design works, only when (1) the team has a good understanding of code base (2) the code is in good shape [21]. He further asserts that the practice can suffer from lack of conceptual integrity [21] (note that Brooks contends that conceptual integrity is the most important consideration in system design [4]). Finally, van Deursen asserts that the practice's overall philosophy is high risk/high return: if TDD works it can lead to time and cost saving, but if it fails, then there is no normal defense as with explicit design and documentation [21].

Researchers have noted that over a period of time, the techniques and notations developed for software design have been integrated into the implementation process. Such integration has tended to blur, if not confuse, the distinction between design and implementation [9, 19]. The TDD practice also blurs the distinct phases of program development (design, code, and test). Since the implementation process, focuses more on how the elements need to be implemented and less on the logical structure, it can be argued that faithful adoption of TDD might result in missing the macro or complete picture of the software.

Applicability of Practice. Some codes are inherently hard to test using TDD (for example GUIs [3]). Further, the TDD practice requires considerable effort to be expended on writing mock test objects. Additionally, since no formal documentation takes place, the rationale behind important decisions is not documented and can get lost.

Reliance on Refactoring. TDD utilizes refactoring and rigorous testing to achieve code understanding and to manage code complexity. The second law of software evolution states "As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it." [15] Refactoring is essential for maintaining or reducing complexity in TDD-developed code.

Skill level. Writing test cases for hard-to-test code requires a high level of experience and determination from programmers. Average programmers might lack the required level of experience, resulting in code without proper test cases or documentation [21]. Further, practitioners have reported that maintaining test assets requires special skills [21, 22].

2.1.2 Benefits

Program Comprehension. Studies indicate that about half of programmers' task during software maintenance is involved in understanding code [7]. The TDD approach helps in program comprehension because it encourages programmers to explain their code using test cases and code itself, rather than by using descriptive words. Secondly, it ensures that the test cases are up to date. However the practice does have the paradox that to understand one piece of code, the reader has to go through another piece of code (test code) and the code itself, a good rendering of the measure twice, cut once principle.

Efficiency. TDD proponents believe that **t**he fine granularity of the test-then-code cycle gives continuous feedback to programmer. With TDD, faults are identified quickly as new code is added to the system; hence the source of the problem is more easily determined. Based on prior research [17, 23], we think that the efficiency of fault/defect removal and the corresponding reduction in the debug time compensate for the additional time spent writing and executing test cases.

Test Assets. TDD enables testability. The use of the TDD practice drives programmers to write code that is automatically testable, such as having functions/methods returning a value that can be checked against expected results. The automated unit test cases written with TDD are valuable assets to the project. Subsequently, when the code is enhanced or maintained, running the automated unit tests may be used to identify newly-introduced defects and to control the uniformity over several releases of the product, i.e., for regression testing.

Reducing Defect Injection. Hamlet and Maybee assert that debugging and software maintenance are often perceived as a low-cost activity in which a working code defect is patched to alter its properties, and specifications and designs are neither examined nor updated [12]. Unfortunately, such fixes and small code changes may be nearly 40 times more error prone than new development [13], and often new faults are injected during debugging and maintenance. The suite of automated test cases are used as a fine-granularity, low-level regression test. By continuously running these automated test cases, one can find out whether a change breaks the existing system.

2.2 Related Research

Recently, researchers have started to conduct studies on the effectiveness of the TDD practice. Two such studies related to our work. These are now described.

University of Karlsruhe Experiment. Müller and Hagner [18] conducted a structured experiment comparing

TDD with waterfall (code then test) programming. The experiment, conducted with 19 graduate students, measured the effectiveness of TDD in terms of (1) development time, (2) resultant code quality, and (3) understandability. The researcher divided the experiment subjects into two groups, TDD and control, with each group solving the same task. The task was to complete a program in which the specification was given along with the necessary design and method declarations. The students completed the body of the necessary methods. The researchers set up the programming in this manner to facilitate objective and randomized automated acceptance testing for their analysis.

The TDD group wrote all test cases prior to starting any implementation code. The control group students wrote automated test cases after completing the code. The experiment occurred in two phases, an implementation phase (IP) followed by an acceptance test phase (AP). After IP, the students were made aware of the acceptance test cases they did not pass. They then were given the opportunity to correct their code. The researchers found no difference between the groups in overall development time. The TDD group had lower reliability after the IP phase and higher reliability after the AP phase. However, the TDD groups had statistically significant fewer errors when code was reused. Based on these results, the researchers concluded that writing programs in test-first manner neither leads to quicker development nor provides an increase in quality. The understandability of the TDD programs was higher, measured in terms of proper reuse of existing interfaces.

IBM Case Study. A TDD case study was run with an IBM software development team [17, 23]. This IBM group has been developing device drivers for over a decade. They have one legacy product which has undergone seven releases since late 1998. This legacy product was used as the baseline in the case study. In 2002, the group developed device drivers on a new platform. In the case study, the seventh release on the legacy platform was compared with the first release on the new platform. Because of its longevity, the legacy system handles more classes of devices on more platforms with more vendors than the new system. The legacy software was an adequate comparison for providing insight into the performance of the TDD methodology.

In the legacy product, the IBM team historically had used only ad-hoc testing techniques. For the new platform, they created 2,400 automated unit test cases after they had completed UML class and sequence diagrams. The team realized about a 40% reduction in function verification test defect density (defects/line of code) of new/changed code when compared with an experienced team who used an ad-hoc testing approach for the legacy product. They achieved this result with no discernable impact to programmer productivity. As usual, empirical concerns with case studies involve the internal validity of the research, or the degree of confidence and generalization in a cause-effect relationship between factors of interest and the observed results [5].

3. RESEARCH APPROACH

Our experimental trial results [11] with professional programmers add to the family of TDD experiments.

3.1 Experiment Details

We ran experimental trials with eight-person groups of programmers at three companies (John Deere, RoleModel Software, and Ericsson). In each of the experimental trials, the programmers were randomly assigned to one of two groups: TDD and control. All programmers used the pair-programming practice [24]. Each pair was asked to develop a bowling game application (adapted from an XP episode [16]) according to a set of requirements. The control group pairs used a conventional design-develop-test (similar to waterfall) [20] approach. Participants were asked to turn in their programs upon completing the activities as outlined. Then, their projects were assessed.

It was presumed that professional programmers would write code to handle all error conditions gracefully. However, our first trial results indicated that the pairs determined their implementation was complete when they could pass our specified acceptance test cases. Therefore, in the latter two trials, the experiment conditions were modified. All the programmers were asked to handle error conditions gracefully and were not provided acceptance test cases. Additionally, in the second two trials, the control group programmers were asked to write automated test cases after development.

The effectiveness of TDD was analyzed based on the time taken to develop and on the results of black-box functional testing. The quality of the test cases written by TDD programmers was measured using code coverage analysis. We supplemented our findings with survey data on the perceptions the participants had about TDD practice.

3.2 External Validity

An important consideration in empirical research design is external validity, that is, the ability of the experimental results to apply to the world outside the research situation. The strength of our results is that the experiment was done with practitioners in their own working environment. However, there are five important limitations that restrict the external validity of our experiment.

- Our sample size was relatively very small (6 TDD pairs, 6 control group pairs).
- As stated in the experiment details sections, after reviewing the results of the first trial, we modified the experiment instructions for the trials that followed. Unfortunately, only one control group pair actually

wrote any worthwhile automated test cases, despite the fact that they were specifically instructed to do so.

- All programmers worked in pairs. John Deere and RoleModel had used the pair programming practice in their day-to-day development, and Ericcson was introduced to the practice. Although not required in TDD, pair programming was used to accommodate the objective of experiment (to evaluate the effectiveness of TDD in the day-to-day development environment). Therefore, our results apply to the combination of TDD with pair programming.
- The application used in the evaluation process was very small (typical size of the code was 200 LOC).
- The subjects of the experiments had varying experience with TDD (from novice to expert). The third set of professional programmers had only three weeks of experience with TDD and pair programming before the experiment. Hence, it is conceivable that the TDD and pair programming approaches were not stabilized with these subjects.

4. EXPERIMENT RESULTS

We now provide the results of our quantitative and qualitative analysis.

4.1 Quantitative Analysis

The external code quality and productivity differences between the TDD and the control group were analyzed and quantified. Additionally, the test coverage of the TDD pairs was examined. However, the validity of the results must be considered within the context of the limitations discussed in external validity section.

4.1.1 External code quality

We developed 20 black-box test cases to evaluate the external code quality of professional programmers' code. The test cases validated the degree to which requirement specifications were implemented and the robustness of the code. The TDD pairs' code passed approximately 18% more test cases than the control group pairs. Figure 1 shows the box plot for the test cases passed. In the box plot, the edges of the box mark the 25th and 75th percentiles, while the horizontal line at the center of box marks the median of distribution. The median value for the TDD programmers' code is higher than of the control group programmers' median.

A hypothesis of this research was that the TDD practice would yield code with superior external code quality. Based on the data analysis conducted, the experimental findings are supportive that the TDD practice yields code with superior external code quality.

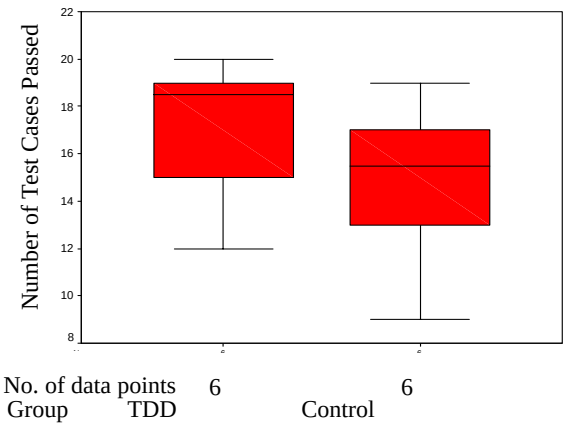

Figure 1: Box Plot for Test Cases Passed

4.1.2 Productivity

As shown in Figure 2, on average the TDD pairs took approximately 16% more time to develop the application than the control group pairs. The medians of the two groups are nearly equal. However, the upper range value is higher for the TDD programmers.

An important consideration in this analysis is that the control pairs were asked to write test cases after they developed code. However, only one group wrote any worthwhile test cases. This resulted in an uneven comparison of the time taken and hence a limitation to this study. There are benefits resulting from the test cases created by the TDD programmers. First, the TDD pairs produced test assets along with the implementation code. Second, the code developed is testable.

It was hypothesized that programmers who practice TDD will be more productive, as measured by the time to complete a program. However, contrary to our hypothesis, the experiment results showed the TDD programmers took approximately 16% more time than the control group programmers.

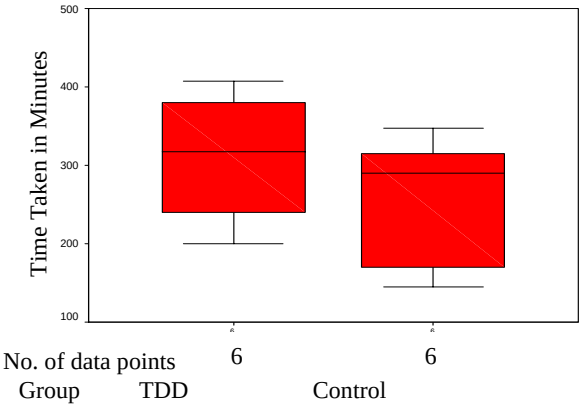

Figure 2: Box Plot of Time Taken by Programmers

4.1.3 *Correlating Productivity and Quality*

On average, the TDD pairs produced higher quality code. However, they took longer time, on average, to complete this work. On analyzing the results of all 12 pairs, we found a moderate correlation between the time spent and the resulting quality. The two-tailed Pearson Correlation had a value of 0.661, which was significant at the 0.019 level. This analysis indicates that the higher quality may be the result of the increased time taken by the TDD pairs and not solely due to the TDD practice itself.

4.1.4 *Code coverage*

We analyzed code coverage as an indication of the quality of the test cases written by TDD programmers. The industry standard for coverage is in the range 80% to 90%, although ideally the coverage should be 100% [8]. As shown in Figure 3, on average the TDD programmers surpassed the industry standards in all the three types of code coverage. The TDD programmers' test cases achieved a mean of 98% method, 92% statement and 97% branch coverage. The testing tool used, JUnit, cannot test the main method (of Java code), and hence the main method was excluded from code coverage analysis. Including the main method into the code coverage analysis would have lowered the TDD programmers' coverage results.

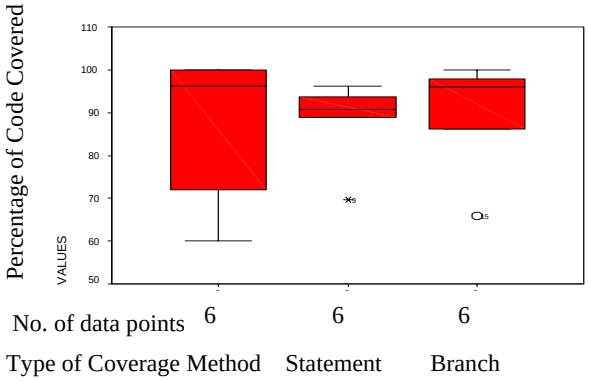

Figure 3: Box Plot of Code Coverage

4.2 Qualitative Analysis

A survey was conducted to the 24 professional programmers. The survey, administrated before the experiment, consisted of nine close-ended questions. The nine close-ended questions were aimed at eliciting the programmers' opinion on three main concerns:

(1) How productive is the practice for programmers?
(2) How effective is the practice?
(3) How difficult is the practice to adopt?

A reliability analysis was performed to determine whether it was statistically valid to aggregate the responses of the nine questions into the stated three concerns, using the Cronbach's Coefficient Alpha test. The alpha test measures the level of consistency of survey responses. This provides an indication on whether the individuals answered all of the questions within the subscale similarly, to aggregate the nine questions into the said concerns. All the survey responses were statistical significant at the 0.01 level ($p < 0.01$), indicating that the aggregation was valid. The statistical significance of each response was then evaluated using the Spearman's Rho test. The results of the survey are found below in Table 1. (Note: the results of only eight of the nine questions is displayed because two of the closed ended questions addressed the same area.)

Table 1: Survey Results

Concern/Sub-concerns	% Agree
Productivity – Aggregate	**78%**
Facilitates better requirements	88%
Reduces debugging effort	96%
Reduces development time	50%
Effectiveness -- Aggregate	**80%**
Yields higher code quality	92%
Promotes simpler design	79%
Is noticeably effective	71%
Difficulties in adoption -- Aggregate	**40%**
Getting into TDD mindset	56%
Lack of upfront design a hindrance	23%

Based on survey comments, it can be concluded that programmers generally feel that TDD is effective in terms of code quality and improves programmers' productivity. However, getting into TDD mindset is difficult. Lastly, some programmers expressed concerns about the increase in development time needed to write the test cases.

5. CONCLUSIONS AND FUTURE WORK

A series of experiments were conducted to examine the TDD practice. Specifically, the following hypotheses were tested and corresponding conclusions were obtained, subject to the limitations of the study:

- TDD practice appears to yield code with superior external code quality, as measured by conformance to a set of black-box test cases, when compared with code developed with a more traditional, waterfall-like model practice.
- The experiment results showed that TDD programmers took more time (16%) than control group programmers. However, the variance in the performance of the teams was large and these results are only directional. Additionally, the control group pairs did not primarily write any worthwhile automated test cases, making the comparison uneven.

- On an average, survey results indicate that, 80% of the professional programmers thought TDD was an effective practice and 78% believed the practice improves programmers' productivity. The survey results are statistically significant.
- Survey results also indicated that TDD practice facilitates simpler design and that lack of upfront design is not a hindrance. However, for some, transitioning to the TDD mindset is difficult.

Further controlled studies on a larger scale in industry and academia could strengthen or disprove these findings.

6. ACKNOWLEDGMENTS

We wish to thank the software programmers at John Deere, RoleModel, and Ericsson who participated in this research. We would also like to thank the North Carolina State University Software Engineering research group for their helpful suggestions on this paper. This research was funded in part by AT&T.

7. REFERENCES

[1] K. Beck, *Extreme Programming Explained: Embrace Change*. Reading, Massachusetts: Addison-Wesley, 2000.
[2] K. Beck, *Test Driven Development: By Example*: Addison Wesley, 2002.
[3] K. Beck, "Aim, Fire," in *IEEE Software*, vol. 18, September/October 2001, pp. 87-89.
[4] F. P. Brooks, *The Mythical Man-Month*: Addison-Wesley Publishing Company, 1995.
[5] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Design for Research*. Boston: Houghton Mifflin Co., 1963.
[6] D. Chaplin, "Test First Programming," *TechZone*, 2001.
[7] T. A. Corbi, "Program Understanding challenge for the 1990s," *IBM Systems Journal*, vol. 28, pp. 294-306, 1989.
[8] S. Cornett, "Code Coverage Analysis," Bullseye Testing Technology 2002.
[9] B. Foote and J. Yoder, "Big Ball of Mud," presented at Fourth Conference on Patterns Languages of Programs, Monticello, Illinois, September 1997.
[10] D. Gelperin and W. Hetzel, "Software Quality Engineering," presented at Fourth International Conference on Software Testing, Washington, DC, June 1987.
[11] B. George, "Analysis and Quantification of Test Driven Development Approach MS Thesis," in *Computer Science*. Raleigh, NC: North Carolina State University, 2002.
[12] D. Hamlet and J. Maybee, *The Engineering of Software*. Boston: Addison Wesley, 2001.
[13] W. S. Humphrey, *Managing the Software Process*. Reading, Massachusetts: Addison-Wesley, 1989.
[14] C. Larman and V. Basili, "A History of Iterative and Incremental Development," *IEEE Computer*, vol. 36, pp. 47-56, June 2003.
[15] M. M. Lehman and L. Belady, *Program Evolution: Processes of Software Change*. London: Academic Press, 1985.
[16] C. R. Martin, *Advanced Principles, Patterns and Process of Software Development*: Prentice Hall, 2001, in press.
[17] E. M. Maximilien and L. Williams, "Assessing Test-driven Development at IBM," presented at International Conference of Software Engineering, Portland, OR, 2003.
[18] M. M. Muller and O. Hagner, "Experiment about Test-first programming," presented at Empirical Assessment In Software Engineering EASE '02, Keele, April 2002.
[19] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT*, vol. 17, pp. 40-52, October 1992.
[20] W. W. Royce, "Managing the development of large software systems: concepts and techniques," presented at IEEE WESTCON, Los Angeles, CA, 1970.
[21] A. vanDeursen, "Program Comprehension Risks and Opportunities in Extreme Programming," CWI, Amsterdam SEN-R0110, ISSN 1386-369X, 2001.
[22] A. vanDeursen, L. Moonen, A. vandenBergh, and G. K. R. t. code, "Refactoring test code," presented at XP 2001, 2001.
[23] L. Williams, E. M. Maximilien, and M. Vouk, "Test-Driven Development as a Defect-Reduction Practice," presented at IEEE International Symposium on Software Reliability Engineering, Denver, CO, 2003.
[24] L. A. Williams, *The Collaborative Software Process*. Salt Lake City, UT: Department of Computer Science, 2000.