An Initial Investigation of Test Driven Development in Industry

Boby George
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
(+1) 919 601 2922
bobygeorge@ncsu.edu

Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
(+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. In our research, we ran a set of structured experiments with 24 professional pair programmers. One group developed code using TDD while the other a waterfall-like approach. Both groups developed a small Java program. We found that the TDD developers produced higher quality code, which passed 18% more functional black box test cases. However, TDD developer pairs took 16% more time for development. A moderate correlation between time spent and the resulting quality was established upon analysis. It is conjectured that the resulting high quality of code written using the TDD practice may be due to the granularity of TDD, which may encourage more frequent and tighter verification and validation. Lastly, the programmers which followed a waterfall-like process often did not write the required automated test cases after completing their code, which might be indicative of the tendency among practitioners toward inadequate testing. This observation supports that TDD has the potential of increasing the level of testing in the industry as testing as an integral part of code development.

Keywords

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

1. INTRODUCTION

Test Driven Development (TDD) [3], a software development practice used sporadically for decades [9] has gained added visibility recently as a practice of Extreme Programming (XP) [1, 2, 11, 12]. TDD is also known by names such as, Test First Design (TFD), Test First Programming (TFP) and Test Driven Design (TDD). The practice evolves the design 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 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 (OOP). 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." [4] TDD proponents argue that reduce coupling occurs because the practice guides towards 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 [3].

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

This 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 more traditional waterfall-like practice. External code quality will be assessed based on the number of functional (black-box test cases) test cases passed.
2. Programmers who practice TDD will develop code faster than developers who develop code with a more traditional waterfall-like practice. Programmers' speed will be measured by the time to complete (hours) a specified program.

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

2. BACKGROUND AND RELATED WORK

In this section, we first describe the TDD practice in detail. Then, we describe an empirical study of TDD that has been completed by researchers in Germany.

2.1 Test-Driven Development

With TDD, before writing implementation code, the developer writes automated unit test cases for the new functionality they are about to implement. After writing test cases that generally will not

SAC 2003, Melbourne, Florida USA.

even compile, the developers write implementation code to pass these test cases. The developer writes a few test cases, implements the code, writes a few test cases, implements the code, and so on. The work is kept within the developer's intellectual control because he or she is continuously making small design and implementation decisions and increasing functionality at a relatively consistent rate. A new functionality is not considered properly implemented unless these new unit test cases and every other unit test cases ever written for the code base run properly.

Intellectually, one can consider why TDD could be superior to other approaches.

- In any process, there exists a gap between decision (design developed) and feedback (performance obtained by implementing that design). The success of TDD can be attributed to the lowering, if not elimination, of that gap, as the granular test-then-code cycle gives constant feedback to the developer [3]. As a result, defects and cause of the defect can be easily identified – the defect must lie in the code that was just written or in code with which the recently added code interacts. An often-cited tenet of Software Engineering, in concert with the Cost of Change [5], is that the longer a defect remains in a software system the more difficult and costly it is to remove. With TDD, defects are identified very quickly and the source of the defect is more easily determined. Hence it is this higher granularity of TDD that differentiates the practice from other testing and development models.
- TDD entices programmers to write code that is automatically testable, such as having functions/methods returning a value which can be checked against expected results. Benefits of automated testing include: (1) production of a reliable system, (2) improvement of the quality of the test effort, and (3) reduction of the test effort and minimization of the schedule [8].
- The TDD test case create a through regression test bed. By continuously running these automated test cases, one can easily determine if a new change breaks anything in the existing system. This test bed should also allow smooth integration of new functionality into the code base.

2.2 Related Work

Recently, researchers have started to conduct studies on the effectiveness of the TDD practice. Muller and Hagner [14] conducted a structured experiment comparing TDD with traditional 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 automated acceptance testing for their analysis.

The TDD group wrote their test cases while the code was written, as described above; 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. However, the understandability of the program increases, measured in terms of proper reuse of existing interfaces.

These experimental results need be considered in the context of its limitations: the sample size was small, the students had limited experience with TDD, and the results were blurred to a degree due to large variance of data points. Additionally, the external validity of their results can be improved by running further studies with professional programmers.

3. RESEARCH APPROACH

We ran three TDD experimental trials [10] with professional programmers. These results add to those previously discussed.

3.1 Experiment Details

We ran experimental trials with eight-person groups of developers at three companies (John Deere, RoleModel Software, and Ericsson). In each of the experimental trials, the developers were randomly assigned to work in pairs in one of two groups: TDD and control. All developers used the pair-programming practice (whereby two programmers develop software side by side in one computer) [16]. Each pair was asked to develop a bowling game application (adapted from an XP episode [13]). The control group pairs used the conventional design-develop-test-debug (waterfall) [15] approach. All experiment participants were asked to develop a small program according to a set of requirements. Participants were asked to turn in their programs upon completing the activities as outlined. Then, their projects were assessed.

We expected that professional programmers would write code that handled all error conditions gracefully. However, after analyzing the results of our first trial, we found this not to be the case. We found that most of these initial pairs determined their implementation was complete when they could pass our specified acceptance test cases. Therefore, in the following two trials, all the developers were specifically asked to handle all error conditions gracefully and none of the pairs were provided acceptance test cases. Additionally, in the second two trials, the control group developers were asked to write automated test cases after development. Additionally, the experience level with TDD of the 24 developers varied from beginner to expert.

The effectiveness of TDD was analyzed using the following data: (1) the time taken by participants to develop the application to evaluate development speed; and (2) the results of black box functional testing to evaluate external quality. Additionally, the quality of the test cases written by TDD developers was measured using code coverage analysis. We supplemented our findings using survey data on the perceptions of the participants on the efficacy of TDD.

3.2 External Validity

An important consideration in empirical research design is external validity, 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 to the external validity of our experiment.

- Our sample size was relatively small (6 TDD pairs, 6 control group pairs).
- After reviewing the results of the first trial, we modified the experiment instructions for the trials that followed: (1) We emphasized the need for the control group developers to write automated test cases upon completing code implementation; (2) we emphasized that all developers need to handle error conditions; and (3) we did not provide any of the developers the acceptance test cases. Unfortunately, only one control group pair actually wrote any worthwhile automated test cases, despite the fact that they were specifically instructed to do so. Inadvertently, our control group may more accurately represent the "state of the practice" of software development in the industry
- In all the experiments, programmers worked in pairs. Two professional developer organizations used pair-programming practice in their day-to-day development and the other group was familiar with the practice. Hence, 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.
- Fourth, the application used in the evaluation process was very small (typical size of the code was 200 LOC).
- Fifth, the subjects of the experiments had varying experience with TDD (from novice to expert). The third set of professional developers had only three weeks of experience with TDD before the experiment. Hence, it is conceivable that the test-first approach on these subjects is not stabilized.

4. EXPERIMENT RESULTS

We now provide the results of our quantitative and qualitative findings of the experiment.

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. The results of these analyses are presented in this section.

4.1.1 External code quality

We developed 20 black-box test cases to evaluate the external code quality of professional developers' code. The test cases validated the degree to which requirement specifications were implemented and the robustness of the code (such as error handling capabilities). 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. First, the median value for the TDD developers' code is clearly much higher than of the control group developers' median.

A hypothesis of this research was that the TDD approach would yield code with superior external code quality. Based on the data analysis conducted, the experimental findings are supportive that the TDD approach yields code with superior external code quality. However, the validity of the results must be considered within the context of the limitations discussed in external validity section.

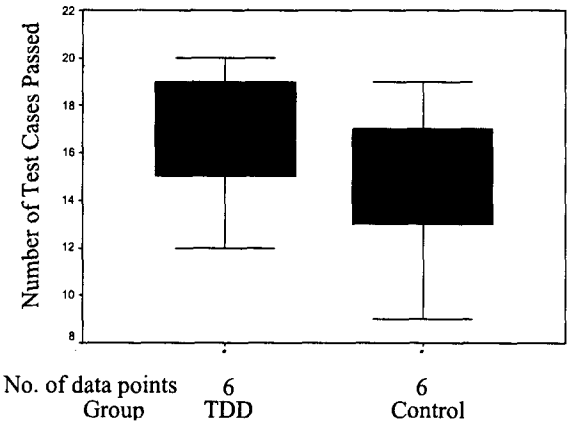


Figure 1: Box plot for Test Cases Passed

4.1.2 Productivity

People can be skeptical about the additional time needed to write and update test cases. As shown in Figure 2, on an 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 developers.

An important consideration in this analysis is that the control pairs were asked to write test cases after they developed code (in a traditional code-then-test fashion). 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. The extra time taken by TDD could be attributed to the time needed to develop test cases.

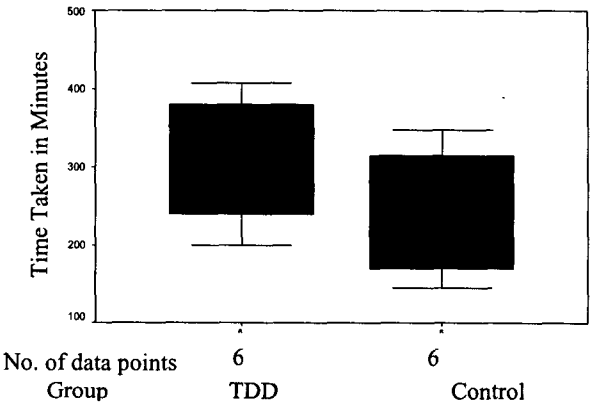


Figure 2: Box plot of Time Taken by Developers

There are many benefits resulting from the test cases created by the TDD developers. First, the TDD pairs produced test assets along with the implementation code. These test assets are very valuable in the product life as the product is enhanced. Second, the code developed is testable. If programs are written without continuous consideration towards being automatically testable, writing such test cases after the fact can be very difficult, if not impossible. Third, the code that enters subsequent testing phases and that is delivered to the customer is of higher quality. This higher quality reduces testing and field support costs. Finally, the overall life cycle time might be less in subsequent iterations as changes can be made more easily.

It was hypothesized that programmers who practice TDD will be more productive, as measured by the time to complete a program. However, contrary to hypothesis, the experiment results showed the TDD developers took approximately 16% more time than the control group developers. However, the validity of the results must be considered within the context of the limitations discussed in external validity section.

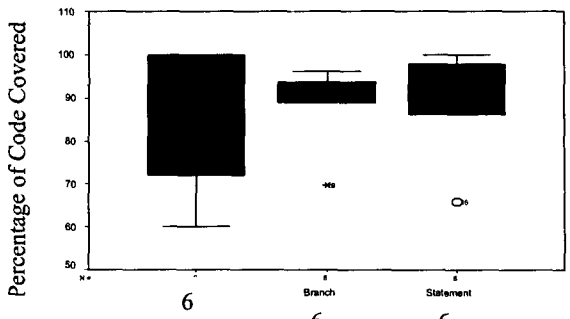
4.1.3 Correlating Productivity and Quality

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. However, one must consider that all pairs turned in their programs when they felt it would run correctly. The TDD pairs did not feel they were done until they wrote higher quality code with a good set of automated test cases. The control group pairs felt they were done with lower quality code, primarily without any worthwhile automated test cases.

4.1.4 Code coverage

One of the concerns about the TDD approach is the thoroughness of the test cases written by the TDD developers. Essentially in TDD, the quality of the tests determines the quality of the code. Analyzing the test cases for code coverage assessed the quality of the test cases written by TDD developers.

The industry standard for coverage is in the range 80% to 90%, although ideally the coverage should be 100% [7]. As shown in Figure 3, on average, the TDD developers surpassed the industry standards in all the three types of code coverage. The TDD developers' test cases achieved a mean of 98% method, 92% statement and 97% branch coverage. It must be noted that the testing tool used, JUnit, cannot test the main method (of Java code), and hence the main method was excluded from code coverage analysis.



No. of data points	6	6	6
Type of Coverage	Method	Statement	Branch

Figure 3: Box Plot of Code Coverage

4.2 Qualitative Analysis

4.2.1 Survey

It is fruitful to substantiate quantitative findings with qualitative feedback from the software developers in the experiment. A survey was conducted among the 24 professional developers who participated in the experiments. The survey, administrated before the experiment, consisted of nine close-ended questions. The nine close-ended questions were aimed at eliciting the developers' opinion on three concerns:

(1) How productive is the practice for programmers?

(2) How effective is the practice?

(3) How difficult is the approach 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 subscales or indexes (productivity, effectiveness, and difficulty of adoption) using the Cronbach's Coefficient Alpha test. The Cronbach's Coefficient Alpha measures this level of consistency of survey responses. This provides an indication of whether all of the questions within a subscale (for example, the productivity subscale) measure the same attribute and, therefore, individuals should answer all of the questions within the subscale similarly. The Alpha test results indicated that it was valid to aggregate the nine questions into the said three sections. The statistical significance of each response was then evaluated for each of these sections using Spearman's Rho test. All the survey responses were statistical significant at the 0.01 level ($p < 0.01$).

On questions on programmer productivity, an overwhelming majority of the developers believed that TDD approach facilitates better requirements understanding (87.5%) and reduces debugging effort (95.8%). However, only half of the developers felt that TDD led to less code development time. Taking the average of all positive comments, about 78% of developers thought that TDD improves overall productivity of the programmer.

For questions relating to effectiveness, 92% of developers believed that TDD yields higher quality code, 79% thought that TDD promotes simpler design and 71% thought the approach was noticeably effective. Hence, aggregating these scores indicates that 80% thought that TDD is effective.

The responses of developers on questions related to difficulties in adopting the approach indicate some concerns. Fifty-six percent (56%) of the professional developers thought that getting into the TDD mindset was difficult. A minority (23%) indicated that the lack of upfront design phase in TDD was a hindrance. Hence taking average of the responses, 40% of the developers thought that the approach faces difficulty in adoption.

Based on survey and student comments, it can be concluded that developers 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 approach 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 developers took more time (16%) than control group developers. 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 (though they were instructed to do so), making the comparison uneven.
- On an average, 80% of the professional developers held that TDD was an effective approach and 78% believed the approach improves programmers' productivity. The survey results are statistically significant.
- Qualitatively, this research also found that TDD approach facilitates simpler design and that lack of upfront design is not a hindrance. However, for some, transitioning to the TDD mindset is difficult.

These results need to be viewed within the limitations of the experiments conducted. 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 developers at John Deere, RoleModel, and Ericsson who participated in this research.

7. REFERENCES

[1] Auer, K. and Miller, R., *XP Applied*: Addison-Wesley, 2001.

[2] Beck, K., *Extreme Programming Explained: Embrace Change.*: Addison-Wesley, 2000.

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

[4] Beck, K., "Aim, Fire," in *IEEE Software*, vol. 18, September/October 2001, pp. 87-89.

[5] Boehm, B. W., *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[6] Chaplin, D., "Test First Programming," *TechZone*, 2001.

[7] Cornett, S., "Code Coverage Analysis," Bullseye Testing Technology 2002.

[8] Dustin, E., Rashka, J., and Paul, J., *Automated Software Testing*. Reading, Massachusetts: Addison Wesley, 1999.

[9] Gelperin, D. and Hetzel, W., " Software Quality Engineering," presented at Fourth International Conference on Software Testing, Washington D.C., June 1987.

[10] George, B., "Analysis and Quantification of Test Driven Development Approach," *North Carolina State MS Thesis*, 2002.

[11] Jeffries, R., Anderson, A., and Hendrickson, C., *Extreme Programming Installed*: Addison Wesley, 2001.

[12] Jeffries, R. E., "Extreme Testing," presented at Software Testing and Quality Engineering, 1999.

[13] Martin, C. R., *Advanced Principles, Patterns and Process of Software Development*: Prentice Hall, 2001, in press.

[14] Muller, M. M. and Hagner, O., "Experiment about Test-first programming," presented at Empirical Assessment In Software Engineering EASE '02, Keele, April 2002.

[15] Royce, W. W., "Managing the development of large software systems: concepts and techniques," presented at IEEE WESTCON, Los Angeles, CA, 1970.

[16] Williams, L. A., *The Collaborative Software Process*. Salt Lake City, UT: Department of Computer Science, 2000.