Comparative Study of Test Driven Development with Traditional Techniques

Shaweta Kumar, Sanjeev Bansal

Abstract - Test-Driven Development is the evolutionary approach in which unit test cases are incrementally written prior to code implementation. In our research, we will be doing comparative study of Test Driven development with traditional techniques through literature study as well as industrial survey. Through this research, we would like to find out the factors encouraging the use of Test Driven Development and also the obstacles that are limiting the adoption of Test Driven Development in the industry. The TDD method is radically different from the traditional way to create software. In traditional software development models, the tests are written after the code is implemented, in other words we could refer it as test-last. This does not drive the design of the code to be testable. Defining the tests with the requirements, rather than after, and using those tests to drive the development effort, gives us much more clearly picture and share focus on the goal. If tests are written after the implementation, there is a risk that tests are written to satisfy the implementation, not the requirements. 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."

Keywords- extreme programming, refactoring, test driven development, test-first methodology, test-last methodology.

I. INTRODUCTION

Test-driven development (TDD) is the core part of the agile code development approach drives from Extreme Programming (XP) and the principles of the Agile Manifesto. According to literature, TDD is not all that new; an early reference to the use of TDD is the NASA Project Mercury in the 1960's. Several positive effects have been reported to be achievable with TDD.

TDD is not a testing technique, as its name indicates, but rather a development and design technique in which the tests are written prior to the production code. The tests are added gradually during the implementation and when the test is passed, the code is refactored to improve the internal structure of the code. This cycle is repeated until whole functionality is implemented. The TDD cycle consists of six fundamental steps:

1. Write a test for a piece of functionality,
2. Run all tests to see the new test should fail,
3. Write code that passes the tests,
4. Run the test to verify they pass,
5. Refactor the code and
6. Run all tests to see the refactoring did not change the external behavior. [1]

The first step involves simply writing a piece of code that tests the desired functionality. The second one is required to validate that the test is correct, i.e. the test must not pass at this point, because functionality has not been implemented yet. Nonetheless, if the test passes, the test is not correct and requires update. The third step is the writing of the code. However, it should be kept in mind to only write as little code as possible to pass the test. Next, all tests must be run in order to verify desired functionality is implemented. Once all tests pass, the internal structure of the code should be improved by refactoring.

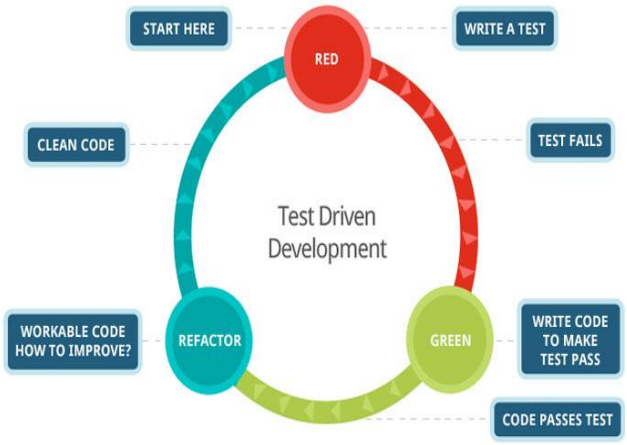

Figure 1: Test Driven development

A. Write test:

Tests in TDD are somewhat like unit tests with the difference that they are written for behaviors, not for methods. It is important that tests are written so that they are order independent, i.e. the result remains the same regardless of the sequence in which the tests are run. [3]

When writing the tests it should be kept in mind that the tests should concentrate on testing the true behaviors, i.e. if a system has to handle multiple inputs, the tests should reflect multiple inputs. [3]

B. Run test:

The test is run to verify if test written is useful or not. In case, test case passes, this indicates that test is worthless. This also validates that the test harness is working correctly and that the new test does not mistakenly pass without requiring any new code. The new test should also fail for the expected reason.

Manuscript Received on March, 2013

Shaweta Kumar working as System Analyst at Aon Hewitt, is currently pursuing M.Tech Computer Science, ASET, Amity University, Noida, India.

Prof. Dr Sanjeev Bansal, Guide, Amity Business School, Amity University, Noida, India.

This increases confidence (although it does not entirely guarantee) that it is testing the right thing, and will pass only in intended cases.

C. *Write Code:*

In TDD, the code writing is actually a process for making the test work, i.e. writing the code that passes the test. Beck proposes three different approaches for doing that: fake it, triangulation, and obvious implementation. The first two are techniques that are less used in concrete development work.

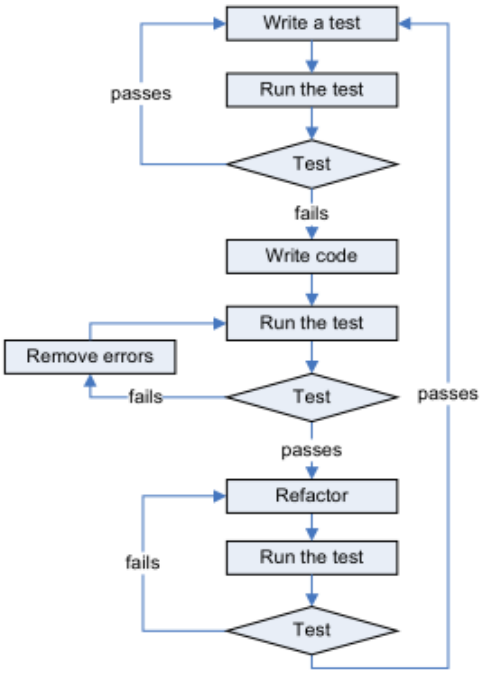

Figure 2: TDD Flowchart

"Fake it" may be employed, for example, by replacing the return value of some method with a constant. It provides a quick way to make the test pass. It has a psychological effect while giving the programmer confidence to proceed with refactoring and it also takes care of the scope control by starting from one concrete example and then generalizing from there. The abstract implementation is driven through sensing the duplication between the test and the code. "Fake it" implementation can give a push towards the right solution, if the programmer really does not know where to begin to write the code. [3]

Triangulation technique can be used to get to the correct abstraction of the behavior, i.e. the "fake it" solution is not usable anymore. For example, there are at least two different cases in the test method requiring different return values, and obviously, returning of the constant does not satisfy both of them. After reaching the abstract implementation, the other assertion becomes redundant with the first one and it should be eliminated. [3]

The obvious implementation is used when the programmer is confident and knows for sure how to implement some operation. Constant practicing of "obvious implementation" can be exhaustive, since it requires constant perfection. When the tests start to fail consecutively, it is recommended to practice the "fake it" or the "triangulation" until confidence returns. [3]

D. *Refactor:*

Refactoring is a process of improving the internal structure by editing the existing working code, without changing its external behavior. It is essential, because the design integrity of software scatters over time due to the accumulated pressure of modifications, enhancements and bug fixes. Now the code can be cleaned up as necessary.

The idea of refactoring is to carry out the modifications as a series of small steps without introducing new defects into to the system. By re-running the test cases, the developer can be confident that code refactoring is not damaging any existing functionality.

In our research, we would be doing comparative study of Test Driven development with traditional techniques and discussing pros and cons of these techniques. This study will reveal factors that encourage the use of Test Driven Development in industry and would try to find the weaknesses of TDD that limits their use in industry.

II. BACKGROUND DETAILS

TDD has been studied in a number of prior experiments. Boby George and Laurie [7] conducted a structured experiment with 24 professional pair programmers to evaluate the External code quality, Productivity, and Code coverage using the TDD and the classical model. One group developed code using TDD while the other a waterfall-like approach. Study involved both quantitative as well as qualitative approach to examine the effects of TDD on external quality and programmer productivity.

Adnan, Daniel and Sasikumar [8] conducted survey to find out the roadblocks in adoption of TDD approach. In this study, they found seven limiting factors viz., increased development time, insufficient TDD experience/ knowledge, lack of upfront design, domain and tool specific issues, lack of developer skill in writing test cases, insufficient adherence to TDD protocol, and legacy code.

A study similar to the one presented in this research tried to evaluate the programmers' productivity, internal and external quality of the product, and the programmers' perception of the methodology [9]. In the study, few undergraduate students used a TDD methodology as opposed to a traditional development process and found using TDD more productive in comparison to traditional approach.

Janzen [10] demonstrated that programmers using TDD in industry produced code that passed in up to 50% more external tests than code produced by control groups not using TDD and spent less time in debugging. Janzen also reported that computational complexity is much lower in test-first code while test volume and coverage are higher.

In this research, we have conducted a survey that was distributed through Survey Monkey to 80 anonymous participants who have came from a range of organizational and team structures; from large multinational companies to small start-ups. Attempts were made to gather different perspectives of Test driven development as represented by different team roles.

III. COMPARATIVE STUDY OF TEST DRIVEN DEVELOPMENT WITH TRADITIONAL TECHNIQUES

According to Test Driven Development, TDD can be described as to "only ever write code to fix a failing test". Before any production code is ever written, the programmer must first write a test that will define the new functionality being coded. This is referred as Test-First. However, there is difference in test first and test driven development.

Test Driven Development = Test First + Refactoring

In Traditional techniques, first code is written, and then code is written and executed. So it is also referred as Test-Last.

The Test driven development and Traditional Techniques approach can be summarized in Figure 3. These figures only describe the detailed design, code, and unit test phases of the software development lifecycle.

According to Koskela, "The final step of the Test-Driven Development cycle of test-code-refactor is when we take a step back, look at our design, and figure out ways of making it better." [5] Although none of the steps in the Traditional technique sequence contain the word "refactor", this does not imply that this activity is omitted. Refactoring occasionally occurs during the test phase of the Traditional methodology when programmers are addressing known software defects.

The following are the steps of Test Driven Development:

1. Pick a feature or a user requirement.
2. Write a test that fulfills a small task or piece of the feature or user requirement (e.g. one method) and have the test fail.
3. Write the production code that implements the task and will pass the test.
4. Run all of the tests.
5. Refactor the production and test code to make them as simple as possible, ensuring all tests pass.
6. Repeat steps 2 to 5 until the feature or user requirement is implemented.

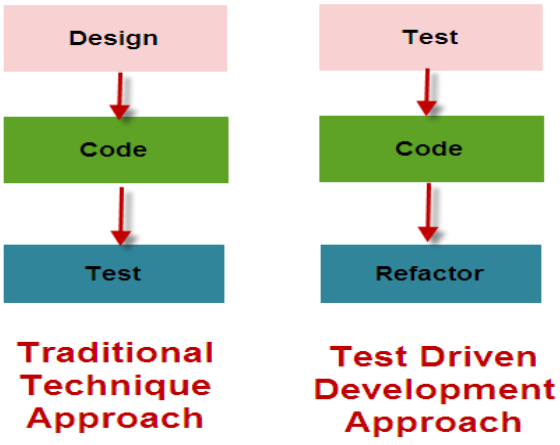

Figure 3: Comparison of TDD and Traditional approach

The following are the steps of Traditional Techniques:

1. Pick a feature or a user requirement.
2. Write the production code that implements the feature or user requirement.
3. Write the tests to validate the feature or user requirement.
4. Run all the tests.
5. Refactor if necessary

Now, we will be doing detail comparative study of different Traditional Techniques and Test Driven Development. We will be discussing the factors that encourage their use and factors that limit their use in industry.

A. Traditional Techniques

Most commonly Software Development techniques are Waterfall Model, Spiral Model and V Model. Though, all of them have different concepts, but all of them have one thing in common i.e. Test is executed only after coding. In below section we will be discussing common stages of traditional techniques.

1. *Requirement Analysis & Specification:* All possible requirements of the system to be developed are captured in this phase. Requirements are set of functionalities and constraints that the end-user (who will be using the system) expects from the system. The requirements are gathered by discussion with client, these requirements are analyzed for their validity and the possibility of incorporating the requirements in the system to be development is also studied. Finally, a Requirement Specification document is created which serves the purpose of guideline for the next phase of the model.
2. *System & Software Design:* Before a starting for actual coding, it is highly important to understand what we are going to create and what it should look like?

The requirement specifications from first phase are studied in this phase and system design is prepared. System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture. The system design specifications serve as input for the next phase of the model.

3. *Implementation and Unit testing*: On receiving system design documents, the works divided in modules/units and actual coding is started. The system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality; this is referred to as Unit Testing. Unit testing mainly verifies if the modules/units meet their specifications.

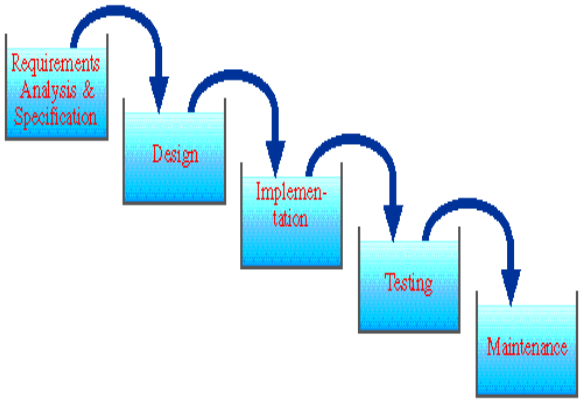

Figure 4: Traditional Techniques

4. *Integration and System testing:* As specified above, the system is first divided in units which are developed and tested for their functionalities. These units are integrated into a complete system during Integration phase and tested to check if all modules/units coordinate between each other and the system as a whole behaves as per the specifications.

After successfully testing the software, it is delivered to the customer.

5. *Operations and Maintenance:* This phase of "The Waterfall Model" is virtually never ending phase. Generally, problems with the system developed (which are not found during the development life cycle) come up after its practical use starts, so the issues related to the system are solved after deployment of the system. Not all the problems come in picture directly but they arise time to time and needs to be solved; hence this process is referred as Maintenance.

Shortcomings:

- All requirements must be known upfront
- No Guarantee if test written will actually be able to test the new functionality.
- Some redundant code got implemented that are never executed.

B. Test Driven Development

On the surface, TDD is a very simple methodology that relies on two main concepts: unit tests and refactoring. TDD is basically composed of the following steps:

• Writing a test that defines how a small part of the software should behave.

• Making the test run as easily and quickly as possible. Design of the code is not a concern; the sole aim is just getting it to work.

• Cleaning up the code. A step back is taken and any duplication or any other problems that were introduced to get the test to run is refactored and removed.

TDD is an iterative process, and these steps are repeated a number of times until satisfaction with the new code is achieved. TDD doesn't rely on a lot of up-front design to determine how the software is structured. The way TDD works is that requirements, or use cases, are decomposed into a set of behaviors that are needed to fulfill the requirement. For each behavior of the system, the first thing done is to write a unit test that will test this behavior. The unit test is written first so that a well-defined set of criteria is formed that can be used to tell when just enough code to implement the behavior has been written. One of the benefits of writing the test first is that it actually helps better define the behavior of the system and answer some design questions

1. Benefits of Test Driven Development

Test Driven Development contributes to software development practice from many aspects such as requirements definition, writing clean and well designed code, and change and configuration management. Few other benefits can be summarized as:

Simpler Development Process: Developers who use TDD are more focused. The only thing that a TDD developer has to worry about is getting the next test to pass. The goal is focusing the attention on a small piece of the software, getting it to work, and moving on rather than trying to create the software by doing a lot of up-front design.

*Improved Communication***:** Communicating how a piece of software will work is not always easy with words or pictures. Words are often imprecise when it comes to explaining the complexities of the functionality of software. The unit tests can serve as a common language that can be used to communicate the exact behavior of a software component without ambiguities.

Improved Understanding of Required Software Behavior: The level of requirements on a project varies greatly. Sometimes requirements are very detailed and other times they are vague. Writing unit tests before writing the code helps developers focus on understanding the required behavior of the software. Each of these pass/fail criteria adds to the knowledge of how the software must behave. As more unit tests are added because of new features or new bugs, the set of unit tests come to represent a set of required behaviors of higher and higher fidelity.

*Reduced Design Complexity***:** Developers try to be forward looking and build flexibility into software so that it can adapt to the ever-changing requirements and requests for new features. Developers are always adding methods into classes just in case they may be needed. This flexibility comes at the price of complexity. In the TDD process, developers will constantly be refactoring code. Having the confidence to make major code changes any time during the development cycle will prevent developers from overbuilding the software and allow them to keep the design simple.

2. TDD relation with other lifecycle stages

2.1 TDD and Software Requirements:

It is a well known fact that one of the main reasons of a software project failure is misunderstood or badly managed requirements. Requirements documented in the form of text or design program have the risk of being incomplete or unclear in comparison to program code, which is formal and by its nature, is unambiguous.

Consequently, simply designed and well decomposed tests reveal the behavior of a piece of code in an unambiguous and clear way. Agile software development methodologies assume that a full set of requirements for a system cannot be determined upfront. Instead, requirements are gathered and modified throughout development leading to a flexible development process. Requirements gathering process is accomplished through small releases and constant feedback from the customer.

In Extreme Programming (XP) requirements are collected as short user stories written on small cards. That means high level business functions are expressed as a collection of finer grained features. This is complementary with the rhythm of test first design, deciding on a feature, writing a small test that validates the feature is working and implementing it. [2]

2.2 TDD and Software Design:

One of the main focuses of agile software development practices is keeping the software as simple as possible without sacrificing quality. That is, any duplication of logic in software must be eliminated and the code should be kept clean. Agile methods proposes a key approach to accomplish these goals; implementing the simplest solution that works for a feature, just enough to pass the tests. Customer requirements constantly change. Trying to estimate possible future requirements and designing the software with this in mind may introduce many unneeded features into the software. This might make the software more complicated and bigger than needed, which in the end would result in additional maintenance burden.

Refactoring simplifies that burden and thus test driven development also helps designing modular and well decomposed software.

2.3 TDD and Software Maintenance:

The design of a software system tends to decay during its lifetime, as new or changing requirements or bug fixes patched into the code unless it is continuously refactored and rearchitectured. Without a suite of complete regression tests, refactoring software is practically impossible because it will not be possible to know which parts of the software are affected from a change. Test driven development includes continuous but small refactoring into the development activity itself. The code, test, refactor cycle is applied at every small step so that the design is always kept clean and the code is always kept working.

2.4 TDD and Software Documentation

Documenting software code is a daunting and mechanical task. Just like software design tends to decay in time, documentation tends to get outdated, including the comments in the code. Besides, it is not possible to validate the documentation by an automated process. In contrast, automated tests are always kept up to date because they are run all the time. An agile methodology does not claim that all documentation should be replaced with automated tests; however they tend to keep the documentation as small as possible.

IV. RESEARCH METHODOLOGY

The methodologies used in this research are systematic literature review using guidelines from Kitchenham [5] and survey. According to Kitchenham, a systematic literature review or systematic review is a mean of identifying, evaluating and interpreting all available research relevant to a particular research question, topic area or phenomenon of interest.

This research study will utilize both qualitative and quantitative research methods to analyze the compare results of TDD and rest Traditional techniques. In this research, as there are a number of studies particularly from software practitioners reporting their experiences of using Test Driven Development in many different ways, the systematic review is selected as an appropriate methodology in order to summarize the existing empirical evidence regarding TDD practices adaptation.

A. Objectives:

The goal of this research is to compare Test Driven Development with Traditional techniques for the purpose of evaluating internal quality, programmer productivity, and programmer perceptions.

The objective of this study is:

- To identify the factors affecting the choice of the Test Driven Development in the software/IT industry
- To determine the factors limiting the adoption of TDD in industry To ensure validity of results, various methods will be used for validation and verification.

B. Data Collection:

Authors have undertaken a survey-based approach to assess use software life cycle models in Indian Software Industry. In a survey based approach the usual proceeding to gather information is the usage of questionnaires or interviews. These are applied to a representative sample group and the outcomes are then analyzed. Both qualitative as well as quantitative data can be derived from this strategy.

Questionnaire survey methodology was preferred for this research since it is a reliable and economical method for data collection. A questionnaire was distributed through Survey Monkey to gather survey data.

C. Survey Design:

The web-based survey contains ten questions in total and can be divided into two main parts: demographics and Comparison of test driven development with traditional development.

Survey Link http://www.surveymonkey.com/s/9LNJ2HZ

The demographics of the respondents are addressed by question one to three. It covers the following information:

- The respondent's role in the company
- Respondent's knowledge of Traditional development
- Respondent's knowledge of Test Driven development

Next section starts from question four to ten which discuss comparative study of Traditional Technology with Test driven development on the basis of respondent's work experience.

It also covers following information:

- It discusses organization's main drivers for introducing Test Drive Development
- Effect of TDD on various parameter are discussed
- Main difficulties impacting industry acceptance to Test driven development

D. Pilot and Implementation

In order to make the survey as comprehensive and compact as possible, a survey pilot was performed to test whether the respondents understand the questions and whether all respondents interpret the meaning of the questions in the same way. The invitation emails for the pilot survey were sent to 5 recipients who are the sample of the target population members. The pilot survey was opened for two days. The feedback from the pilot participants were collected and used to modify the survey design. The results from the pilot show that the survey design was good enough, only some minor changes were made.

After the survey design was finalized and updated on the online survey tool, the survey was released over a period of three weeks (9 Jan – 5 Feb 2013). The invitation emails were sent to numerous software companies. We also identified several online discussion groups e.g. LinkedIn and Face book that focus on Test driven development and posted a solicitation message inviting the group members who had experience using Test driven development approach. Attempts were made to gather different perspectives of Test driven development as represented by different team roles.

In all 15 software development companies were selected on random basis. The lists of companies/Institutes the questionnaire has been distributed are as follows:

1) Aon Hewitt, Gurgaon
2) Naukri, Noida
3) Clear2pay, Noida
4) HCL, Noida
5) Sapient, Bangalore
6) SAP labs, Gurgaon
7) Nucleus, Noida
8) Accenture, Noida

9) Reliance (ADA), Mumbai
10) TCS, Pune
11) Avaya, Pune
12) Erikson, Gurgaon
13) AMDOCS, Pune
14) UHG, Noida
15) Infosys, Hyderabad

In total 52 software tester participated in the study. The percentage for each item was calculated to reach the inference. The view point of the participant was tabulated and graphs were made to shows the results.

E. Experience Sharing:

At the end of our survey, which consisted mostly of structured questions, we invited our respondents to share additional comments about Test Driven Development and traditional techniques and their experiences with using them. This helps us to get to know some other facts and key points that we might not able to cover in survey. However, in reviewing the literature and with questionnaire, these methods were seen as most reflective of current usage and were considered most likely to produce practical information. The use of a mix of qualitative and quantitative research methods provided an opportunity to gain a better understanding of the factors that impact developers experiences in Test Driven development.

V. DATA ANALYSIS AND RESULTS

Participants came from a range of organizational and team structures; from large multinational companies to small start-ups, and from entirely self-regulating teams to teams with high levels of management supervision.

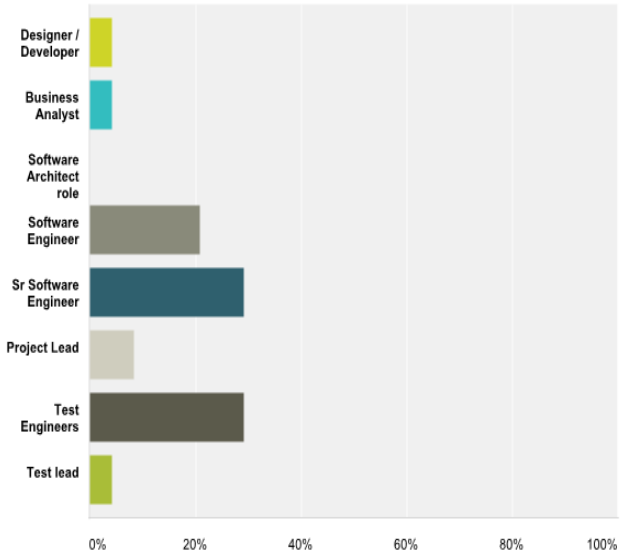

Figure 5: Respondent's profile

Survey Question 1

The first research question was, ''How would you rate your knowledge of Traditional Methodologies?''

From the findings discussed above, the answer is clearly "Average" or "Extensive". These Traditional developments are in trend from past so many years that we have worked on either of them. Thus consider their knowledge extensive. Few people believe these traditional technologies are so vast that it would be wrong if they say their knowledge is so extensive. But half of them believe, they have worked so much on these methodologies that had that much knowledge

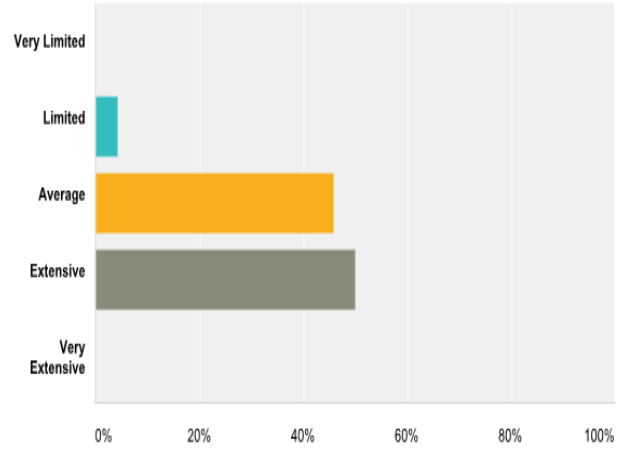

Figure 6: Respondent's knowledge of Traditional techniques

Survey Question 2

The first research question was, ''How would you rate your knowledge of Test driven development?''

From the findings discussed above, the answer is clearly "Average". People somehow feel TDD approach is too new to the market, so in comparison to Traditional Technology their knowledge of TDD would be average or less.

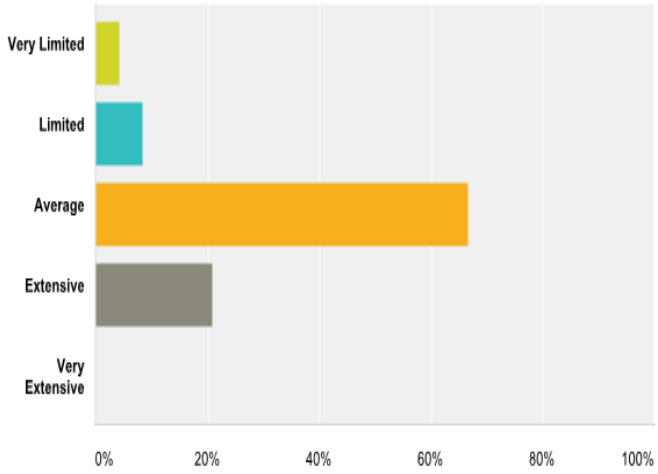

Figure 7: Respondent's knowledge of TDD

Few respondents rated their knowledge of TDD as extensive as they have prior that much experience for this.

Survey Question 3

The next research question was, ''what is your personal belief in the effectiveness of Developer TDD (Check all that applies, if any)? '' Majority of the participants considers TDD "Has some potential for quality improvement" and "Will increase ability to react to stakeholders changing needs". Few participants believe TDD will increase the change of project failure. Rest People somehow don't want to give any opinion on this.

While sharing their experience, we came to know that respondents are not against the TDD but the main factor that limits TDD adoption is Company acceptance to TDD.

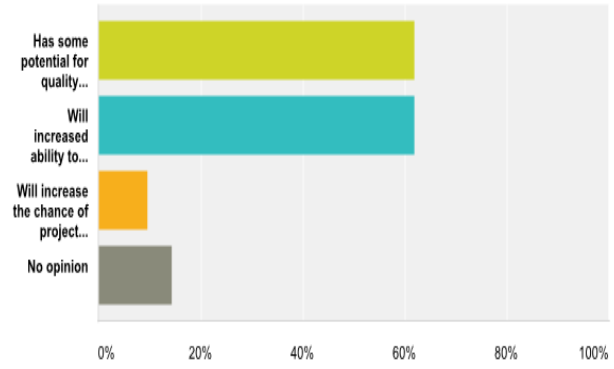

Figure 8: Respondent's belief for TDD

Survey Question 4

The next research question was, ''What benefits of Developer TDD have you Actually experienced (check all that apply, if any)?"

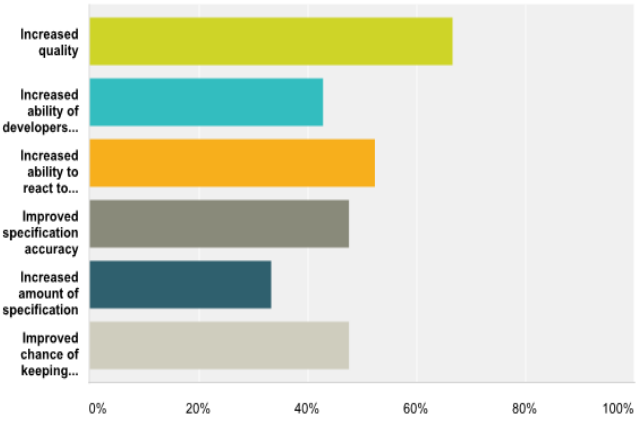

Figure 9: Benefits respondent experienced

From the finding below, this clearly shows, all benefits are equally important for TDD.

"Increased quality", "Increased ability of developers to safely change software", "Increased ability to react to changing stakeholder needs", "Improved specification accuracy", "Increased amount of specification" and "Improved chance of keeping specifications in sync with the code"

Survey Question 5

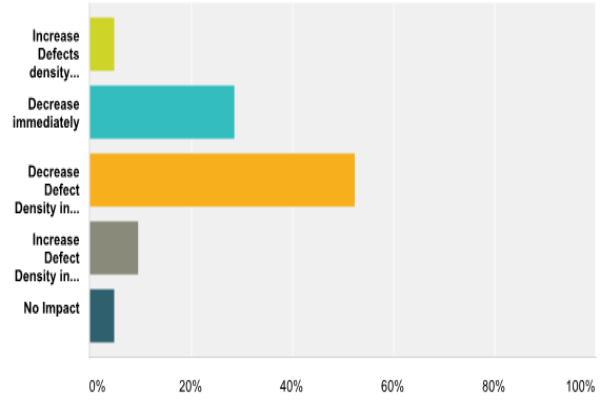

Figure 10: Effect of TDD on defect density

The next research question was, "How much Test Driven Development impact Defect Density in comparison to traditional technique?" With regards to defect density, all respondents cited that TDD decrease defect density either immediately or in the long run. Simply designed and well decomposed tests enables them to understand the requirement more clearly and unambiguous which decrease defect density.

Survey Question 6

The next research question was, "How much Test Driven Development impact Productivity of software in comparison to traditional technique?" When the impact of productivity of software was presented to the respondents the majority of the respondents were inclined to say that TDD improves the productivity in the long run.

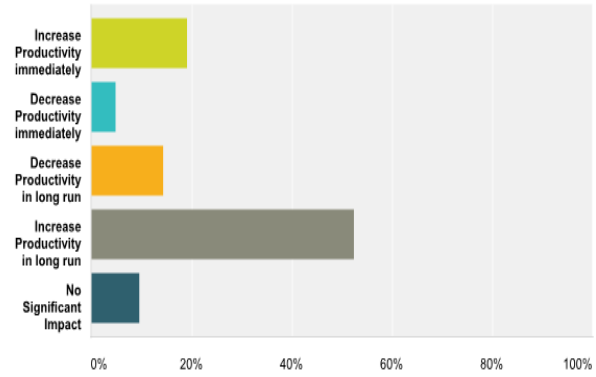

Figure 11: Effect of TDD on productivity

Few of the respondents stated that they have experienced increase productivity immediately where as few experienced the opposite i.e. decrease productivity immediately.

Survey Question 7

The next research question was, "How much Test driven developments impact the Complexity of source code in comparison to traditional technique?"

The findings of the effects of TDD approach on code complexity of source code do not support my earlier reading and observed study in this section. Through Literature survey, we noticed that refactoring decrease code complexity. But more than 60% of the respondents stated that there was no significant impact on Code complexity while using TDD approach. Few of the respondents, find that TDD approach often results into decreasing code complexity.

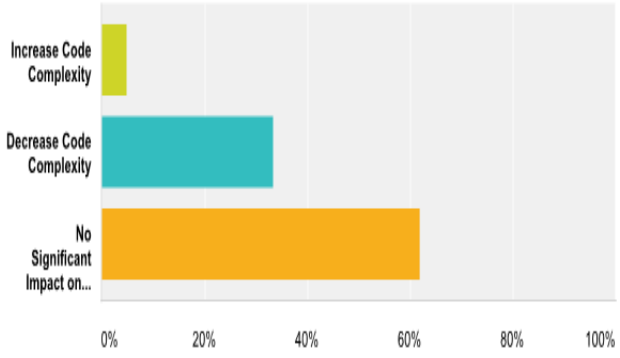

Figure 12: Effect of TDD on code complexity

Survey Question 8

The next research question was, "What is your opinion about ease of learning TDD approaches (Junit, acceptance TDD)?"

When asked about ease of learning TDD approaches, Most of the respondents had neutral opinion of learning TDD approaches. Few people think it was difficult to learn them. When asked about the reason that we lead to conclusion inadequate training material, reluctance to accept new approach is somehow the factor that make them difficult to learn TDD approach.

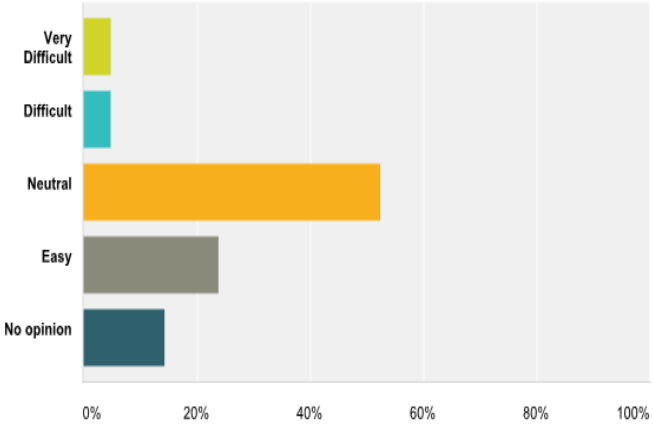

Figure 13: TDD ease of learning

Last Question was "What are the factors that are limiting the adoption of Test Driven development in industry (check all that apply, if any)?"

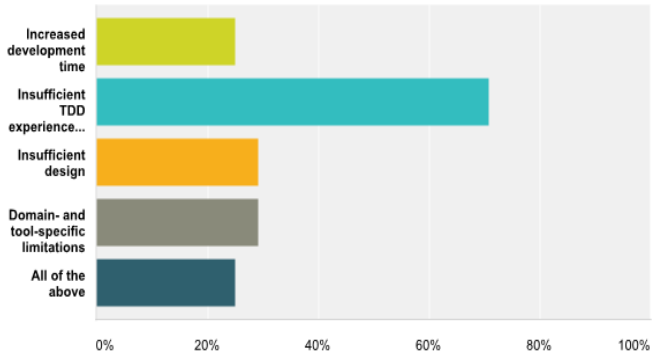

Figure 14: Factors limiting TDD adoption

Finally we tried to find out the biggest barrier to the adoption of Test Driven Development in industry, and came to conclusion that Main factor that limits TDD adoption is in sufficient TDD experience. Also With any significant process change, the biggest barrier is the ability to change organizational culture followed by general resistance to change. The other common concerns listed by respondents were increased development time, insufficient design and Domain and tool specific limitation.

Threats to Validity

In this section, we list threats that potentially would invalidate our results and findings, in order to limit the scope our claims as well as explain their utility.

Experience: Although many respondents showed great theoretical and practical knowledge about TDD, it is hard to know if they do exactly the way they reported. In addition, a considerable amount of them talked about practices that differ from what TDD suggests. As some of them do not follow TDD steps the way they theoretically should, it may influence their opinion.

Subjective biases: There is the possibility of survey participants were subjective biases such as TDD proponents trying to claim TDD success in introductory projects (in order to promote the adoption of their methodology), and the lack of independent, non-TDD advocates in the survey.

Population Size: Lastly, the sample size was still small, considering the large TDD community population. A larger sample size could provide more robust and accurate statistical calculation and analysis, and also could include other agile methods that were missing in this sample size.

VI. CONCLUSIONS AND FUTURE WORK

Though, the existing studies provide valuable information about Test-driven development, but most were based on empirical studies or qualitative study amongst the university students. Our research is more focused on industrial survey, Software developers, Testers and other software personnel from different MNC participated in this survey and discussed their personnel experiences while working on TDD.

Through the literature review and observations resulted on analyzing the data collected in the survey, we came to following conclusion:

1. *Benefits:* Although we have discussed many benefits for TDD in literature survey, Main benefits that were observed in this survey are TDD has potential for quality improvement and increase ability to react to stakeholders changing needs. On further discussion with the participants, we tried to find out the reason behind these benefits. With changing industry trends, Software companies have had to dramatically change their approach to quality to create the higher quality products that consumers are now demanding, TDD approach enables thorough unit testing which improves the quality of the software and enhance customer satisfaction.

Productivity: This study provided substantial evidence that Test-Driven Development is, indeed, an effective tool to improve productivity in long run. Though it was observed that productivity gets decreased immediately while implementing TDD because of increased development time, but with time, Productivity starts **increasing**.

2. *Complexity of source code:* Regarding code complexity, our literature results were quite different with the survey results. In literature review, we noted refactoring decrease code complexity but most of the respondents believe neutral effect on code complexity.
3. *Defect density:* With regards to defect density, Most of the respondents believe TDD can significantly reduce the defect density of developed software either immediately or in the long run. We have got similar picture through literature survey.
4. *Ease of learning:* Despite, most of the respondents give neutral feedback for ease of learning, but on further discussing, we found that main factor that drive ease of learning is TDD requires the change in mindset for those who have chosen to learn it.

Factor limiting the adoption of TDD in industry: Most common factor that limits TDD adoption is insufficient TDD experience and tools specific limitation. As TDD is new approach, so we don't have that much experience in that domain as comparison to Traditional approach. Increased development time, insufficient design description, reluctance to new approach and upper management support are other factors limiting TDD adoption.

These results need to be viewed within the limitations of the experiments conducted. Further controlled studies on a larger scale in industry could strengthen or disprove these findings. Also, this research includes qualitative study, adding Empirical study will strengthen the results.

ACKNOWLEDGEMENT

I would like to thank all of the participants of the survey whom we were fortunate enough to work with throughout this research. I would also like to thank my guide, Dr Sanjeev Bansal (Amity University) for his helpful feedback.

REFRENCES

1. Shrivastava and Jain, "Metrics for Test Case Design in Test Driven Development", International Journal of Computer Theory and Engineering, Vol.2, No.6, December, 2010, Pg: 1793-8201.
2. Astels, D., Test-Driven Development: A Practical Guide. Upper Saddle River, New Jersey, USA, Prentice Hall, 2003
3. Beck, K, Test-Driven Development By Example. Boston, Massachusetts, USA, Addison-Wesley, 2003
4. Maria Siniaalto, "Test driven development: empirical body of evidence", Agile Software development of Embedded Systems, 2006
5. Kitchenham, B, Procedures for Performing Systematic Reviews. United Kingdom and Australian: Department of Computer Science Keele University, Australian Technology Park, 2004
6. L. Koskela, Test Driven, Manning Publications, Greenwich, Connecticut, USA, 2008.
7. B. George and L. Williams, "An initial investigation of test driven development in industry," presented at ACM Symposium on Applied Computing, Melbourne, Florida, 2003.
8. Adnan Causevic, Daniel Sundmark and Sasikumar Punnekkat "Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review" IEEE Computer Society Washington, DC, USA ©2011
9. John Huan Vu, Niklas Frojd, Clay Shenkel-Therolf, and David S. Janzen "Evaluating Test-Driven Development in an Industry-sponsored Capstone Project" 2009 Sixth International Conference on Information Technology: New Generations
10. Janzen, D., Software Architecture Improvement through Test-Driven Development. Conference on Object Oriented Programming Systems Languages and Applications, ACM, 2005.