Impact of Using Test-Driven Development: A Case Study

Sumanth Yenduri
Dept. of Computer Science
Univ. of Southern Mississippi
Long Beach, MS 39560
Sumanth.Yenduri@usm.edu

L.A. Perkins
Dept. of Computer Science
Univ. of Southern Mississippi
Long Beach, MS 39560
louise.perkins@usm.edu

Abstract

In this paper, we conduct an experimental study over two groups of students comprising of undergraduate students (seniors) who develop software using the conventional way of performing unit testing after development and also by extracting test cases before implementation as in Agile Programming. Both groups developed the same software using an incremental and iterative approach. The results showed that the software had less number of faults when developed using Agile Programming. Also, the quality of software was better and the productivity increased.

Keywords: Test driven development, agile programming, case study

1. Introduction

Test-Driven Development (TDD) is a technique that involves writing test cases first and then implementing the code necessary in order to pass the tests. The goal is to achieve immediate input and thereby construct a program. This technique is heavily emphasized in Agile or Extreme Programming [1, 2, 3]. This process of designing test-cases prior to the implementation is termed as "Test-First" approach. We consider unit-testing only (by the programmers) and have nothing to do with integration or acceptance testing. But we do take into account the number of faults found while SQA performs formal unit-testing, integrating and performing acceptance testing in order to measure the quality of the software produced.

The first step in this approach is to quickly add a test, basically just enough code to fail. Then we run our tests, generally all of them but in order to finish the process quickly, may be only subsets of tests are run to make sure that the new test does fail. We then update the code in order to pass the new tests. Now, we again run our tests. If they fail, we again have to update and retest, else we add the next functionality. There are no particular rules to form the test-cases but more tests are added throughout the implementation. Though, refactoring should be performed in agile programming that is programmers alternate between adding new tests and functionalities to improve its consistency. It is done to improve the readability of code or change of design or removal of unwanted code.

There are various advantages in employing TDD. Programmers tend to know immediately whether the new feature has been added in accordance with the specifications. The process in performed in steps comprising of small parts and hence easier to manage. Low number of faults are tend to be found during acceptance testing and maintenance can be viewed as another increment or addition of feature which would make it easier. There is no particular design phase and software is built through the process of refactoring. In short, TDD improves programmer productivity and software quality. There have been a number of studies [4, 5, 6, 7] that have been performed to test the effectiveness of TDD and the results give mixed opinions. We perform an experiment with 2 groups of students, one developing software the conventional way of testing it after implementation and the other group through TDD. In both groups, test cases were developed by programmers and regression testing was performed. Only difference is test cases are written prior to implementation and are tested throughout the production in TDD and test cases are written and tested after implementation in the conventional way. Each group consisted of 9 undergraduate students and time period for the whole study was 3 months. We investigate in this paper through experimental studies the promise of "Test-First" strategy emphasized in agile programming.

2. Methodology

The following figures demonstrate the approaches employed by both the teams. The methodology for TDD is depicted by figure 1 and the conventional method by figure 2. In TDD, a test is added and tested with the code produced. Code is reworked until all the tests are passed. Additional features are added one by one in an incremental process. Additional tests are added to the test bed as and when needed. In the traditional way all the implementation is completed and then test cases are designed. The code is run through all the tests and is reworked until desired functionality is achieved.

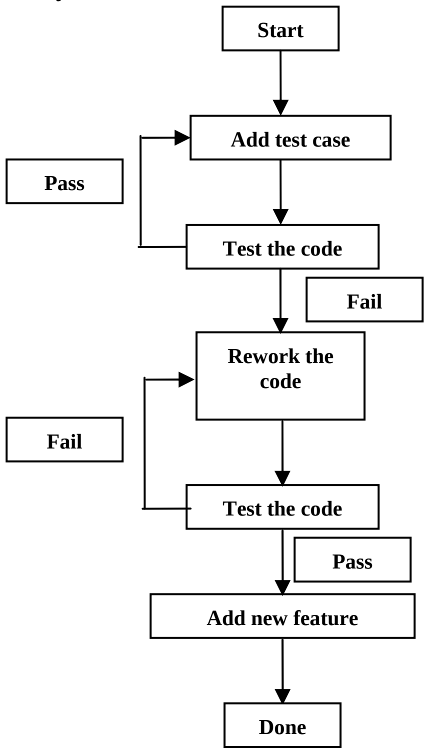

Figure 1. Test Driven Development

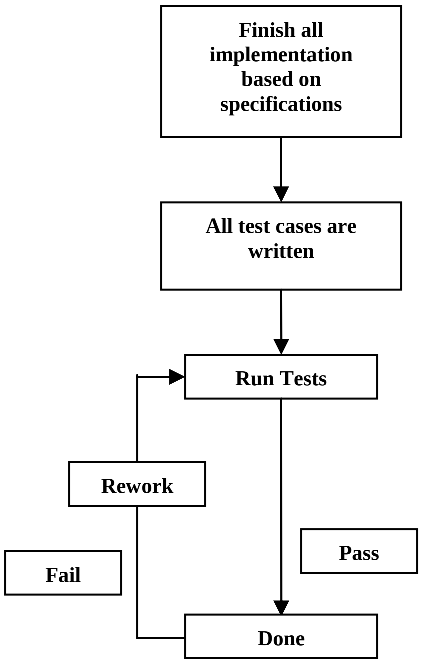
Finish all implementation based on specifications
All test cases are written
Run Tests
Rework
Pass
Fail
Done

Figure 2. Traditional Development

3. The Case Study

Our primary objective is to compare and evaluate the impacts of Test-Driven Development with Traditional Development taking into account factors related to software quality assurance, productivity factors etc through an incremental process. Numerous references [1, 2] quote that in TDD the amount of test code written is comparable to the implementation code which is not the case in traditional approaches. A simple logic is that more test cases obviously would enhance software quality and hence software quality of TDD should be better than that of the traditional approach of testing. Fault free code ensures less time in maintaining the software which is a very important thing as maintenance accounts to almost 70% of total costs during the life time of software [8]. We thus measure the number of test cases written by both groups and determine if quality improved and thus evaluate if TDD is a better approach for testing software.

The students forming each group were part of the software engineering course being taught at an academic setting with each group having 9 students. All of them were senior students and had already completed the required programming courses that were needed for the study. They were provided a lab with 25 terminals that had all the required software ranging from internet web browsers to the computer aided software engineering tools to accomplish the task. Configuration tools too were available to keep track of the different versions and variations. All of them were given required user manuals and a short description for forming testing cases (using the testing tools) and all the necessary specifications and tasks to be implemented were briefed. They were trained in applying both the approaches too. During the lab sessions, students were asked to do their job. Separate accounts were provided to each of them. Students could work at home also and were not restricted to lab sessions alone. The goals of the study were known to the students before the experiment was conducted. Participation though was voluntary. Their skill set was assessed based on a test given to them after the end of all the training sessions and the grades they made in the programming courses in the earlier semesters. Java was the programming language used. All the students were undergraduates majoring in computer science. Everyone was a senior with a background of at least 3 high level languages which was verified by examining their transcripts. 2 students though proved their capabilities through work experience. There were no drop outs.

We measured the following metrics for both the groups. The number of test cases written, the number of faults found later by the quality assurance group that performed unit-testing after the units were handed over by the programmers. Integration testing was performed and the number of faults was recorded. The number of faults during acceptance testing also noted. Finally, the total person hours for each group taken in order to accomplish the tasks were noted. The following table tells us the facts:

Table 1. Metrics Recorded after the experiment

	TTD	Traditional Approach
Number of test cases written	629	211
Number of faults detected by the SQA group during all the units tested	74	109
Number of faults detected by the SQA group during integration testing	13	15
Number of faults detected by the SQA group during acceptance testing	14	31
Total Person Hours spent	928	1245

4. Discussion of Results

The results clearly reveal the fact that the total number of faults in unit, integration and acceptance testing was lower in TDD when compared to the traditional approach. The number of faults during unit testing was considerably low in TDD though the number of faults during integration testing was almost similar. But the real catch comes during the acceptance testing. The number of faults detected by the SQA group in TDD was less than half that of the traditional approach. Most faults were with respect to the functionalities noted in the specification document. The overall time taken by TDD was 317 hours less than the traditional approach. Caution was taken while noting this metric. All users who worked from home had to logon to an online time-noting system which also monitored the whole sessions including the keystrokes etc. Any person hour that had less than 15 key strokes was discarded. The productivity and quality of the software developed through TDD was clearly better than the traditional approach. However, the software developed can be termed as a small-sized project and obviously easier to manage.

The decrease in the number of faults during the following testing phases in the TTD may be attributed to a number of factors such as the immediate feedback provided by the test cases and the high number of tests performed by the programmers in that group. As Edsgar Dijkstra said *"The programmer should let the program proof and program grow hand in hand"* which means for example when a loop is incorporated in the code, a corresponding loop invariant test case is generated and the code is checked as the design is refined or refactored. He also said *"The only effective way to raise the confidence level of a program significantly is to give a convincing proof of correctness"*. In the same work Dijkstra quotes that *"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence"* [9]. The only counter attack would be to design as many number of test cases as possible while at the same time designing them with a view-point to demonstrate program correctness. To an extent, this is the general philosophy of agile programming which emphasizes on building test cases in sync with program development. This is the reason we identified for the quality in the software developed using TTD.

There was a better productivity in TTD when compared to the traditional approach as the functionalities were implemented unambiguously by the TTD group members as it was a small sized project and the features could be easily implemented in an incremental approach. There was much less number of person hours spent on rework when compared to the traditional approach. Also the rework was easy as only a subset of tests were considered most of the time. Furthermore, focus of the programmer was better as the number of test cases were low at each step adding to the productivity. This is a direct consequence of Millers Law which says that human brain cannot concentrate on more than seven chunks of information at one time [10]. These were the reasons that could be attributed to the better productivity levels achieved in TTD in comparison with the traditional approach.

5. Conclusions

We noted the significance of test-driven approach emphasized by agile programming in order to develop software quality and productivity and lower the costs in maintaining it at a later time. Through our experiment we showed the TTD approach yielded better results with respect to both quality and productivity. This is because of the way test cases were designed as the software was being developed. Focus on smaller subsets of information and a large number of test cases designed along with program development lessened the rework time and promoted quality. Though, our results encourage the employment of TTD approach, one should consider the size of the software project implemented. We can with confidence say that TTD approach may be a good choice in comparison with the traditional approach for small sized software projects. The usefulness of the approach needs to be validated by using it for larger projects involving larger group numbers.

6. References

[1] K. Beck, Test-Driven Development: By Example. Addison Wesley, 2003.

[2] D. Astels, Test Driven Development: A Practical Guide. Prentice Hall, 2003.

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

[4] Hakan Erdogmus, Maurizio Morisio, Marco Torchiano, “On the Effectiveness of the Test-First Approach to Programming”, Ieee Transactions On Software Engineering, Vol. 31, No. 3, March 2005.

[5] M.M. Muller and W.F. Tichy, Case Study: Extreme Programming in a University Environment, Proc. Intl Conf. Software Eng. (ICSE), 2001.

[6] M.M. Muller and O. Hagner, Experiment about Test-First Programming, Proc. Conf. Empirical Assessment in Software Eng. (EASE), 2002.

[7] S.H. Edwards, Using Test-Driven Development in the Classroom:Providing Students with Concrete Feedback on Performance, Proc. Intl Conf. Education and Information Systems: Technologies and Applications (EISTA 03), Aug. 2003

[8] Stephen R. Schach, Object-Oriented and Classical Software Engineering, Sixth Edition, McGraw Hill, 2005.

[9] E.W. Dijkstra, “The Humble Programmer”, Communications of the ACM 15, pp 859-866, 1972.

[10] G.A. Miller, “The magical number seven, plus or minus two: some limits on our capacity of processing information”, The Psychological Review 63, pp81-97, 1956.