THE IMPACT OF TEST-DRIVEN DEVELOPMENT ON SOFTWARE DEFECTS AND COST: A COMPARATIVE CASE STUDY

Dadi Mohankrishna Rayudu\textsuperscript{1}, Naresh. E\textsuperscript{2}, Dr. Vijaya Kumar B. P\textsuperscript{3}

\textsuperscript{1}(Dept. of ISE, M S Ramaiah Institute of Technology, MSR Nagar, Bangalore, INDIA)
\textsuperscript{2}(Dept. of ISE, M S Ramaiah Institute of Technology, MSR Nagar, Bangalore, INDIA, Research Scholar, Jain University, Bangalore)
\textsuperscript{3}(Dept. of ISE, M S Ramaiah Institute of Technology, MSR Nagar Bangalore, INDIA)

ABSTRACT

Present business world competition between the companies is increasing a lot with respect to quality, services and trust worthiness. Test driven development approach is one of the key factors for quality delivery of products. Test driven approach is one of amongst best practices in software from past decades. One of the most time consuming and costly processes of the Software Development Life Cycle has been the maintenance of production code. As of 2000, the overall maintenance costs have been more than 90%. To overcome this challenge many tools, frameworks and software paradigms have risen. On the forefront of this effort is writing unit test cases by the developers under the banner of Test Driven Development. This practice tackles the problem in the development phase. This work discus on TDD an issue that tackles to reduced cost and defect.

Keywords: Software Paradigm, TDD, Unit Test, Cost Estimation and Reduction, Quality.

1. INTRODUCTION

Test Driven development or TDD is related Extreme programming which gained prominence in 1999. Kent Beck is supposed to have rediscovered this method in 2003 and has gained prominence as a software paradigm [1][2][3]. This methodology of software development does not compartmentalize the development process as in traditional models of Software Development Life Cycle (SDLC). However it views unit testing, development and re factoring of code as a single short incremental atomic unit.
1.1 Definition

It is a software development process where the developer writes a test case for a new requirement, runs the test case which will fail, goes on to write the code to satisfy the test case, runs the test suite again and finally re factors. All these processes are closely tied to each other and happen as a single unit [4][5].

According to Beck, (Beck, 2002), it should follow the below two principles:

1. Never write a single line of code unless you have a failing automated test.
2. Eliminate duplication [1][2][3].

Section 2 provides background on the TDD unit testing techniques. Section 3 provides Libraries used for testing. Section 4 presents TDD Experiment design. Section 5 presents the results of our analysis. Section 6 Provides feature work and finally Section 7 conclusion.

2. TEST-DRIVEN DEVELOPMENT

The following diagram illustrates the process in its individual steps [1][6]

![TDD Process Diagram]

**Fig 1: TDD Process Diagram**

Add Test: In this step the developer adds a new test case into the test suite after understanding the requirements then new test will run. The new test case will inevitably fail because there is no corresponding code to back it up. An important feature of this step is that the developer is forced to
comprehend the requirement specifications before actually writing the production code. Writing unit tests prior to development, rather than after, means the developer has to think of all the ramifications and branches the code can evolve into[6].

Compile Test: After adding the new test, all the tests in the test suite are run. One can optionally run only the newly created test case. Most often all tests are run to verify that all the existing features are behaving as expected and acts as an acknowledgment to the developer that this new test case is the new feature going to be added.

Write Code: Once the developer confirms that the newly added test fails, code is written just enough to pass the test case. This ensures that the code is prevented from bloating or writing code for a feature that already exists.

Re-factor: This step is executed once the developer is confident all the tests have passed. This step is necessary because it eliminates duplication. Once the re factoring is done, the test suite is run again so that the re factoring has not broken any existing code. These steps are repeated yet again when requirement for a new feature arrives.

2.1 Advantages of Test Driven Development

2.1.1 Detects Design Flaws Earlier

One of the foremost advantages of TDD is that it provides a check on misunderstandings or communication gap when actually implementing the requirements. The developer is forced think about all the branches that has to be tested upfront. This may lead to further discussions and design debates which act as a safety net for any flaws trickling down. According to (Hawley, 2004), 87.5% of developers reported better requirements understanding and 79% of developers believed TDD promoted simpler design[4][6].

2.1.2 Debugging Time Is Reduced

Since most of the pitfalls of design are caught in the earlier stages, the developer spends lesser time in using the debugger after implementation. According to (Hawley, 2004) 95.8% of developers reported reduced debugging efforts [4][6]. This greatly reduces cost during the maintenance phase where most of the bugs are found.

2.1.3 Provides a Good Metric for Completion

Many times managers would want to know how much of the development is actually complete. Before Test Driven Development, the developer would give a qualitative or quantitative answer that was hard to substantiate. However by using this technique, the developer can show the total number of test cases, percentage of test cases failing or passing. This greatly increases the productivity of the developer. According to (Hawley, 2004) 78% of developers reported TDD improved overall productivity.

2.1.4 Implicit Documentation

In reality, due to the fast pace of product roll out time, the developer rarely documents the code being written. However with unit tests being written upfront, there will be a ready example on the behavior of a feature – the input, the usage and the expected output.
2.1.5 Provides Great Code Coverage
Before adding an *else* condition to existing code the developer has to write a test case that covers the else case too. This helps in tracking what changes have been made and what has to be covered in the future.

2.2 Drawbacks of TDD

2.2.1 Acceptance by Developers
Asking developers to write unit tests first is counter intuitive. A proper process of inducting these principles into the process needs to be in place.

2.2.2 Boxed Developer Vision
Since the developer who writes the tests is the same one who is going to write the code, the blind spots might seep into the code as well.

2.2.3 Legacy Code
Inducting test driven development into existing legacy code can become an arduous task. Very little or no documentation and non-existent unit tests make it hard to employ TDD.

2.2.4 Maintaining Test Cases
Over time new features have to be added into the existing system. However due to a small change in the code, many test cases may fail and need refactoring. This is one of the major overheads when employing TDD.

3. LIBRARIES USED FOR UNIT TESTING

As part of unit testing for the existing java code base, JUnit was used for creating the test cases and JMockit was used for mocking classes and behavior. The sections below will give a brief overview of these two libraries used.

3.1 Junit
JUnit is a java framework for writing unit test cases in Java developed by Kent Beck and Erich Gamma on flight from Zurich to Atlanta in 1997[7][8]. It is actually a part of the xUint family from which other testing frameworks have been derived.

JUnit gained immense popularity and boosted the growth of Test Driven Development and Extreme Programming which depend on a short feedback loop in the software process. Other frameworks followed after JUnit were CPPUnit for C++ and NUnit for .Net.

3.2 Junit Features
Test fixtures help in setting up base conditions for running your steps namely resource initialization and reclamation. JUnit provides the following annotations for setting up test fixtures.
@Before – A method annotated by this will execute before every test method
@After – A method annotated by this indicates the method should execute after every test method
@BeforeClass – The method annotated by this annotation should be public and static. This method will be executed before all the test methods.
@AfterClass – The method annotated by this annotation should be static. This method will be executed after all the test methods [7][8].
3.2.1 Test Method
This is the actual test case that executes the code to be tested. It forms the heart of the test case. JUnit provides the following annotation for marking a method as a test method.

@Test – The method to be tested is marked with this annotation. The method should be public void [5].

3.2.2 Assertions
These are static methods of the Assert class that help us check the expected versus actual outputs. Below are a few of the methods that have been used in implementing the test suite.

assertEquals(message, expected, actual) - Asserts the equality of expected and actual objects.
assertNull(object) - Asserts that the object is null.
assertTrue(condition) - asserts that the condition is true.
assertSame(message, expected, actual) - Asserts if the expected and the actual are the same object. This is equivalent to object comparison using the == operator.

3.2.3 Test Suite
JUnit provides annotations to combine multiple test classes into a single test bundle - called as a Test Suite. This is a great utility for grouping related tests into a single class. The following two annotations must be used to mark a class as a test suite [9].

@RunWith(value = Suite.class)
@SuiteClasses(value = {TestClassA.class, TestClassB.class})

3.2.4 JMockit
Any unit testing invariably needs to be used with a mocking framework that supports unit testing. The Test Suite has used JMockit library for mocking. Following are some of the features of JMockit that have been used in creating the test suite [9].

Expectations class: These classes specify the expected/allowed invocations on mocked objects or classes. They are used for setting up the preconditions before the code being tested is invoked as anonymous inner classes or extending a class with Expectations.

Inline Mockups: These are template based classes that allow us to override a class’s methods by passing them as the template’s type. They are defined inside individual test methods as anonymous inner classes.

Deencapsulation class: This class provides a set of static methods that can be used to invoke private methods and access private variables.

4. TDD EXPERIMENT DESIGN
Doing Experimentation in software engineering is very difficult because it needs so many resources. In our experiment we are to find so many interesting facts about software characteristics.

4.1 Goals
The Goal is to decrease cost and defects in software so by that we can increase the quality of final software using test-Driven Development technique.
4.2 Task and Tools Used

We experimented with two different kinds of approaches one is developing software with normal way of approaches and second one is using test-Driven development [9] [10].

To do this work the tools are used in experiment are:
1. Eclipse-IDE for developing environment
2. Junit for writing for unit test cases

4.3 Task Description

In our context we are trying to develop software which it will store critical information of system and try to retrieve back when it is needed to restore system. So we divided into two groups of team and try to develop using same programming language (JAVA).

One team followed agile methodology to develop and other team followed Test-Driven development approach. The first team is continue the already existing software (Legacy code) other team is going to start from scrap using test-Driven Development approach. The details of is shown in below Table 1.

<table>
<thead>
<tr>
<th></th>
<th>Legacy code(Ad-hoc approach)</th>
<th>New code(TDD approach)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Team Size(Developers)</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Team Experience(Language and Domain)</td>
<td>Experienced</td>
<td>Some Inexperienced</td>
</tr>
<tr>
<td>Code Size(KLOC)</td>
<td>14.2</td>
<td>20.3</td>
</tr>
<tr>
<td>Language</td>
<td>JAVA</td>
<td>JAVA</td>
</tr>
<tr>
<td>Unit Testing</td>
<td>Ad-hoc</td>
<td>TDD</td>
</tr>
</tbody>
</table>

Table 1. Task developing environment

4.4 Unit testing practices

The approach of legacy code team is Ad-hoc and new code team is following TDD approach. In our approach unit test cases need to cover 80% of main classes so according to that we developed unit test cases in different real time use case scenarios[11][12].

We will design unit test case like how the execution of some classes in real time using some tools like Junit. So first we should get know about the behavior of the class and some domain knowledge about the software how it works then only we can write perfect unit test cases [13].

So many people while writing test cases they will see only in coverage based type of testing but it will not give the correct results so developers should think in the different use case based scenarios where users get more problems[14][15].

The main theme in Unit testing is creating simulation environment like as software works.

5. RESULTS

According to our experiment we found some good results in numbers. After developing software using both approaches compare to legacy code team with Ad-hoc approach the team with TDD approach has less number of defects. If the number of defects are more than the cost also will be more so TDD approach is proven it was cost effective also the below Table2 has no of defects in each phase.
As we can see in Table 2 the most defects are appearing in testing phase in Ad-hoc approach because while coding was done the most use cases will ignore by developers.

![Fig.1 Total no of defects per KLOC comparison](image)

### Table 2. No of defects in each phase in two approaches

<table>
<thead>
<tr>
<th>Phases</th>
<th>No of defects (Ad-hoc) per KLOC</th>
<th>No of defects(TDD) per KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Design</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>Coding</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>Testing</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>Maintenance</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>50</td>
<td>10</td>
</tr>
</tbody>
</table>

As we can see in graph the normal way of approach produces 50 defects per KLOC so it is not good to software product these many defects are not a problem in normal systems but when you are trying to develop any critical system these defects are leads to huge damage also.

If you compare in New TDD approach comparatively the number of defects are less and we found out these defects also may solve in short time.

We also observed in our software how these defects are increased cost of development and maintenance also. According our project we represented how much cost it will take for each defect in different phases of development. So Using TDD development we found that no of defects are very less in Maintenance phase compare to Ad-hoc approach so the cost of maintainer will be very less.

According to defect and cost relation the no defects were increasing the cost of the product also increase. In requirement phase the cost of defect is very less compare to next phases so if we found defects itself in requirement phase itself very good but it is difficult. In normal approaches every one do test after design phase but in our TDD approach is parallel to coding unit testing will also done so there we can decrease lot of defects so cost of software production also will decreases.
In Table 3 we can see the cost of defect and how these defects cost in each phase

<table>
<thead>
<tr>
<th>Phases</th>
<th>Cost per defect</th>
<th>Cost of defects in each phase(Ad-hoc) per KLOC</th>
<th>Cost of defects in each phase(TDD) per KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>$130</td>
<td>5*$130 = $650</td>
<td>2*$130 = $260</td>
</tr>
<tr>
<td>Design</td>
<td>$450</td>
<td>10*$450 = $4500</td>
<td>3*$450 = $1350</td>
</tr>
<tr>
<td>Coding</td>
<td>$970</td>
<td>10*$970 = $9700</td>
<td>2*$970 = $1940</td>
</tr>
<tr>
<td>Testing</td>
<td>$7000</td>
<td>20*$7000 = $140000</td>
<td>2*$7000 = $14000</td>
</tr>
<tr>
<td>Maintenance</td>
<td>$14000</td>
<td>5*$14000 = $70000</td>
<td>1*$14000 = $14000</td>
</tr>
<tr>
<td>Total</td>
<td>$224850</td>
<td></td>
<td>$31550</td>
</tr>
</tbody>
</table>

Table 3. Total cost due to defects with different approaches

We can observe there is huge cost deference. So TDD approach can be possible for developing new software products with less cost than. We can see in Fig 2 testing phases cost difference is so more because the previous stages so many issues and use cases missed in normal way of developing but in TDD there is more possibilities to find bugs and some of the bugs will not able to find using black-box testing so those bugs will effect in operation of the software.

But still we still some of the bugs in TDD development but for those bugs we give advance solution to the customer like diagnose problem in operational of the software. So the cost will decrease but we should spend some time and cost at the initial stages of development because developer should have good knowledge on TDD other again we have to give training to them. Here some extra money we should spent but still we will get more benefit like in this stage if we spent $100 later stages we can save $1000 and customer will believe in our company also so TDD has advantage than normal way of developing.

The cost reduction in TDD approach not only due to defects the effort require by the developers are also less if we allocate less resources to the project it will be cost effective Because in normal approaches in initial stages the resources need somewhat less compared to TDD but later on stages like testing and maintenance phases the resources allocated is more in normal development approach.
<table>
<thead>
<tr>
<th>phases</th>
<th>Effort estimation (Ad-hoc)persons/month</th>
<th>Effort estimation(TDD)persons/month</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Design</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Coding</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Testing</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Maintenance</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>41</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 4. Effort estimation in all phases

So we can see Table 4 the total effort estimation is less in TDD approach because the all defects and bugs are fixed in early stages only so if we compare cost now. For each resource if we spend $2000 per month then the total cost will increases so much in Ad-hoc approaches. So the total cost for effort we can see below Table 5

<table>
<thead>
<tr>
<th>Approach</th>
<th>No of resources</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ad-hoc</td>
<td>41</td>
<td>41*$2000=$82000</td>
</tr>
<tr>
<td>TDD</td>
<td>22</td>
<td>22*$2000=$44000</td>
</tr>
</tbody>
</table>

Table 5. Total cost for effort

![Effort estimation comparison](image)

Fig 3. Effort Estimation Comparison in Approaches

6. FUTURE WORK

In Test-Driven development there is need lot effort to do automation so we are thinking to develop such kind of framework to do all process automated so developer can give his own inputs according to that, the unit test framework should develop and it should only take care of running all test cases.
7. CONCLUSION

As TDD is not oldest technique but it is inexperienced by so many people and organizations so the researchers across the world who are working in companies has to look once empirical database how the techniques are followed by TDD. Very few empirical studies available on this topic. We try to prove with our experiment how TDD is more effective than normal approaches respect of defects and cost. In present world, customers’ needs effective, efficient and quality software and that will possible with TDD approach.

REFERENCES