Management College of Southern Africa

Understanding the effectiveness and benefits of implementing Test Driven Development in Software Development Process for Company XYZ

Yoganand Aiyadurai

MBA

2012

Understanding the effectiveness and benefits of implementing Test Driven Development in Software Development Process for Company XYZ

By

Yoganand Aiyadurai

Dissertation submitted in partial fulfilment
of the requirements for the degree of
Masters of Business Administration in the

Department of Business Studies
Management College of Southern Africa (MANCOSA)

Supervisor: Ms. Trishana Ramluckan

2012

Declaration

I, Yoganand Aiyadurai, do hereby declare that this dissertation is the result of my investigation and research and that this has not been submitted in part or full for any degree or for any other degree to any other University.

__________________	31st August 2012
Yoganand Aiyadurai	Date

Acknowledgements

My utmost *gratitude to my company,* the *valuable experience,* gained all these years has shaped me as an ethical and demanding professional in my field, and I thank for the MBA sponsorship.

I appreciate my colleagues and team members who helped me during my absence at work. I thank my Manager *Eugene J.,* who drafted my initial motivation letter for MANCOSA, approved all my study requests/leaves without questioning and to *Jodi J.* who proactively approved/arranged all my course fees well in time.

I dedicate this degree to *my daughter Tanvi,* the purpose of my life became questionable, meaningful and purposeful after she was born. She was my inspiration and motivator for my studies, and she will be in the future. To *my wife Arunmozhi*, who kept me comfortable, supported, and tolerated me all the way.

Sincere thanks to two esteemed people, my friend *Manjula Bandara* and his wife *Sureshna Bandara*. They helped and stood behind me during difficult times, in the initial period of my stay in South Africa, and because of whom I was able to stay in this country till today. This mile stone was unreachable without them.

To *my Mother and Father* who always wanted to see me successful, especially my father, who worked hard to elevate me to this level. It is unfortunate that my Father is no more to witness this event, but where ever he is, he always has an eye on me, protecting, and clearing obstacles in my life's path. Also to my Mother, who always blessed me and provided me with all my needs.

To *Safari books online digital library* (www.safaribooksonline.com) that provided me, the latest technology books used in this research and for making the content search easy.

I appreciate the enthusiasm and guidance of my supervisor *Ms. Trishana Ramluckan,* who made possible the evaluation and signing off my dissertation in time. I was under stress to graduate this year, so that I can qualify for my D. Tech studies starting in January 2013.

Finishing off with a quote from Swami Vivekananda "*We want that education by which character is formed, strength of mind is increased, the intellect is expanded, and by which one can stand on one's own feet*". Equally the nurturing, education and degree received from the college will take me and my profession to the next greatest levels. Thank you *MANCOSA.*

Abstract

The business management focus of this research is to find the effectiveness and benefits of implementing Test Driven Development (TDD), in software development process and to come up with recommendations for company XYZ, on how they can make a transition from the existing software development methodology followed, to the new software development methodology in the most effectual way. This research also highlights the key factors that must be considered and dealt with, when adopting TDD. The provided recommendation is applicable for any software development companies or organisations wanting to move to TDD.

In the traditional software development process, unit and functional tests are written after the code is implemented. Recently agile software development methods were introduced which also change traditional testing practice. TDD is a practice of extreme programming (XP) where unit and functional tests drive the development of the code. This means that the tests are written before the actual production code that is going to be tested. TDD is also known as test-driven design as the tests drive the design of the software. With TDD any change to the code is tested, refactored and the test is performed again. The process is iterated until each unit is functioning according to the desired specifications. So the design of the software always evolves all the time. TDD can produce applications of high quality, in less time when compared with the traditional software development methods. The methodical nature of TDD ensures that all the units in an application have been tested for optimum functionality, both individually and in synergy with one another. Because tests are conducted from the very beginning of the design cycle, time and money spent in debugging at later stages is minimized.

Published empirical studies of TDD, claims that TDD produces good software design, better code quality, application quality, and developer’s productivity.

The goal of this research is to collect and confirm all the claims. Evaluation in this research was done in two steps. The first step was to study the literature for supporting or contradictory evidences. The second step was to analyse and study the results of evidence-based research on TDD. This research took a two sided approach by comparing the TDD survey done by Scott Ambler in 2008 across the world and the perceptions about TDD by practitioners, who participated in the direct interview. In addition the study also validates the claimed disadvantages of adopting TDD from the real practitioners of TDD. This research furthermore provides information on the precautions and steps that are needed for any organization to successfully implement TDD.

The researcher concludes this abstract with a caution. This is that even though it requires a large amount of effort and investment to move towards and adopt TDD, it will result in improving the way you design, develop, test and deliver the software and software products.

Table of Contents

List of Acronyms

BA	-	Business Analyst
BDUF	-	Big Design Up Front
BDD	-	Behaviour Driven Development
CMM	-	Capability Maturity Model
DRY	-	Do not repeat yourself
ISACA		Information Systems Audit and Control Association
IT	-	Information technology
KISS	-	Keep It Simple Stupid
KLOC	-	Thousand lines of code
LOC	-	Lines of Code
MVC	-	Model-View-Controller
MVVM	-	Model View View-Model
OO	-	Object Oriented
SDL	-	software development process
SDLC	-	System Development Life Cycle
SOLID	-	Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion
TDD	-	Test driven development
XP	-	Extreme Programming
YAGNI	-	You Aren’t Going to Need It

List of Tables

List of Figures

Chapter 1 – Introduction

1.1 Introduction

Software development phase is perhaps the most important part of any software project. A significant amount of time and effort are expended in checking and eliminating errors in the code. To deal with size, complexity, and rapid change IT (information technology) organizations are increasingly looking for better software development methodologies. Agile methodologies have been adopted by many organizations in the last decade. Academic researchers claim that using agile methodologies, organizations can enhance better productivity and deliver a higher quality of software.

Test Driven Development (TDD) is a software development methodology derived from Extreme Programming (XP) and the principles of the Agile Manifesto (Agile Alliance, 2000). In TDD the unit and acceptance test cases are incrementally written prior to writing the production code that guides the design of the target software. TDD has gained considerable attention among practitioners and researchers, outside its original context in Extreme Programming (XP), with the promise of a number of benefits to the software development process.

In traditional development, tests are built after the target product feature exists, also known as “Test-Last”. In contrast to the traditional Test-Last development methodology, TDD requires developers writing automated tests prior to developing functional code in small, rapid iterations. This is highlighted in the rule suggested by Beck (2003:8) for developers using TDD: “Write new code only if an automated test has failed”.

Krishnan et al. (2000:745) found that higher personnel capability, early deployment of resources in the design phase of development and improved process factors aid in achieving a higher quality in software product. One of the most prevalent techniques to assure quality of software is through testing. To be fully assured of the quality of software, exhaustive testing is needed and a study by Kuhn et al (2004:418) claims that a finite number of tests can potentially assure the highest level of quality in a software product. Regardless of the number of the tests required, the quality of the software product is widely accepted to be connected to testing.

Nachiappan et al. (2008:292) provide significant supporting evidence that TDD improves the application quality, design and reduces defect rates. The results from some experiments conducted by TDD researchers, Janzen et al (2008:71) that TDD improves the overall productivity by reducing the design and rework efforts.

Koskela (2007:22) argues that slicing the overall task into a sequence of small tests makes the measuring of progress toward the ultimate goal easier, and TDD by its nature encourages such small steps with evolutionary design. It is common that the section of code that requires a change is part of the interface between two components and therefore, it is much more difficult to change. No developer dares to change the code due to fear of breaking the functionality. Evolutionary design is a mind-set that requires the developer to make that change, nurture the living, and grow the code base, instead of protecting it from the bad world that wants change. Thus, improvement in the quality of design and the quality of the whole system takes place.

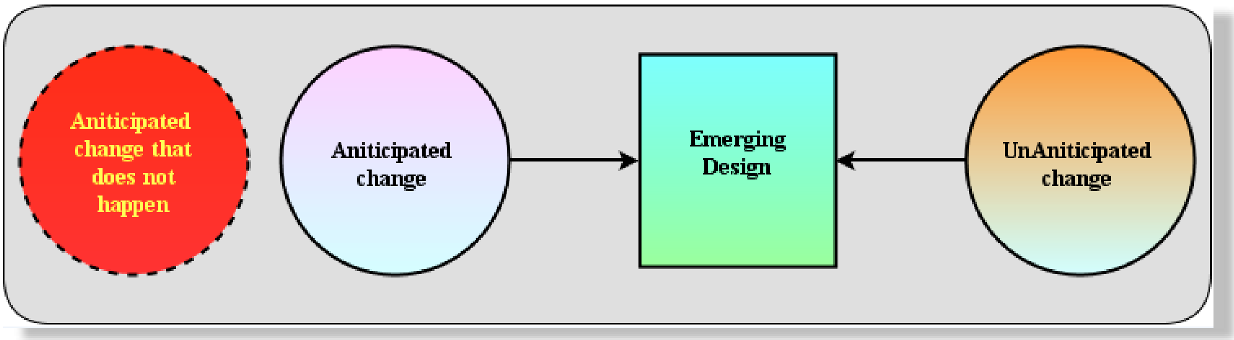


Figure 1: Evolutionary design (Koskela, 2007:22)

It is therefore no surprise that the bulk of progress made in the field of software engineering in the last forty years has been focused on improving the techniques used to build software. From the *Capability Maturity Model* (CMM) for software process improvement to the *Agile Manifesto*, from *Big Design Up Front* (BUFD) to *Iterative Design*, and *from Waterfall to eXtreme Programming* (XP), significant efforts have been made within the industry to improve the ability of professional programmers to write high-quality software for large, critical systems. Figure 2 and 3 shows the increase in the demand of jobs with TDD skills over the years. Test-Driven Development is catching on in the industry, but as a practice it is still remarkably young, which was proposed eleven years ago by Kent Beck.

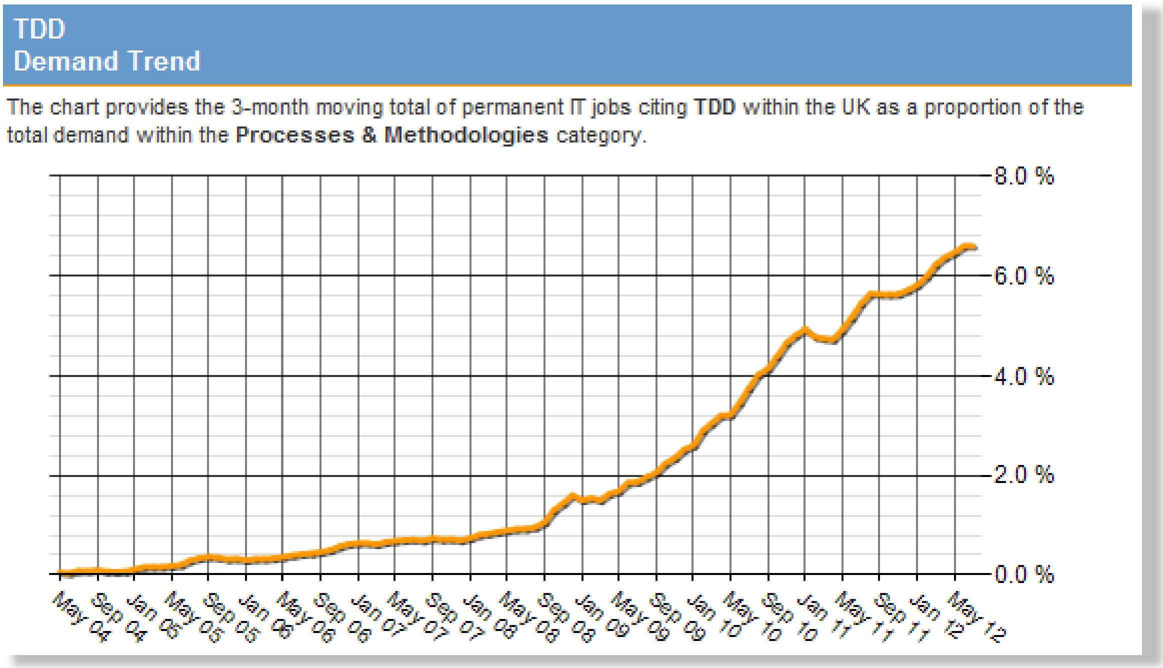


Figure 2 - Job trends for TDD from May 2004 to May 2012. (IT jobs watch, 2012)

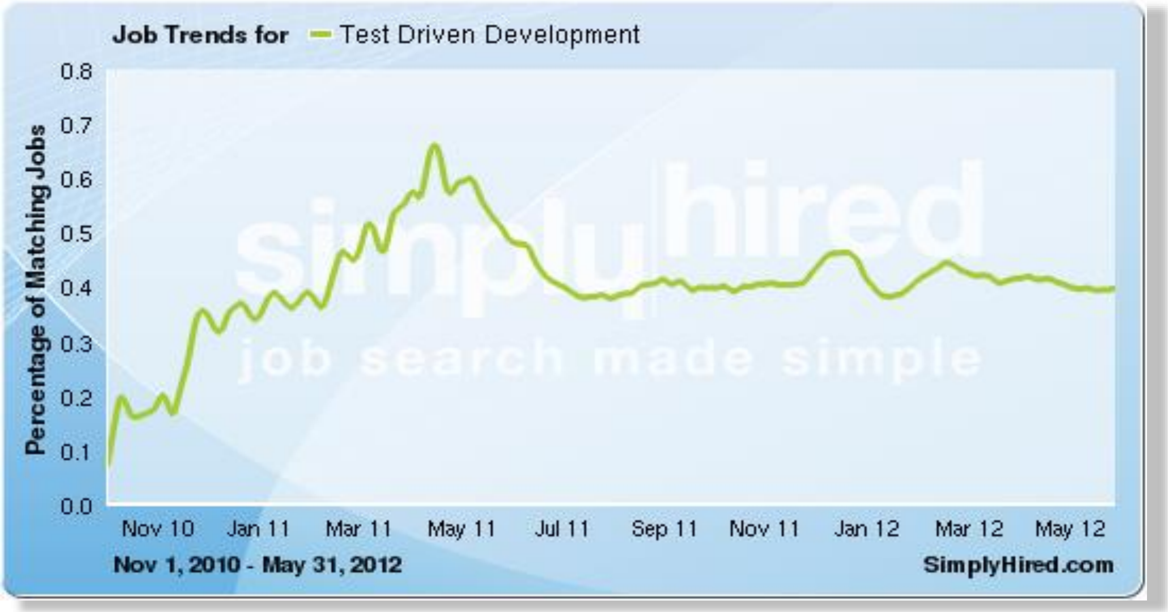


Figure 3 – Job trends for TDD from Nov 2010 to May 2012 (Simply hired, 2012)

The research therefore aims to contribute to the evidence-based research in TDD by conducting the investigation in an industrial context. This study focuses on understanding practitioner’s perceptions of software quality, developer’s productivity, and the contributing factors. Researchers can claim benefits, but it is whether or not the practitioners perceive those benefits as “real” that will influence the pressures for adoption of TDD to improve the software development process (SDP). This study is also expected to provide insights that will be useful for improving current software development practice, methodologies, and it should be of value to both the software development research community and the companies keen to adopt TDD.

1.2 Background to the study

Company XYZ is an extremely large, mature, and pioneer software development company with more than 18 years of experience, delivering hi-tech, robust, and innovative software solutions and products. Company XYZ has a work force of over 1200+ people and around 400+ software developers. Company XYZ is the market leader in its domain globally, and the urgency to get the software products first into the market during demanding times is exceptionally high. There is an ongoing drive over the years to improve the software development process and the quality of the software produced. Implementing the new software development methodologies like TDD addressed in the aim of study will undoubtedly help Company XYZ to amend improvements to the software development process and will contribute to delivery of high quality software products quicker. This research also focus on the developer’s productivity by using TDD that can also add to the cost effectiveness of the software production.

1.3 Problem statement

Company XYZ have adopted the agile software methodology using the scrum framework and have gained a respectable control over the software development process (SDP) for the last 4 years. The software release time using agile software methodology is more than 4 weeks, with regression testing of the software taking 2 weeks. The maintenance and support of the software in production are also becoming a challenge due to unmanageable code and lack of unit tests. Introducing TDD into the SDP for Company XYZ has many benefits like higher software quality, reducing maintenance cost by 80%, quick regression test, adding new features quickly and easily.

1.4 Aim of the study

The aim of the research is to find out about the benefits of TDD by means of case study from the practitioners of TDD in the industry, weigh these benefits and to make recommendations on how best TDD can be implemented to boost the software design, code quality, application quality and developers productivity.

1.5 Objectives of the study

The following research objectives will guide the researcher to explore the research problem:

1. To investigate the factors that contribute to the implementation of TDD in the software development process from the software developers who have implemented TDD.
2. To find out the benefits of using TDD in the software development process from the software developers using TDD in the Industry.
3. To examine and compare the benefits of TDD as seen by the software developers practicing TDD, against the claims from the researchers on code quality, application quality, and developer's productivity.
4. To find the perception of software developers, using TDD to suit their needs.

1.6 Research questions

The research questions related with the above mentioned objectives are as follows:

1. ***Research objective 1***
   - i. What are the benefits of TDD found in practice across companies/organizations?
   - ii. Are there any disadvantages of using TDD in practice across companies/organizations?
2. ***Research objective 2***
   - i. What are the underlying factors that contribute to the benefits of TDD?
3. ***Research objective 3***
   - i. Do the benefits of TDD argued in literature on code quality, application quality, and developer's productivity match with the benefits agreed by the software developers in the industry?
4. ***Research objective 4***
   - ii. Are the software developers happy practicing TDD currently in the SDP compared to SDP before TDD?
   - iii. Do the software developers feel that the TDD theory in the literature needs any adaptation from their experience?

1.7 Significance of the study

The areas of software development process are wide, so this research focuses on 3 crucial outputs of software development and the effect of TDD on the following factors:

1. Effect of TDD on software design and code quality.
2. Impact of TDD on the overall software application quality.
3. Significance of TDD on the software developer's productivity.

This research will help the companies that are planning to practice TDD in their software development process. Apart from the scope of research, additional stats will be collected from the experience of the software developers who have hands on experience with TDD that can help the researcher community to weigh TDD and for carrying out further research.

1.8 Format of the study

1. The study will commence in chapter 2 with a review of the relevant literature. The focus will be on effectiveness and benefits of implementing TDD. The literature review in chapter 2 starts with concepts of software, the software development process, and the software development life cycle. The discussions are followed by the comparison of the traditional waterfall software development methodology with the new agile software development methodologies. The study looks in-depth into the concepts of TDD and the theories behind them.
2. Chapter 3 will present the research methodology chosen for this study. This will include explaining the design, philosophy, strategies, sampling and the instruments that will be used for research.
3. The findings of the primary and secondary research and analysis of data will be presented in chapter 4.
4. A thorough discussion of the findings will follow in chapter 5. The analysis and the interpretation of data collected will be argued convincingly with the literature review in chapter 2.
5. Chapter 5 will conclude with the feasible recommendations and final conclusions keeping with the objectives of the research.

1.9 Conclusion

TDD is one of the profound practices of extreme programming. TDD being a disciplined software development technique adds many positive benefits to the traditional software development process. This will be proved in the literature review as we proceed to chapter 2. The TDD practitioner’s involvement in this research will add significant value to evaluate the literature claimed benefits and disadvantages if any against the actual practice of TDD. So the recommendations and conclusions from this research is expected to be feasible and practical, that can be used by the research community and for any organization that is keen to move towards the implementation of TDD.

Chapter 2 - Literature Review

2.1 Introduction

The chapter provides the literature review for TDD. The chapter begins with the definition of software, the software development process, and the software development life cycle. The chapter proceeds with the comparison of the traditional waterfall software development methodology with the new agile software development methodologies. The literature review looks in-depth into the concepts and theories of TDD and the contributing factors that make TDD successful.

2.2 Software

Agarwal et al. (2009:3) defines software as a set of instructions used to acquire inputs and to manipulate them to produce the desired output in terms of functions and performance as determined by the user of the software. Software includes instructions also known as computer programs, when executed provide desired functions and performance. The data structures enable the programs to adequately manipulate information and store data (end results) in a desired format using the data structures which are created by users with the software.

Mishra et al. (2011:2) defines software as a set of instructions also called as computer. Software is a general term for the various programs and data stored on computer storage devices to operate the computer and related devices. Software controls the operation of computer hardware. Without the instructions provided by the software, computer hardware is unable to perform any of the tasks associated with computers.

2.3 Software Development Process

Esposito et al. (2010:24) states that software development is a process that is created and formalized to handle complexity and its primary goal is to ensure expected results. The ultimate goal of software development process is to control complexity and not creating it. To achieve this goal, a methodology is required that spans the various phases of a software project. According to Mishra et al. (2011:22) software development process is a structured set of activities that transform the user’s requirements into a quality software product. El-Haik et al. (2010: 22) argues that software developmental processes aid management and decision making where both requires clear plans and a precise, quantified data to measure project status and make effective decisions. Defined developmental processes provide a framework to reduce cost, increase reliability, and achieve higher standards of quality. There are many software development methodologies that are in use or that were used in the past for various type of software projects based on the complexities and sizes of software for the requirements of different industries. Below is a list of some of the software development process or methods (El-Haik et al., 2010: 23).

1. Waterfall
2. Sashimi Model
3. V-Model
4. Spiral
5. Joint Application Development
6. Rapid Application Development
7. Model Driven Engineering
8. Iterative Development Process
9. Agile Software Process
10. Unified Process
11. Extreme Process (XP)
12. Wheel and Spoke Model
13. LEAN method (Agile)

2.4 Systems development life cycle

Researchers have explained the process of software development using different models. Systems development life cycle (SDLC) is generally agreed upon as an accepted way of modelling the process of software development.

Marchewka (2012:33) state that the development of new products, services, or information systems follows a product life cycle. The most common product life cycle in information technology is the systems development life cycle (SDLC), which represents the sequential phases or stages an information system, follows throughout its useful life. The SDLC establishes a logical order or sequence in which the system development activities occur and indicates whether to proceed from one system development activity to the next. Even though there are variations of the SDLC, the life cycle depicted in Figure 4 shows the generally accepted activities and phases associated with systems development.

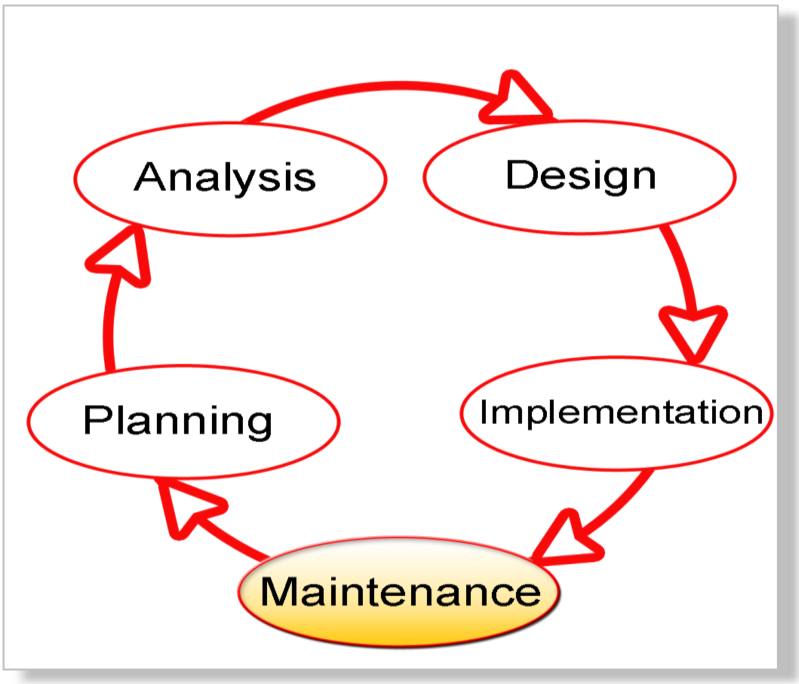


Figure 4: Systems development life cycle (Marchewka, 2012:34)

Eventually, the system becomes part of the organizational infrastructure and becomes known as a legacy system (Marchewka, 2012:34).

Cannon (2011:295) argues that all computer software programs undergo a life cycle of transformation during the journey from inception to retirement. The system development life cycle (SDLC) used by Information systems audit and control association (ISACA**),** which is an international professional association that deals with information technology governance, is designed as a general blueprint for the entire life cycle. A client organization may insert additional steps in their methodology. This international SDLC model comprises seven unique phases with a formal review between each phase as shown in figure 5.

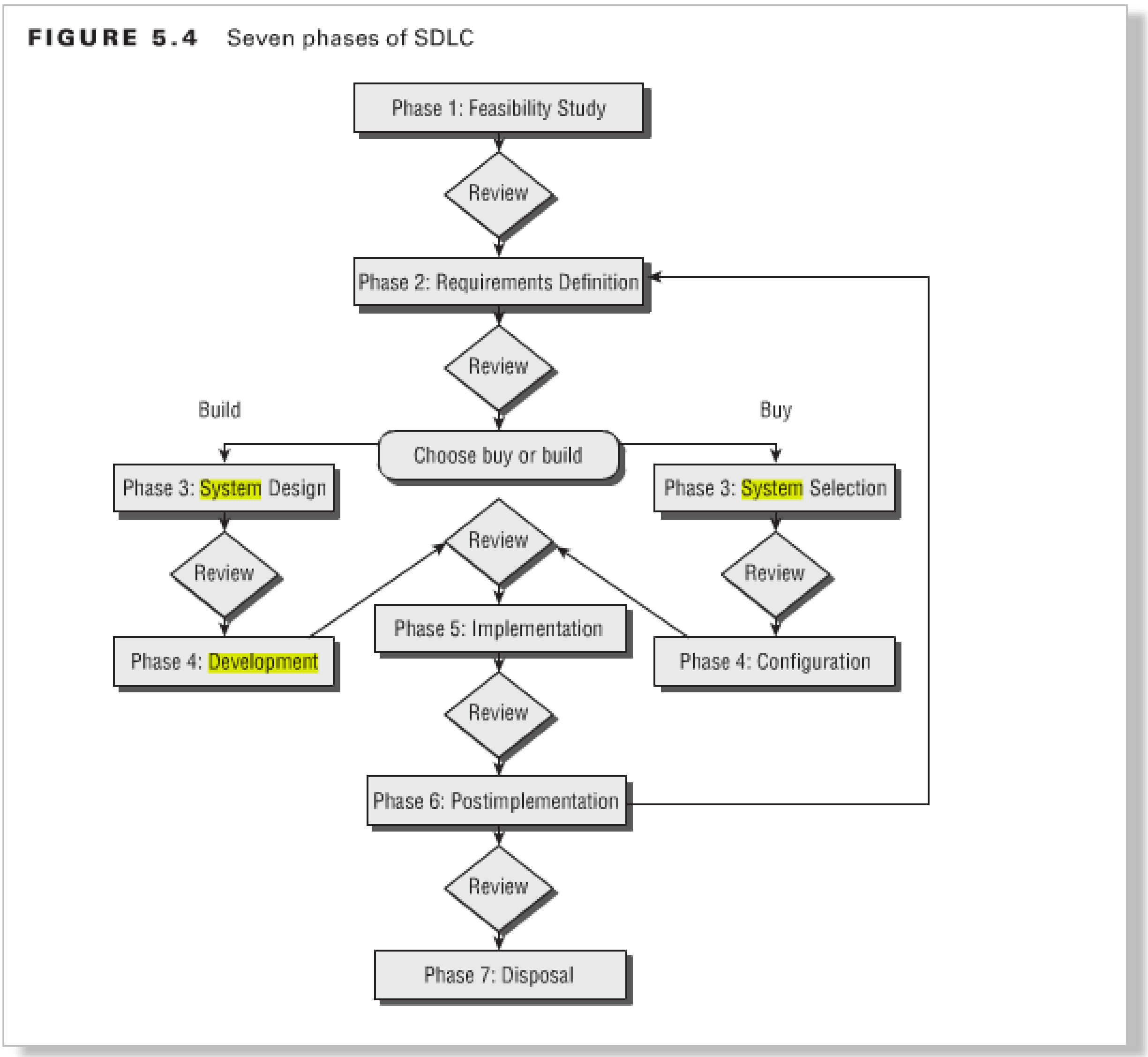


Figure 5: Seven phases of systems development life cycle (Cannon, 2011:297)

2.5 Traditional software development Methodologies

The best-known and the oldest methodology that is still followed are the waterfall model. In the waterfall model, software development proceeds from one phase to the next in a purely sequential manner. Figure 6 shows the steps that are followed in a waterfall model (Esposito et al., 2010:26).

The primary characteristic of a waterfall model is “Big Design Up Front” (BDUF). But in the real world changes are frequent especially with business requirements. Waterfall is a simple and well-disciplined model, but it is unrealistic for nontrivial projects.

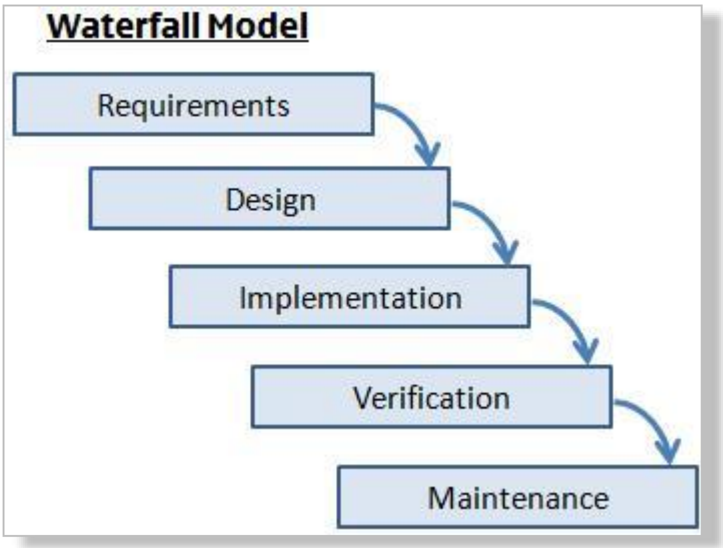


Figure 6: Waterfall Model (Esposito et al., 2010:26)

2.6 New software development methodologies

Modern software development approaches also follow the SDLC cycle. But agile methodologies emulate the incremental process of software development and the same sequence are followed iteratively. Agile software development methodology is a group of software development methods based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. The benefit of agile software development methodology is that it promotes adaptive planning, evolutionary development, and delivery, a time-boxed iterative approach, and encourages rapid and flexible response to change. It is a conceptual framework that promotes foreseen interactions throughout the development cycle.

According to Ambler (2005) Agile is an iterative and incremental (evolutionary) approach to software development which is performed in a highly collaborative manner by self-organizing teams with “just enough” ceremony that produces high quality software in a cost effective and timely manner which meets the changing needs of its stakeholders. This methodology gives priority to incremental software development methods, which is called iteration in agile software development method.

Agile methodologies iterative development is a cyclic process that was developed in response to the waterfall method, and it emphasizes the incremental building of the software (Esposito et al., 2010:27).

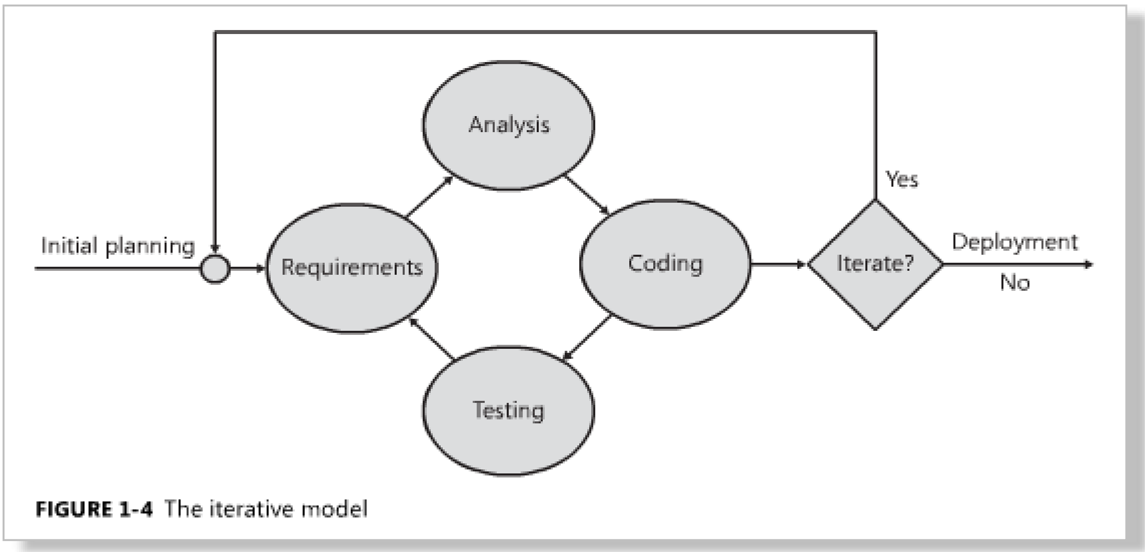


Figure 7: The iterative process (Esposito et al., 2010:27)

Scrum is a framework of agile software development where work is confined to a regular repeatable work cycles also known as a sprint or iteration as show in figure 7.

2.7 Extreme Programming

Extreme programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. Extreme programming advocates frequent "releases" in short development cycles (time boxing) which is intended to improve productivity and introduce checkpoints where new customer requirements can be adopted.

Extreme Programming (XP) is a core agile method that focuses primarily on construction-oriented practices and forces discipline on the practitioners (Ambler et al., 2012:38). Figure 8 reflects the XP practices that are popular and widely adopted by the software developers.

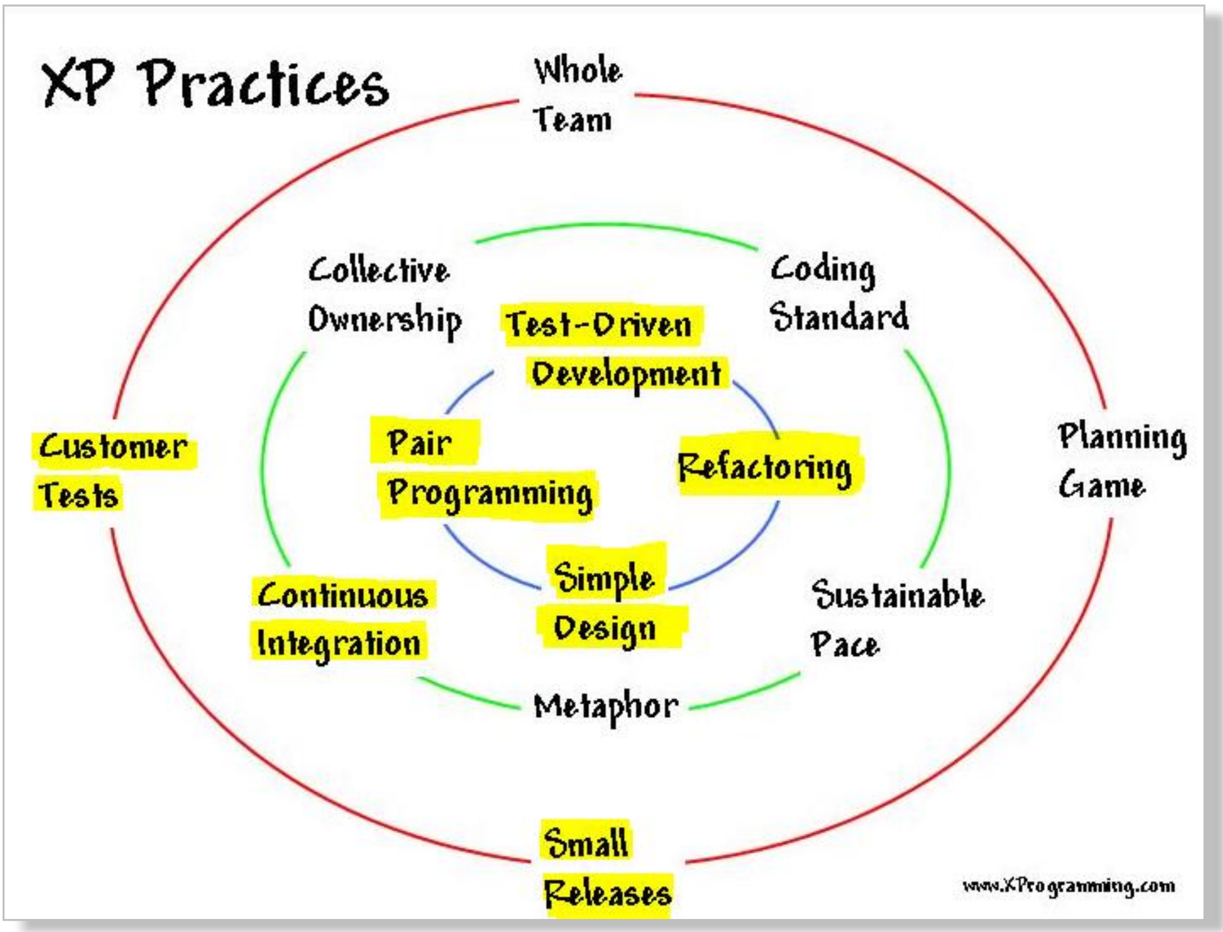


Figure 8: Extreme programming practices (Jeffries, 2001).

From the above extreme programming practices, the practices that are of interest for this research are as follows (Ambler et al., 2012:38-39):

1. ***Continuous integration*** - Team members should check in changes to their code on a frequent basis, integrating the system to ensure that their changes work. The integration effort should validate whether the system still works via automated regression tests.
2. ***Customer tests*** (acceptance test-driven development) - Detailed requirements are captured on a just in-time (JIT) basis in the form of acceptance tests.
3. ***Refactoring*** **-** A refactoring is a small change to the code to improve its design and thereby make it easier to understand and to modify. The act of refactoring enables you to evolve your work slowly over time, to take an iterative and incremental approach to development.
4. ***Pair programming*** (non-solo work) - Pair programming is a practice where two programmers work together on the same artefact synchronously. One programmer

types the code, while the other programmer looks at the bigger picture and provides real-time code review. Non-solo work provides opportunities for people to learn from one another, leads to higher quality as defects are often spotted as they are being injected into the work, and spreads knowledge of the work artefacts across a greater number of people.

5. ***Simple design*** - Programmers should seek the simplest way to write their code while still implementing the appropriate functionality. This increases productivity. Regular periodic refactoring may be required to ensure that the design remains simple and of high quality.
6. ***Small releases*** - Frequent deployment of working software into production is encouraged. This functionality should be delivered in increments that provide the greatest value to the customer. Frequent deployments build confidence in the team and trust from the customer.
7. ***Test-driven development* -** In TDD the first step is to quickly code a new test, basically just enough code for the test to fail. This could be either a high-level acceptance test or a more detailed developer test. You then update your functional code to make it pass the new test. Once the tests pass the next step is to refactor any duplication out of your design. When you write a test before you write the code to fulfil that test, and the test not only validates at a confirmatory level that your code works as it is expected it also effectively specifies your work in detail on a just-in-time (JIT) basis.

Extreme programming promotes testing of code as soon as possible. The best way to do that is to take an idea through analysis, design, code, and have it tested in the shortest possible time (Chromatic, 2003:17). Extreme Programming uses the process as exemplified in Figure 9. The software development process progresses incrementally, and upon testing, one such iteration is completed.

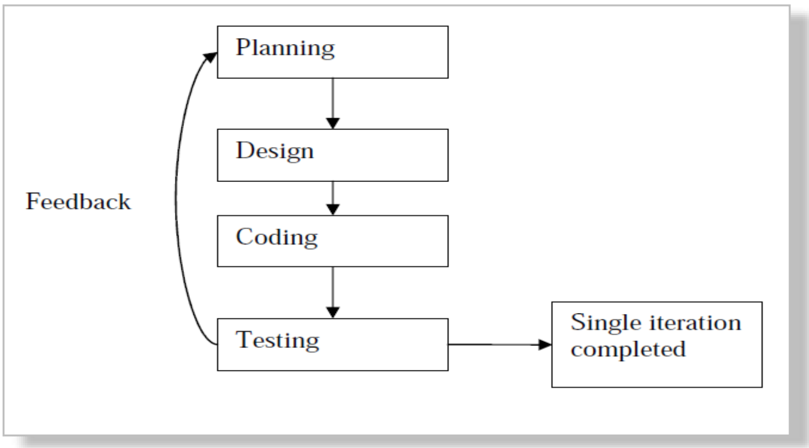


Figure 9: Feedback Loop in Iterative Process of Extreme Programming (Bhadauria, 2009: 11)

2.8 Test Driven Development (TDD)

Sterling et al. (2010:68) state that Test-Driven Development (TDD) is a disciplined practice in which a team member writes a failing test, writes the code that makes the test pass, and then refractors the code to an acceptable design. Effective use of TDD has been shown to reduce defects and increase confidence in the quality of code. This increase in confidence enables teams to make necessary changes faster, thus accelerating feature implementation throughput. TDD is about creating a supportable structure for imminent change.

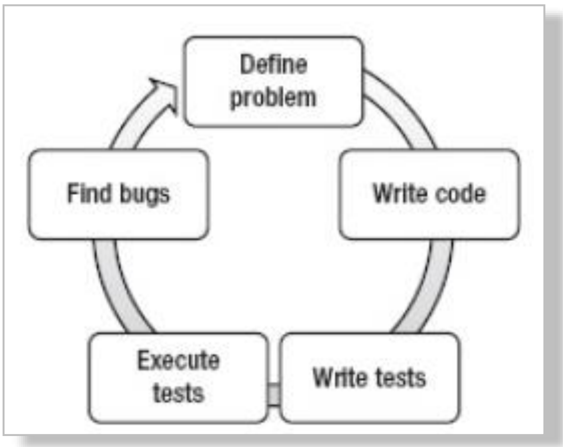


Figure 10: A non-TDD work flow
(Alexander et al., 2011:203)

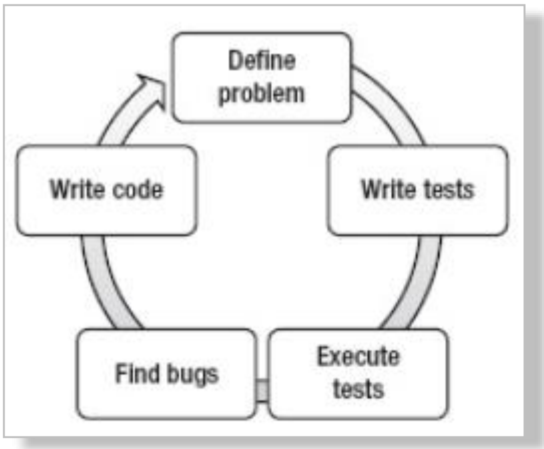


Figure 11: A TDD work flow
(Alexander et al., 2011:203)

An Alexander et al. (2011:202) point out that TDD is a development methodology that places primary focus on testing. In most development workflows or methodologies, testing is used as a means to measure the quality of code during, but usually after, development as shown in figure 10. TDD goes a step further than the traditional methodologies by using tests to help design and plan your codebase before you start writing code as illustrated in figure 11.

Sterling (2010:68) points out that TDD is about creating a supportable structure for imminent change, and this is achieved by writing tests to drive the software design in an executable fashion.

In all the software development process models, one common theme is that since the testing phase is always preceded by the coding phase, the developers are not able to get a complete feedback on their codes until at least a single iteration is not completed. Figure 12 shows the feedback loop in the case of TDD process.

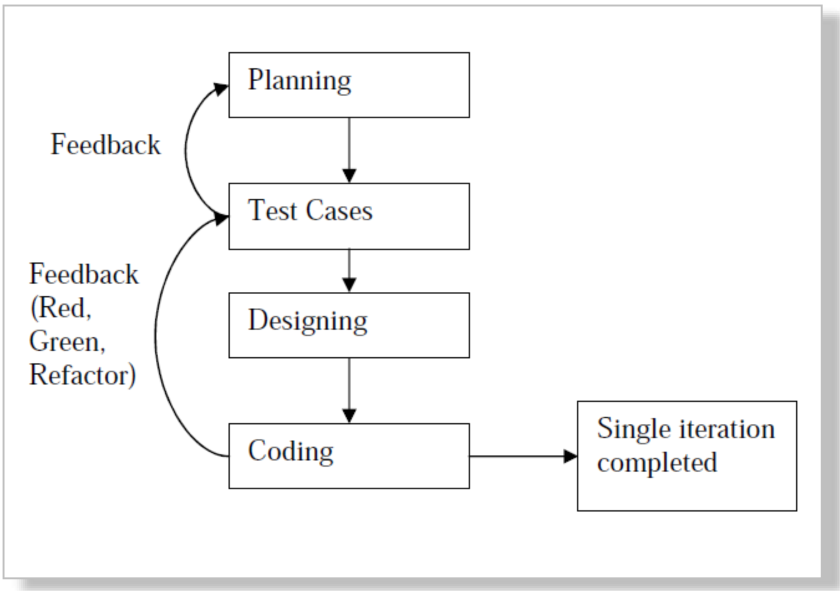


Figure 12: Multiple Feedback loops in the TDD Process (Bhadauria, 2009: 12)

Garofalo (2011:209) points out that each cycle consists of a test case that defines the code requirements and the code that will satisfy these requirements.

Sterling et al. (2010:68) state that TDD is defined by three simple laws as shown in figure 13.

1. You must write a failing unit test before you write production code.
2. You must stop writing that unit test as soon as it fails; and not compiling is failing.
3. You must stop writing production code as soon as the currently failing test passes.

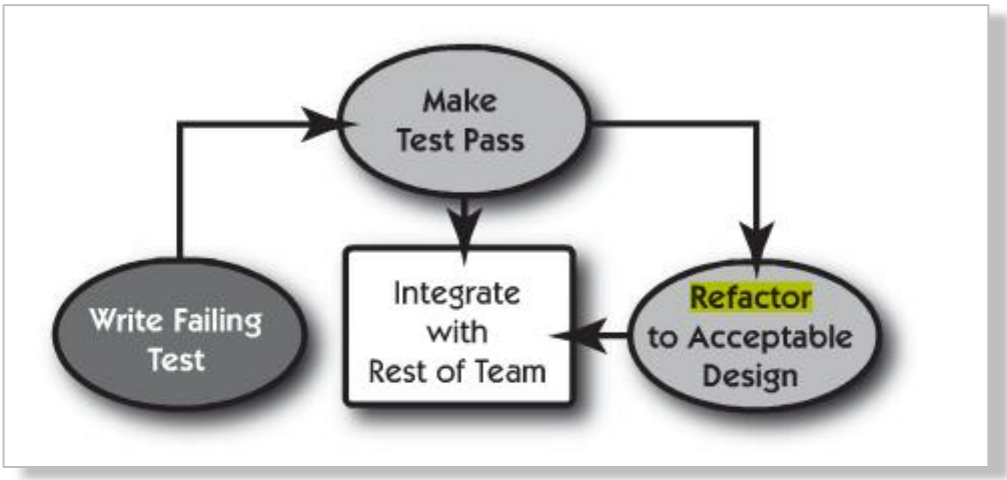


Figure 13: The three basic steps of TDD (Sterling, 2010:69)

TDD follows the Red-Green-Refactor cycle of coding. This simple phrase describes the basic steps of TDD. Garofalo (2011:210) argues that TDD is also known as red-green-refactor because the lifecycle of a test includes three simple steps:

1. Write the test and the assertions and fail them (red).
2. Implement the code to pass the assertions (green).
3. Refactor the code and test it again.

TDD forces the developer to develop according to the requirement and to refine the code to respect and honour the application requirements. TDD forces the developer to write code that is loosely coupled, as the code must be quick and easy to test. TDD allows testing the code written in an incremental process without having a graphical layer to execute the core code. TDD compels to write a lot of single tests in order to cover at least 80-85% of the code with tests, and this approach helps make the coder and other developers confident about the quality of the code and the number of defects covered by tests. Figure 14 shows the basic steps involved in the TDD approach.

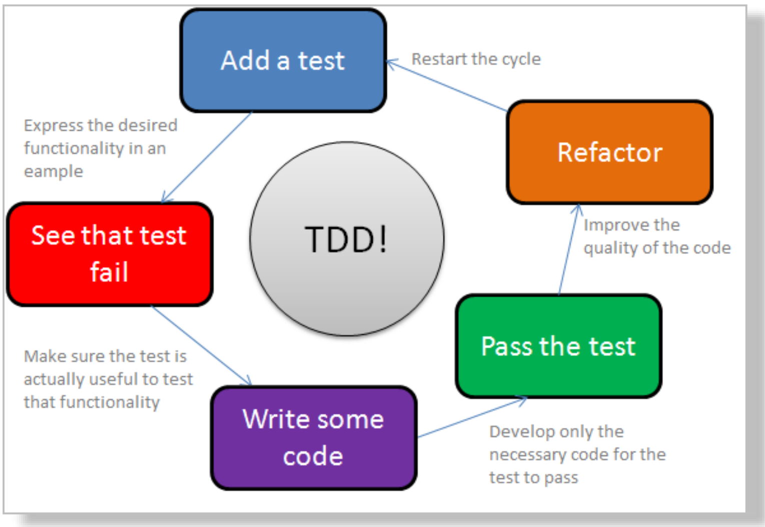

Add a test
Restart the cycle
Express the desired functionality in an eample
Refactor
TDD!
See that test fail
Improve the quality of the code
Pass the test
Make sure the test is actually useful to test that functionality
Write some code
Develop only the necessary code for the test to pass

Figure 14: Red, Green, Reactor cycle in TDD (Maliandi, 2010)

Sterling et al. (2010:70) argues that following the three simple laws of TDD benefit the developer with the following:

1. Reduce your debug time dramatically.
2. Significantly increase the flexibility of your system, allowing you to keep it clean.
3. Create a suite of documents that fully describe the low level behaviour of the system.
4. Create a system design that has extremely low coupling.

Oram et al. (2011:208) argues that writing test cases before production code requires the programmer to consider the design of the solution, how information will flow, and the possible outputs of the code, and exceptional scenarios that might occur before writing the production code. Running the newly written test case before writing production code helps to verify that the test is written correctly and that the system compiles. The TDD also forces the developer in writing just enough production code to pass the test case, which encourages an uncluttered, modular design. TDD users create a growing library of automated test cases that can be executed at any time to verify the correctness of the existing system whenever changes are made. (Oram et al., 2011:208).

2.9 Benefits of Test-Driven Development

Hilton (2009:7) in his research claims that TDD offers many benefits to software engineers. These include:

1. ***Predictability***: TDD allows software developers to know when they are finished because they have written tests to cover all of the aspects of a feature, and all of those tests pass.
2. ***Learning***: TDD gives software developers a chance to learn more about code. Hilton argues that if you only slap together the first thing you think of, then you never have time to think of a second, better thing".
3. ***Reliability***: One of the greatest advantages of TDD is having a suite of regression tests covering all aspects of the system. Engineers can modify the program and be notified immediately if they have accidentally modified functionality.
4. ***Speed***: TDD helps develop code quickly; since developers spend very little time debugging and they find mistakes sooner.
5. ***Confidence***: TDD greatest strengths is that no code goes into production without tests associated with it, so an organization whose engineers are using Test-Driven Development can be confident that all of the code they release behaves as expected.
6. ***Scope Limiting:*** TDD helps teams avoid scope creep Scope creep is the tendency for developers to write extra code or functionality just in case," even if it isn't required by customers. Because adding the functionality requires writing a test, it forces developers to reconsider whether the functionality is really needed.
7. ***Manageability:*** Human beings can only focus on about seven concepts at a time, and using TDD forces programmers to focus on only a few pieces of functionality at a time. Splitting the big system into smaller pieces via tests makes it easier for software developers to manage complex systems in their heads.
8. ***Documentation:*** TDD creates software developers documentation automatically. Each unit test acts as a part of documentation about the appropriate usage of a class. Tests can be referred to by programmers to understand how a system is supposed to behave, and what its responsibilities are.

While these advantages are substantial, Beck (2003:8) summarizes the greatest benefit of TDD as “clean code that works."

2.10 Key attributes of Test Driven Development

TDD is a technique for developing software that uses the writing of tests to guide the design of the target software. There are four key characteristics of TDD suggested in literature by Janzen et al. (2005:43) and Erdogmus et al. (2005:226) that are:

1. Small unit focus.
2. Test-orientation.
3. Frequent regression testing
4. Automated testing.

2.10.1 Small Unit Focus

Developers focus on detailed features in development is a specific characteristic of TDD. TDD is described as developing software in very short iterations with little upfront design. As noted by Maximilien et al. (2003:2), new functionality is not considered properly implemented unless the new unit tests cases and every other unit test cases written for the code base runs correctly.

Take the example of a user story that has two different aspects of functionality, A and B. By comparing the two working processes illustrated in Figure 15, in a traditional test-last development process, in order to implement the whole story a developer firstly writes the production code for both functionalities. Then they write tests to validate the whole story, and finally rework the production code until all tests pass. But in TDD, a developer firstly writes tests for specifying and validating the functionality A and then produces the code to make the tests pass. This is reworked until functionality A is fully completed. The work cycle is repeated for the implementation of functionality B. The testing of functionality A is repeated every time functionality B is refined and tested. The whole story is completed at the end of the second cycle. Therefore, one difference between TDD and the traditional test-last development approach is that developers are focusing on a small unit for design and implementation when using TDD.

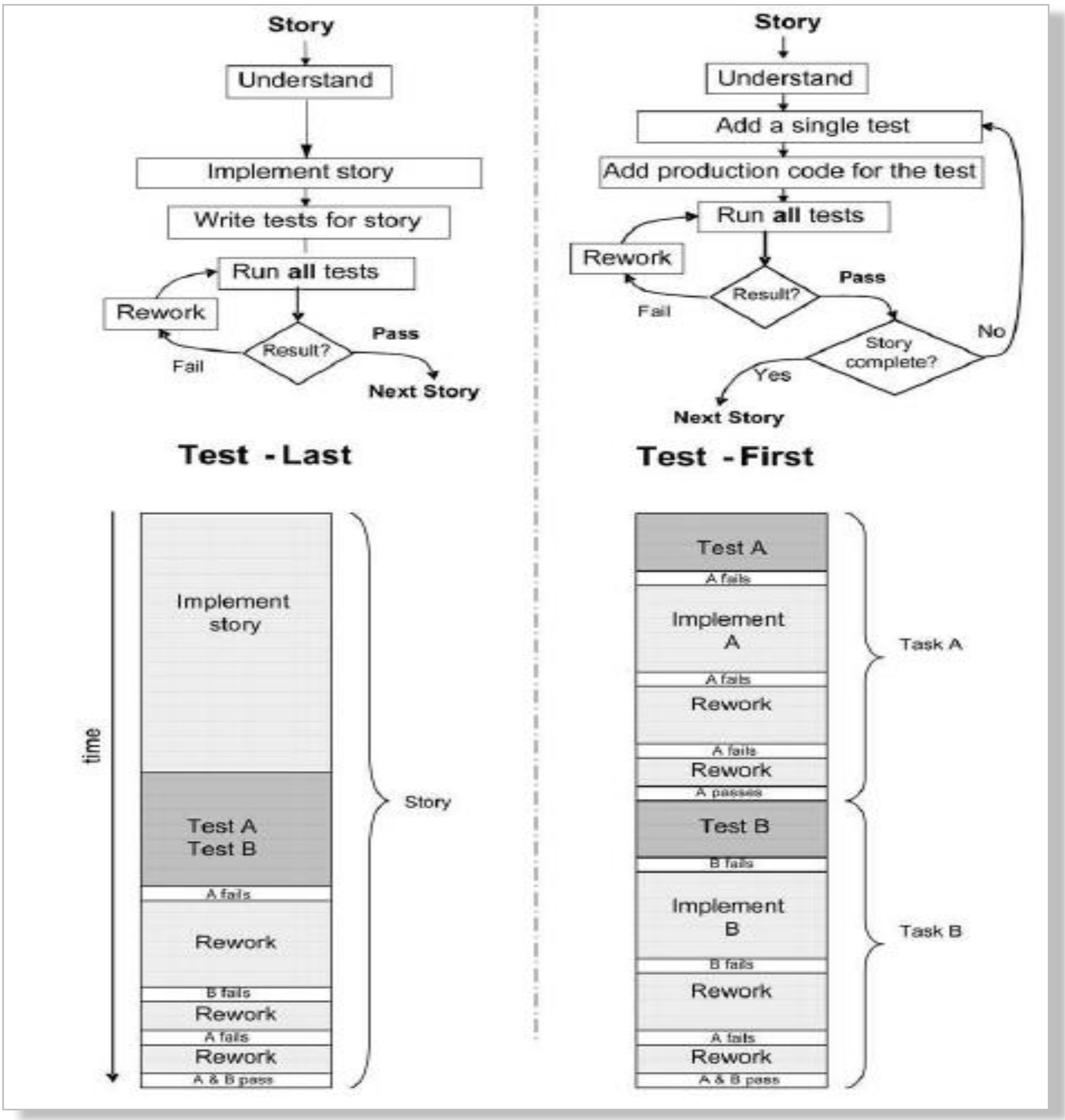


Figure 15: Working Process of Traditional Test-Last Development vs. TDD (Erdogmus et al., 2005:4)

2.10.2 Test-orientation

TDD is also known as test-first programming, so the tests drive the coding activity. Erdogmus et al. (2005:1) refer to this as “task-orientation”. It is one of the most significant characteristics of TDD. In traditional test-last development process, tests are generally built after the target product feature exists and are for verification and validation purposes. Whereas, in TDD, developers write tests that generally break the system for each small piece of functionality, before starting coding the functionality. As Lui et al. (2007:122) emphasize, these tests are used as the specification for the functionality in addition to verification and validation.

Janzen et al. (2005:15) state that TDD given by the agile alliance, developers should produce exactly as much code as will enable that test to pass. Therefore, tests in TDD also are used to define the scope of production code under development, since code that is not needed to pass the test is out of scope. There are three special features of the TDD approach represented within this characteristic:

1. Making breakpoint tests
2. Using tests to specify functionality and scope and
3. Using tests to ***Drive*** the production of code and design.

2.10.3 Frequent Regression Testing

Devised from the theory and principles of TDD, the development process comprises the following steps (Liu et al., 2007:133):

1. Add a unit test for functionality, in the selected user story.
2. Run all tests including the new added one, and see the new test is failed as the code has not yet been implemented.
3. Write code to pass the new test.
4. Run all tests and see them all succeed.
5. Refactor the code if necessary.
6. Run all tests and see them all succeed.

Repeat step 1-6 for another functionality. This shows that regression testing is an integral part of TDD. All of the test cases will be run before and after new code is added to the code base and the process is iterated when each piece of functionality is implemented. If the regression test fails, even if the new functional test passed, this means that the new functionality has introduced a defect that affects the previously written code. Therefore, frequent regression testing is one of key characteristics in TDD.

2.10.4 Automated Tests

Huttermann (2011:26) defines automation as the use of solutions to reduce the need for human work. Automation can ensure that the software is built the same way each time, that the team sees every change made to the software, and that the software is tested and reviewed in the same way every day so that no defects slip through or are introduced through human error. There is an important rule in TDD that: "Have all tests run at 100% all the time".

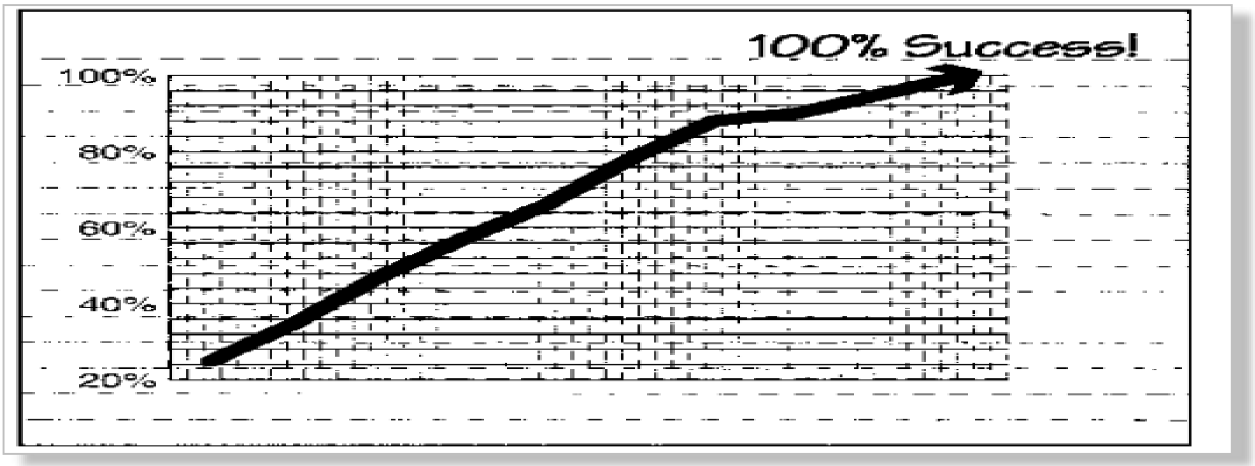


Figure 16: Test running at 100% time (Jeffries et al., 2000:60)

Consequently, automated tests are indispensable in TDD. Also sets of automated tests are useful for regression testing. There are automated unit testing frameworks developed to simplify both the creation and execution of software unit tests. Janzen et al. (2005) suggest that they have played an important role in the emergence of TDD as they were argued to aid to improve and increase developer productivity. Erich Gamma and Kent Beck developed JUnit, an automated unit testing framework for Java, essential for implementing TDD with Java. JUnit-like frameworks have been implemented for several different languages, creating a family of frameworks referred to as xUnit.

2.11 Software Development Skills in TDD

Skill is defined as "the ability to do something well" (Oxford Dictionaries, 2012). The definition in business dictionaries states skill as "an ability and capacity acquired through deliberate, systematic, and sustained effort to smoothly and adaptively carryout complex activities or job functions involving ideas i.e. cognitive skills, technical skills, and/or people i.e. interpersonal skills (Business Dictionaries, 2012).

Bhadauria (2009: 12) points out that software development requires two types of skill sets. One is the technical skill set (computer related) and the other set is of functional skills (task related). Technical skills are more grounded in logic and algorithm while functional skills are more oriented towards abstract conceptual thinking with focus on real world business applications.

The skills required for the design and development phase are different from those required at the testing phase. While the priority in the design and development phase is on the technical skills, in testing more emphasis is on the task related skills. It is argued that the role of software tester is much closer to the end user, in the sense the tester has to look at the code from an end user's perspective with the view to find possible scenarios when the code could fail. Hence, both phases are important in the systems development process and they emphasize different skill sets. Consequently, in a typical traditional setting, the coder and the tester are two different people. In TDD the skills of the coder and the tester are more closely related.

2.12 Acclaimed Benefits of TDD

A number of benefits are claimed and evidenced in literature regarding the use of TDD in software development, as compared to a traditional Test-Last approach. These can be categorized as the following high level benefits: simpler software design, improved software code quality, improved software application quality, and enhanced software developer productivity.

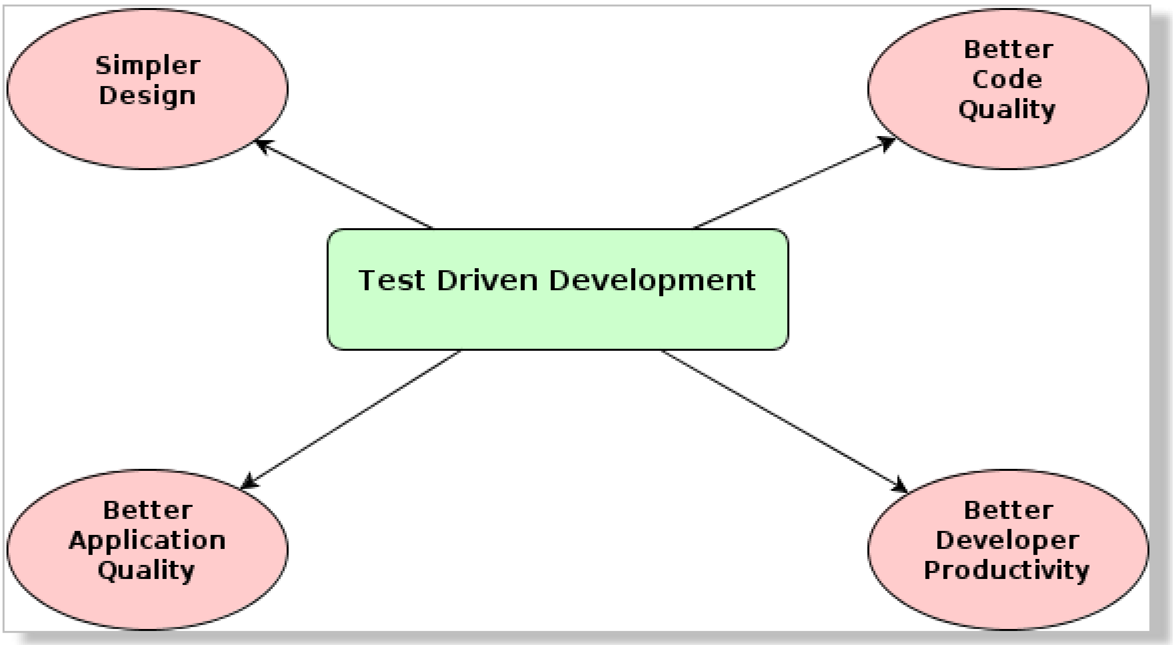


Figure 17: Acclaimed Benefits of TDD (Source: Self-generated)

2.12.1 Simpler 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. Having a full coverage of programmer tests makes it possible to refactor the code or introducing new requirements.

Roodyn (2005:65) points out that refactoring is changing the structure of existing code without changing the behaviour of that code. Refactoring is changing the structure of existing code without changing the behaviour of that code. Oram (2010:209) state that though TDD is primarily interpreted as a development practice, it is considered as a design practice as well. An incremental and simple design is expected to emerge when using the TDD principles. The simple design is driven by the modularity needed to make code testable, by writing minimal production code to complete simple tasks, and by constant refactoring.

Beck (2003:141) argues that by coding only what you need for the tests and removing all duplication, you automatically get a design that is perfectly adapted to the current requirements and equally prepared for all future stories. The mind-set that you are looking for just enough design to have the perfect architecture for the current system also makes writing the tests easier.

Test driven development also helps designing modular and well decomposed software. In order for a piece of code to be easily tested in isolation, its dependencies to other modules and functions must be minimized. This drives the developer to write more decoupled classes and modules.

Koskela (2008:14) points out that test-driven development is a development and design technique that helps us build up the system incrementally, knowing that we're never far from a working baseline. And a test is our way of taking that next small step. Test-driven development or TDD, is a programming technique based on a very simple rule: Only ever write code to fix a failing test. In other words, write the test first and only then write the code that makes it pass. This rule is controversial to many of us who have been schooled to first produce a thorough design, then implement the design, and finally test our software in order to find all those bugs we've injected during implementation. TDD turns this cycle around, as illustrated in figure 18. By definition, the code we write must be testable otherwise it should not exist. The design of software is not just about structure. It's also about the suitability of the software for the current needs.

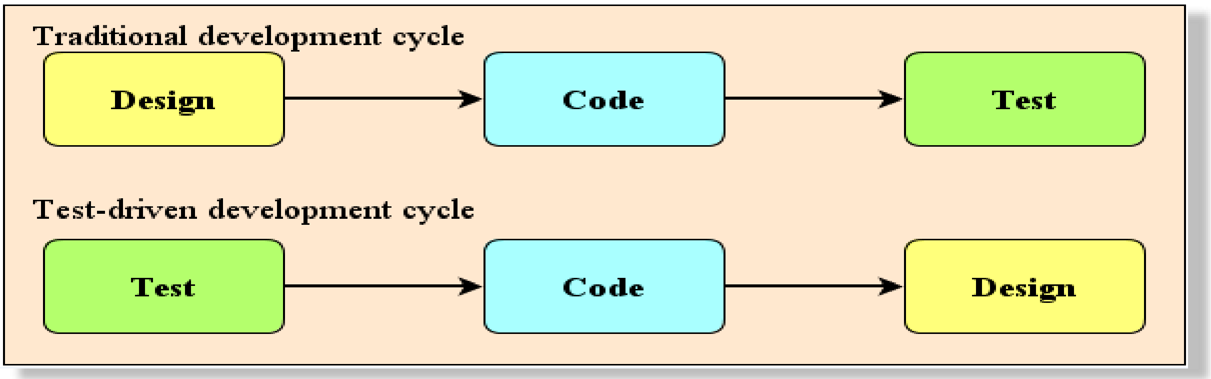


Figure 18: TDD turns around the traditional design-code-test sequence (Koskela, 2008:15)

Koskela (2008:18) points out one way tests drive the design in TDD is that they tell you exactly what your software needs to be able to do now. Not tomorrow, not yesterday but now. Proceeding in these small steps, implementing just enough functionality to gets that next test passing and so we are in control of our software and its design and prevent over engineering.

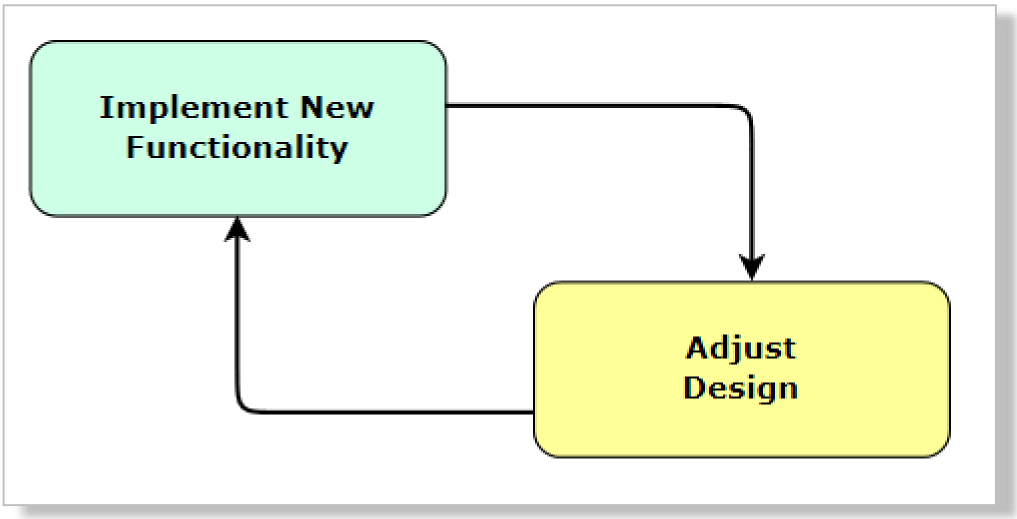


Figure 19: Incremental design (Koskela, 2008:21)

Figure 19 shows how this iterative, incremental process moves back and forth between the small step of adding functionality and adjusting our design and architecture to properly accommodate that added functionality. Incremental design leads to the code that exhibits the best design the developers could conceive for supporting the present functionality. This way, we evolve an empirically proven architecture. Koskela (2008:18) also argues that instead of thoroughly analysing all the possible scenarios imaginable before finalizing the design, we opt for making our design decisions based on knowledge not assumptions acquired during implementation. In this way the architecture is rock solid and supports all the features the product will have. TDD by its nature encourages such small steps with evolutionary design that promotes constant improvement, changing the design in small increments.

Evolutionary design as shown in figure 20, is a mind-set that requires us to make that change nurture the living and growing code base instead of protecting it from the bad, bad world that wants change and thus improve the quality of our design and, indirectly, the quality of the whole system (Koskela, 2008:22).

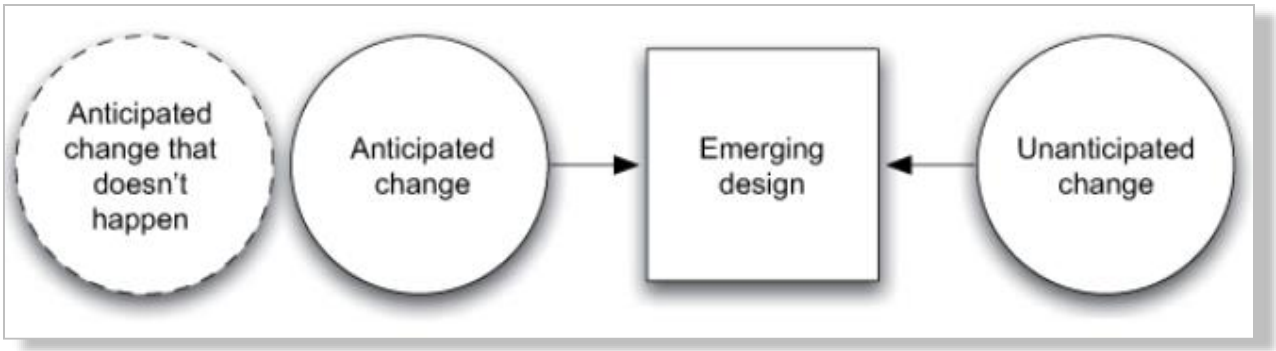


Figure 20: The emerging, evolutionary design (Koskela, 2008:22)

The emerging, evolutionary design of a system is influenced by both anticipated and unanticipated change. A significant amount of anticipated change doesn't happen or happens in a different form than expected, essentially making it unanticipated change. Our job is to apply common sense and good judgment when preparing for change that we know will happen.

2.12.2 Better Software Quality

Literature claims that TDD improves the code quality by providing simpler design and cleaner code. Gunvaldson et al. (2007:158) states that the quality of the software improves constantly due to constant improvement of code, continuous integration, and constant refactoring provided by TDD. This also closely resembles Deming's continuous improvement cycle that advocates Plan, Do, Check, and Act. The research by Janzen et al. (2008:77) also demonstrated that TDD is better in producing simple and loosely coupled. Hilton (2009:54) experiments revealed TDD improved the level of cohesion by about 21%. The research conducted by Janzen et al (2008:77) included three quasi-controlled experiments and one case study involving 12 undergraduate and graduate software engineering students at the University of Kansas to compare the TDD approach with the traditional Test-Last approach. The experiments interwove the approaches and mixed up their order in completing individual projects.

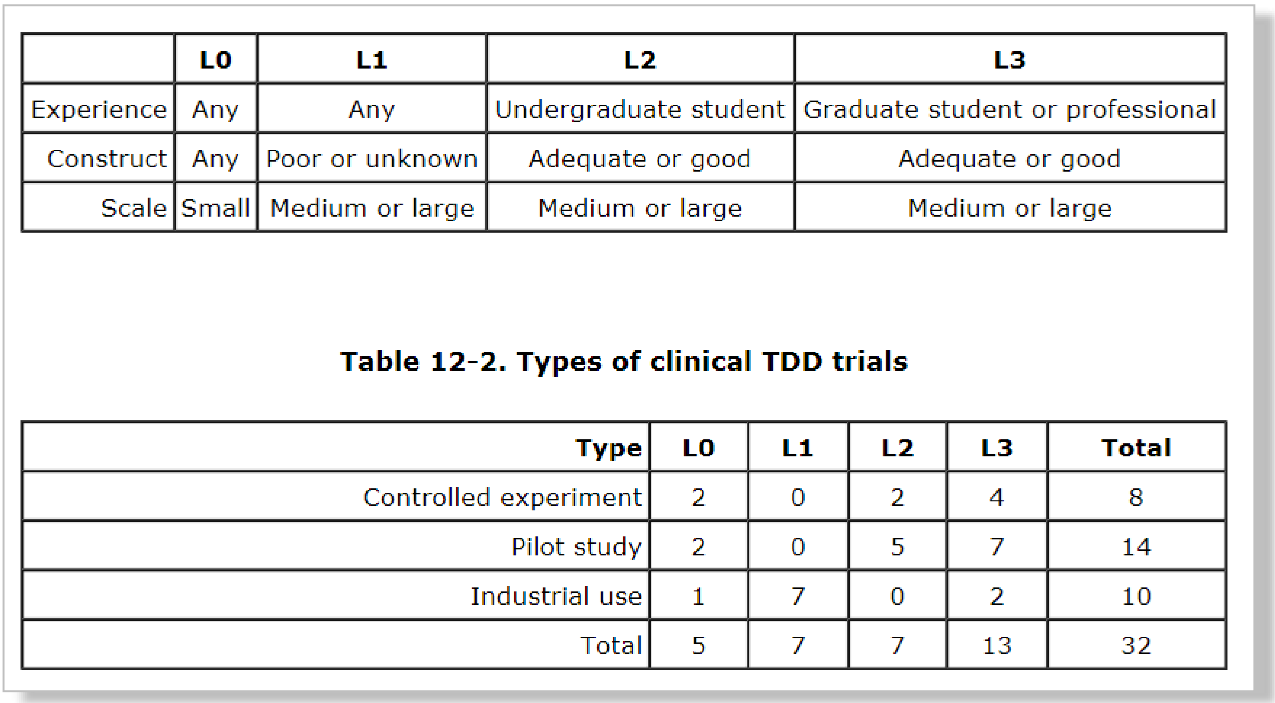
The first experiment involved a test-last project with no automated tests, followed by a second phase of the same project completed with a TDD approach. The second quasi-experiment involved a Test-Last project followed by a TDD project. The third quasi-experiment involved a TDD project followed by a Test-Last project. The case study interviewed the developers from 15 software projects completed in one development group over five years. The 15 projects included the five TDD and Test-Last projects from the industry quasi-21 experiments, and 10 used a test last approach. The results showed that the TDD had smaller modules and fewer methods per class than Test-Last; TDD had lower complexity and better cohesion.

Tahchiev et al. (2003:96) claim that the use of TDD is effective at producing cleaner code and also that it “eliminates duplication of code”. No specific empirical evidence is presented in the book to support this. Crispin (2006:70) does provide some supporting evidence for cleaner code with TDD in software development projects.

Two researches from Microsoft, Bhat and Nachiappan conducted a case study in 2006 attempted to evaluate the effectiveness of Test-Driven development in a corporate professional environment. Bhat et al. (2006) observed two different teams at Microsoft primarily on defect reduction and analyzed two case studies as part of their research. Bhat and Nachiappan picked two projects at Microsoft, one that was part of Windows and one that was part of MSN. The two teams tried TDD for a short period of time and the two researchers looked at the bug tracking system to determine if TDD helped reduce the number of defects that were found in the code. They also kept track of how long it took the for the software engineers to complete their work. For each of these projects, Bhat and Nagappan found a similar project of similar size and used the same bug tracking system to determine how many defects were found at the completion of that effort. The two projects were then compared.

The Microsoft researchers found substantial improvement in quality when test driven development was used. In the first case study, the non-TDD group produced 2.6 times as many defects as the TDD group. In the second case study, the non-TDD group produced 4.2 times as many defects. The managers of the groups also provided estimates for how much adopting TDD slowed the teams down. In the first case study, the manager estimated that the TDD group took 25-35% longer because of TDD, while the manager from the second case study estimated 15% causing development time to increase.

Oram (2010:209) in 2009 analysed 22 reports containing 32 unique trials extracted key information from the reports regarding study design, study context, participants, treatments and controls, and study a results. This report revealed that the effect on internal code quality and external code quality and it appears to yield better results over the control group for certain types of metrics (complexity and reuse). Participants in these trials had different experience levels, ranging from undergraduate students to graduate students to professionals. The number of participants per trial ranged from a single individual to 132 persons. The effort spent on the trials spans a wide interval, ranging from a few person-hours to 21,600 person-hours. Each trial compares the effects of the TDD with respect to test-last development. The subjects in the treatment groups were comprised of various units, such as individuals, pairs, teams, and projects. Figure 1 shows the levels of the clinical trials.



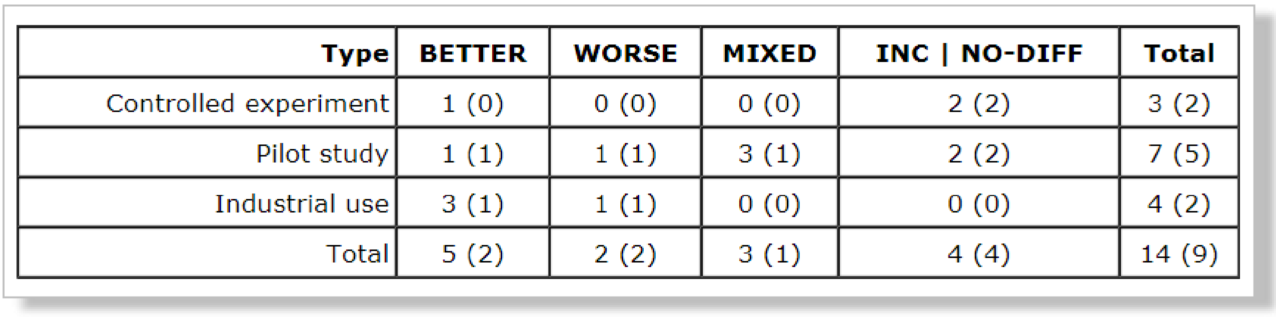
	L0	L1	L2	L3
Experience	Any	Any	Undergraduate student	Graduate student or professional
Construct	Any	Poor or unknown	Adequate or good	Adequate or good
Scale	Small	Medium or large	Medium or large	Medium or large

Table 12-2. Types of clinical TDD trials

Type	L0	L1	L2	L3	Total
Controlled experiment	2	0	2	4	8
Pilot study	2	0	5	7	14
Industrial use	1	7	0	2	10
Total	5	7	7	13	32

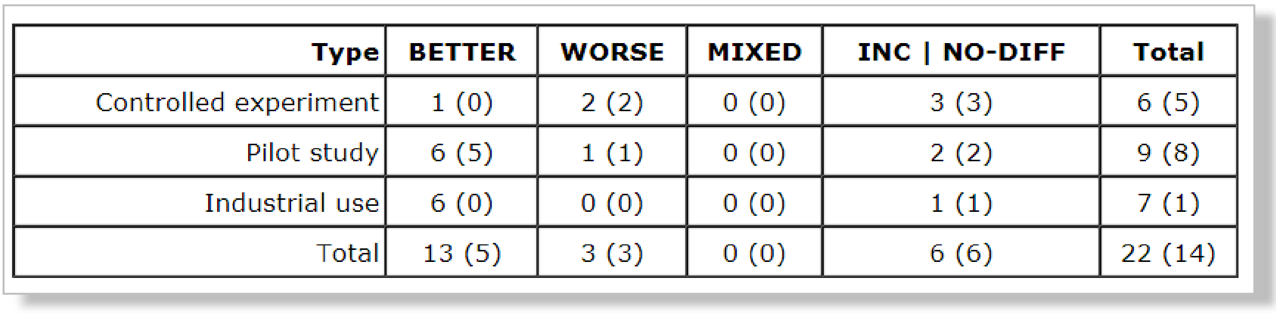
Table 1: Levels of clinical TDD trials (Oram, 2010:211)

The table 2 and 3 shows the summary value of better, worse, mixed, or inconclusive/no-difference for internal and external quality.



Type	BETTER	WORSE	MIXED	INC | NO-DIFF	Total
Controlled experiment	1 (0)	0 (0)	0 (0)	2 (2)	3 (2)
Pilot study	1 (1)	1 (1)	3 (1)	2 (2)	7 (5)
Industrial use	3 (1)	1 (1)	0 (0)	0 (0)	4 (2)
Total	5 (2)	2 (2)	3 (1)	4 (4)	14 (9)

Table 2: Effects on internal quality (Oram, 2010:212)



Type	BETTER	WORSE	MIXED	INC | NO-DIFF	Total
Controlled experiment	1 (0)	2 (2)	0 (0)	3 (3)	6 (5)
Pilot study	6 (5)	1 (1)	0 (0)	2 (2)	9 (8)
Industrial use	6 (0)	0 (0)	0 (0)	1 (1)	7 (1)
Total	13 (5)	3 (3)	0 (0)	6 (6)	22 (14)

Table 3: Effects on external quality (Oram, 2010:213)

Therefore, the internal and external quality associated with TDD seems at least not worse and often better than alternative approaches.

2.12.3 Software Application Quality

Jones et al. (2011:9) considers the following 10 most important attributes for defining the software quality.

1. Elegance or beauty in the eye of the beholder.
2. Fitness of use for various purposes.
3. Satisfaction of user requirements, both explicit and implicit.
4. Freedom from defects, perhaps to Six Sigma levels.
5. High efficiency of defect removal activities.
6. High reliability when operating.
7. Ease of learning and ease of use.
8. Clarity of user guides and HELP materials.
9. Ease of access to customer support.
10. Rapid repairs of reported defects.

Supporters of TDD claim that it enhances the quality of the software application developed using TDD. Common aspects of application quality mentioned in literature are reliability and correctness. Spinellis (2006:150) defines software reliability as its capability to maintain the specified level of performance under the specified conditions (i.e. fewer defects). Both industrial and academic studies provided significant evidence to support the occurrence of these benefits in practice.

Humphrey (2010:6) argues that software quality affects development costs, delivery schedules, and user satisfaction. The quality of a software product must be defined in terms that are meaningful to the product's users. Thus a product that provides the capabilities that are most important to its users is a quality product.

The research results from Crispin (2006:70) indicated that TDD leads to the better requirement understanding and this consequently improves the correctness of the software. Crispin also investigated TDD benefits by investigating TDD practitioner's experiences and perceptions and found that practitioners felt they could understand the requirement better with TDD compared to the traditional test-last development practice. The practitioners also acknowledged that customers were more satisfied with the software developed with TDD because the developers felt they anticipated the features better.

George et al. (2004:341) also conducted a controlled experiment but with practitioners. They showed that TDD produces higher quality software with fewer bugs. The experiment used 8 person groups at three companies who are professional programmers. The programmers were divided evenly into two groups with one using TDD approach and the other using the traditional test-last approach. The authors found that the TDD group produced higher quality code which passed 18% more functional tests. They further stated that such findings may result from TDD encouraging more frequent and tighter verification. They also indicated that the programmers following a Test-Last process often did not write the required tests after completing their code, which might cause inadequate testing and reduce the application reliability. TDD, on the other hand, has the potential of increasing the level of testing, and this may contribute to higher quality software.

Industrial case studies also provide strong evidence that TDD helps with producing more reliable software. It is shown in both those studies that the defect rate is lower in the project developed using TDD compared to the project developed traditionally. In one study Williams et al. (2003:4) carried out a case study in IBM which reported up to a 40% reduction in defect density for the TDD group compared to the other group. In a more recent study of two similar case studies conducted with slightly different context factors conducted at Microsoft by Bhat et al. (2006:293), observed that a similar result that the use of TDD reduces the number of defects. Their results of the two case studies also indicated that more test coverage contributed to lower defect density.

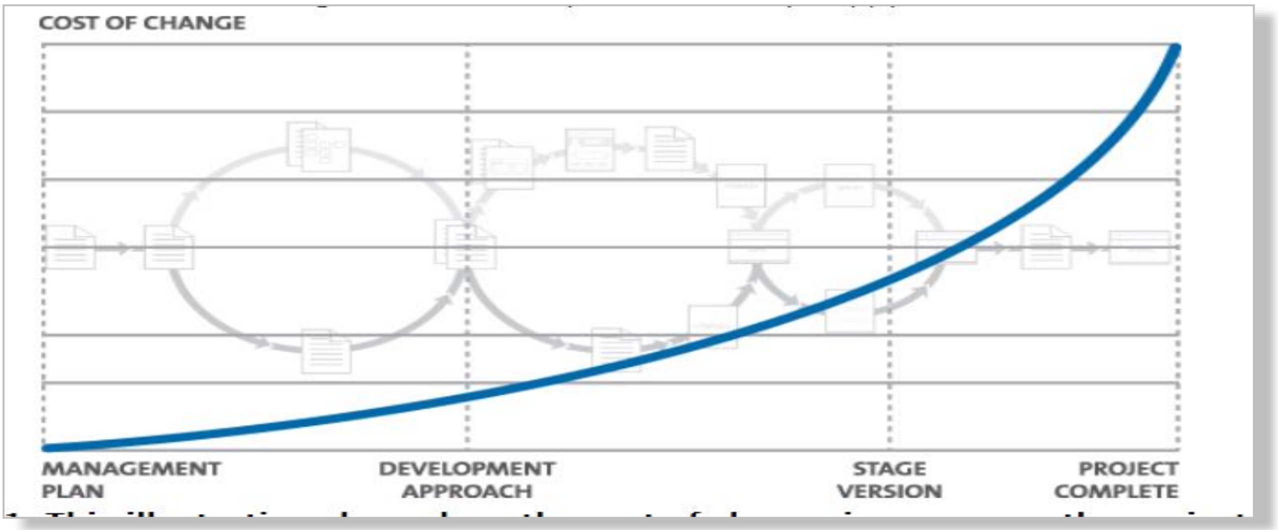


Figure 21: Cost of change in the phases of software development process (Lyons, 2012:109)

Figure 21 presents the five major phases of the waterfall approach. The curve represents Boehm’s "Cost of Change". If we can move the testing phase further to the left we might stand a chance of reducing costs whilst enjoying the natural effect of a longer testing period. TDD helps us to achieve this by embracing agile techniques that sport iterative or incremental development.

Therefore, TDD can improve the application’s quality by introducing testing much earlier into the process. By maintaining a suite of tests that are both repeatable and automated, we can enjoy the luxury of being able to make sweeping changes to our code with the knowledge that any deviations and bugs will be picked up by re-running the test suite. So the application’s quality is checked with each iteration and therefore leading to a high quality of software application.

2.12.4 Software Programmer/Developer Productivity

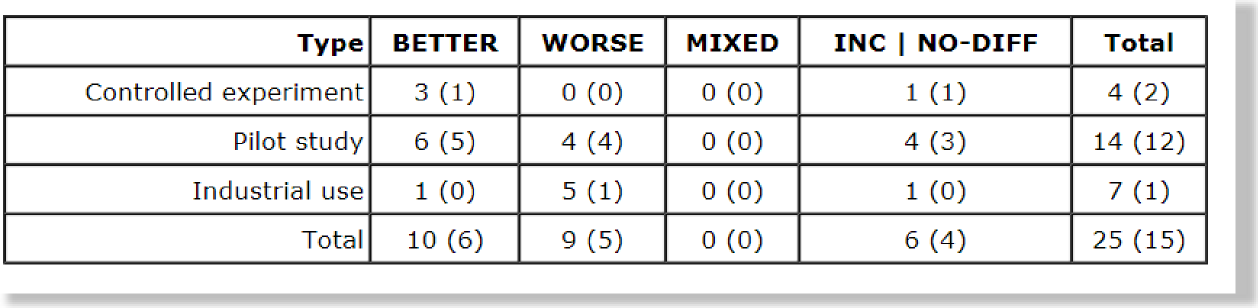
Arthur (1983) defines productivity as "the ability to create a quality software product in a limited period with limited resources." Arthur also asserts that productivity, quality, and software measurement are intimately connected.

TDD has received criticism for increasing the coding time, but supporters claim TDD enhances the overall productivity by reducing the efforts on design, testing, and fixing bugs. A number of studies can be found in the literature to support these claims. Three recent examples of these studies are now discussed.

Erdogmus et al. (2005:11) conducted an experiment to evaluate the use of TDD against the test-last strategy with two groups of undergraduate students. They were given the task of developing a small program using alternate strategies. Both groups followed incremental development and applied unit testing. The researchers found that the TDD group wrote more tests but were more productive in terms of overall development effort, specifically less rework effort at the testing and fixing stages. The research conducted by Janzen et al. (2006:3) was also a controlled experiment with undergraduate students in a software engineering course. Students in three groups completed semester-long programming projects using either an iterative TDD, iterative test-last, or linear test-last approach. Results from this study indicated that TDD programmers were more productive than the test-last programmers since the TDD team spent less effort per line-of-code and 57% less effort per feature than the test-last team.

It should be noted that a number of researchers have provided evidence from empirical studies that the use of TDD increases the time for the coding phase (Bhatt et al., 2006:297). George et al. (2004:342) also indicated that transitioning to the TDD mindset is difficult, and it might be one factor that increases the efforts put in the coding phase with the use of TDD. These studies, however, do not include end-to-end productivity in their investigations and make no claim about overall productivity. The previously discussed studies argue that productivity gains in other development stages (e.g. design, testing and bug fixing) more than compensates for increased coding time.

Oram (2010:213) argues that the productivity dimension produces the most controversial discussion of TDD. Although many admit that adopting TDD may require a steep learning curve that may decrease the productivity initially, there is no consensus on the long-term effects. The evidence from controlled experiments suggests an improvement in productivity when TDD is used as shown in table 4.



Type	BETTER	WORSE	MIXED	INC | NO-DIFF	Total
Controlled experiment	3 (1)	0 (0)	0 (0)	1 (1)	4 (2)
Pilot study	6 (5)	4 (4)	0 (0)	4 (3)	14 (12)
Industrial use	1 (0)	5 (1)	0 (0)	1 (0)	7 (1)
Total	10 (6)	9 (5)	0 (0)	6 (4)	25 (15)

Table 4: Effects on productivity (Oram, 2010:213)

Koskela (2008:14) argues that TDD and acceptance TDD don’t make us type any faster, but they help us cut time from less productive activities such as debugging and cursing at unreadable code, or rework due to misunderstandings regarding requirements.

Bulajic et al. (2012:174) from his experiment found that TDD increases the development time from 16% to 25%. But due to better requirements understanding and reduction in debugging time the overall productivity is high.

Another study conducted by George et al. (2004:337) involved pair programmers working in the industry, they found that though the initial productivity went down, the developers using TDD produced code of a higher external quality that passed 18% more functional black box tests as compared to the codes developed by using the traditional test last approach.

Helper (2009) points out that test driven development (TDD) is well known and practiced methodology that reduces the amount of defects introduced during the coding stage of the development process. Even if a development team has more defects and produces the features in less time, those defects will be caught during QA and integration testing. There is a known fact in software development that the longer it takes to find the bug the more it would cost to fix it. Figure 22 shows the cost to fix the bug at each phases of the software development stage.

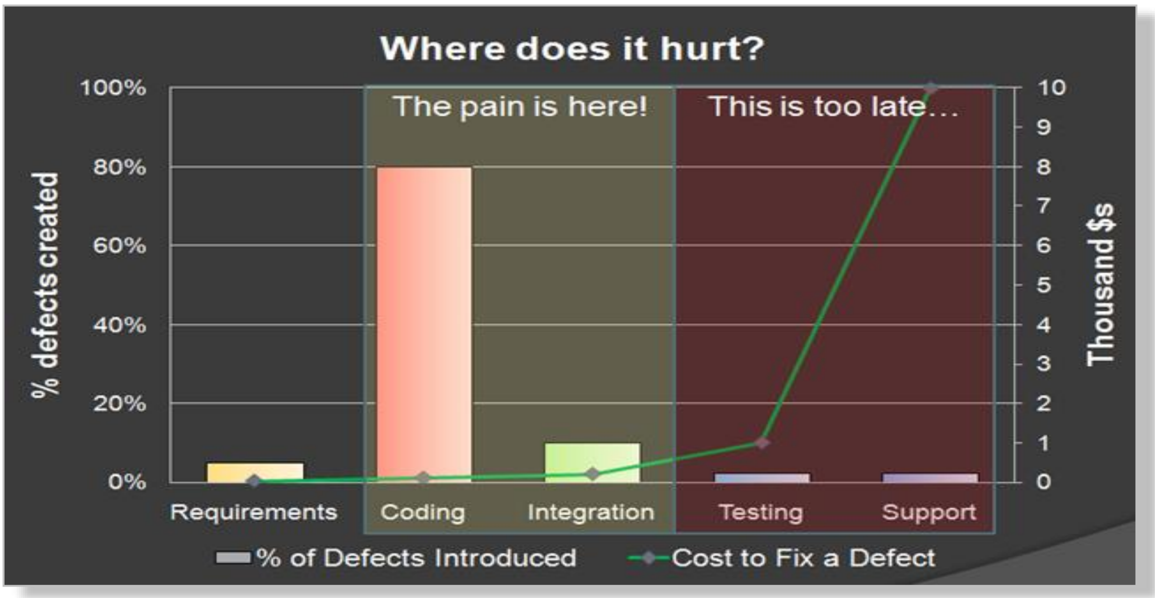


Figure 22: Defect created verses Cost to fix (Helper, 2009)

The research proved three points:

1. Using TDD reduces the amount of bugs in the code significantly.
2. Using TDD takes more time then not using TDD.
3. Using TDD the productivity is more as the defect rate is less during long run.

2.13 Software Developer’s Paradigm shift

Downey (2012:3) defines paradigm shift as a process in the history of science where the basic assumptions of a field change, or where one theory is replaced by another. Researchers claim that the hardest part of TDD is changing the way software developers think about writing the code (Gold, 2005:26). This change translates into a paradigm shift. The role of the programmer is now completely reversed. The software developer now is not thinking in terms of algorithm and structure, but he/she has to conceptualize the situation more from the perspective of the end user. The focus therefore shifts from the algorithmic logic to the functional utility. For most of the programmers coming from the traditional software development this is the hardest part to learn.

2.14 Conclusion

The claimed benefits of using TDD from the literature studies are conceptualized into three high level categories. The meanings for each category of high level benefit, are also identified. Most existing studies describe and verify the benefits, but few of these probe into the reasons or contributing factors of these benefits. This dissertation seeks to confirm these benefits, but also to investigate the factors contributing to the realization of the benefits.

The first key benefit of TDD is ***better software code quality*** which means simple design (Janzen et al., 2008:77), clean code (Crispin, 2006:71), no duplication of code (Tahchiev et al., 2003:99), low coupling of code modules (Janzen et al., 2008:78; Beck, 2001:88), easier to understand-more meaningful (Crispin, 2006:71) and high cohesiveness of code modules (Hilton, 2009:54; Beck, 2001:88). The contributing factor to better software code quality is that the developers have clearer knowledge of the scope of stories or features (Janzen et al., 2005:43; Crispin, 2006:71).

The second key benefit of TDD ***is better software application quality*** which means low defect rates (Bhat et al., 2006:238), high customer satisfaction with applications (Muller et al., 2002:131) and passing of more functional tests (George et al., 2004:340). The contributing factors to better application quality are better understanding of requirements (Crispin, 2006:71), frequent and tighter verification and validation (George et al. 2004:341), potential of increasing the level of testing (George et al., 2004:341) and more tests coverage (Bhat et al., 2006:298).

The third key benefit of TDD is ***better developer productivity***. TDD requires the developer to put in less effort to produce the final product from start to finish (Janzen et al, 2006:78; Erdogmus et al., 2005:11). The contributing factors to better developer productivity are less rework effort during the testing and fixing stages (Erdogmus et al., 2005:11) and savings in big up-front design effort (Gupta et al., 2007:285).

The next chapter discusses the methodology, design, strategy and instruments used for this research.

Chapter 3 - Research Methodology

3.1 Introduction

This chapter discusses the research methodology used in this case study to gather the data and looks at how the questionnaire responses were collected and analyzed. Research is about acquiring knowledge and developing understanding, collecting facts and interpreting them to build up a picture of the world around us (Walliman, 2011:15). Bhattacharya (2006:12) points out that research always begins with a question and a problem and research is a systematic approach towards purposeful information. Krishnaswamy et al. (2006:4) argues that research is a systematic self-disciplined enquiry and its aim is to understand a phenomenon. When an enquiry is aimed at understanding the phenomenon then it is termed as a fundamental research and when it is aimed at applying the available knowledge for practical or commercial use then it is termed as applied research. Management research is primarily applied research that is directed towards aiding managers in their decision making. The topic selected by the researcher is a management research and therefore it is more suited to a qualitative research strategy. As such, the research approach that is explained for this case study is qualitative.

3.2 Rationale for the Methodology

Qualitative approach to research is concerned with subjective assessment of attitudes, opinions, and behaviour and researchers study things in their natural settings. Denscombe (2003:267) describes qualitative research is the study which focuses on meanings and the way people understand things. Qualitative researchers aim to gather an in-depth understanding of human behaviour and the reasons that govern such behaviour and investigates the why and how of decision making. Therefore to deal with the collection of data with the above mentioned attributes, unstructured interview was the best suitable research methodology.

3.3 The Research Design

Research design deals with the defining of procedures that will be adapted to carry out the research study and also tells us if the research is carried out in the field or in the laboratory. The details of the data collection procedures and the analysis procedures to be used in order to accomplish the research objectives are also dealt with in the research design (Krishnaswamy et al., 2006:24).

A case study method was selected as the research design for this study. Burns (2000:460-461) claims that a case study method focuses on a bounded system which is an entity in itself and allows examination in depth and the researcher can probe deeply, undertaking intensive analysis of the subject of the case study examining the various phenomena. The case study method can be used to collect extensive data to produce the understanding of the entity being studied, allowing the investigation to retain holistic and meaningful characteristics of real life events.

Saunders et al. (2007:315,344) state that using non-standardised qualitative research interviews allows the researcher to collect a rich, in-depth and detailed set of data about the selected topic. The three principal ways of conducting the exploratory research are (Saunders et al., 2007:133):

1. A search of the literature.
2. Interviewing experts in the subject.
3. Conducting focus group interviews.

The advantage of this type of study is that it is flexible and adaptable to change your direction as a result of new data that appear and new insights that occur during the study (Saunders et al., 2007:135). Therefore an interpretive qualitative approach with exploratory-unstructured research design was used in this case study.

3.4 The Research Philosophy

Positivist and the phenomenologist research are the two types of views that are held on research philosophy. According to Walliman (2011:21) positivist approach to scientific investigation is based on acceptance as fact, that the world around us is real and that we can find out about these realities. It is not always appropriate for business research as all social phenomenons can’t be adequately measured. Groenewald (2004:5-6) state that in phenomenologist research the aim of the researcher is to describe as accurately as possible the phenomenon, refraining from any pre-given framework, but remaining true to the facts. The phenomenologist researchers are concerned with understanding social and psychological phenomena from the perspectives of people involved. A researcher applying phenomenology is concerned with the lived experiences of the people. Shajahan (2005:30) defines phenomenology as the approach of research based on the way people experience social phenomenon in the world in which they live.

3.5 Research strategies

3.5.1 Positive research strategy - Surveys

Survey research is the study of attitudes, beliefs, and behaviour of people and their settings through questionnaires administered by mail, hand-outs, personal and, telephone interviews, and the Internet. Saunders et al. (2007:138) points out that survey is a popular and common strategy in business and management research and is most frequently used to answer who, what and where and how much. For the purpose of the secondary research, surveys done by Ambler (2008) on TDD across North America and Europe with 121 professional software developers were used. Saunders et al. (2007:249) defines survey based secondary data as the data collected using a survey strategy usually by questionnaires that have already been analysed for their original purpose.

3.5.2 Phenomenological research strategies – Case study

Robson (2002:178) defines case study as a strategy for doing research which involves an empirical investigation of a particular contemporary phenomenon within which its real life context using multiple source of evidence. Yin (2006:111) argues that case study method helps to examine in-depth a case within its real life context. Marshall et al. (2010:19) argues that phenomenological approaches seek to explore, describe, and analyse the meaning of individual's lived experience about how they perceive, describe, feel, judge, remember, make sense, and talk about it with others.

3.5.3 Combined research strategies

Condelli et al. (2004:2) state that combined methods research involves the use of more than one method. Scientific research combined with professional wisdom is the definition of "evidence-based research". The combined research strategy was used for this case study due to the following reasons:

1. TDD is not yet a popular practice in South Africa and TDD still needs to catch-up with the rest of the world.
2. This case study will be more interesting to compare the views of TDD practitioners in South Africa with other practitioners around the world.
3. The survey by Ambler (2008) done in North America and Europe combined with the data collected in South Africa can prove to be a good sample size and spread of data.

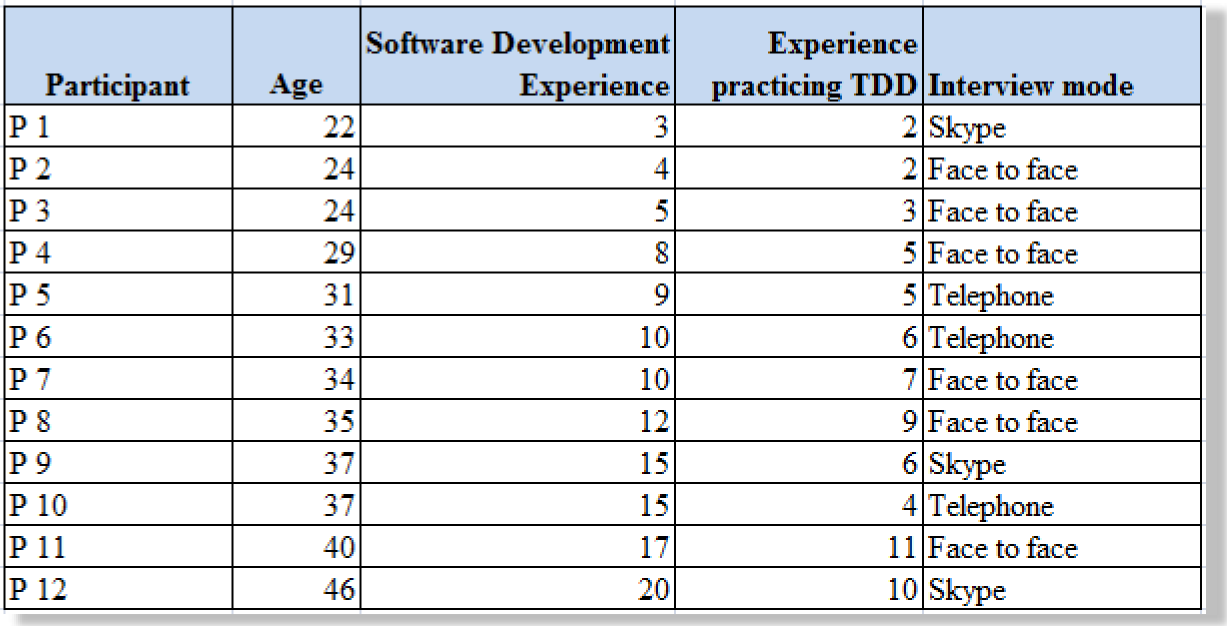
4. This strategy will also reveal the practice difference of TDD between South Africa and the rest of the world if there are any.

3.6 Target population

The source of secondary data was from Ambler (2008) survey and its statistics are as follows:

1. 121 participants.
2. 74% were software developers/modellers and 15% were in management.
3. 52% had 10+ years in IT.
4. 22% worked in organisation of 1000+ people.
5. 96% worked in commercial firms.
6. 50% North American, 26% European.

The primary data was collected via direct interviews and the participant’s overall software development experience and their TDD experience and the interview mode used to collect primary data is shown in table 5.



Participant	Age	Software Development Experience	Experience practicing TDD	Interview mode
P 1	22	3	2	Skype
P 2	24	4	2	Face to face
P 3	24	5	3	Face to face
P 4	29	8	5	Face to face
P 5	31	9	5	Telephone
P 6	33	10	6	Telephone
P 7	34	10	7	Face to face
P 8	35	12	9	Face to face
P 9	37	15	6	Skype
P 10	37	15	4	Telephone
P 11	40	17	11	Face to face
P 12	46	20	10	Skype

Table 5: Participant’s working experience with software development and TDD (Source: Self-generated)

3.7 Sampling

The type of sampling used for this case study is purposive or judgmental which is a non-probability sampling. Saunders et al. (2007:230) states that purposive or judgmental sampling enables the researcher to use their judgment to select cases that will best enable to answer the research questions and to meet the research objectives. The case study on TDD requires software professionals with sound working experience, which naturally falls under judgmental sampling.

3.8 Research instrument

The research instrument that was used to collect secondary data from the internet was the Google search engine. The survey data files (pdf, word document, and power point presentation) published at Ambysoft's web site (Ambler, 2008) were used for analysis. The primary data was collected via unstructured interviews. The participants who were within reach were interviewed face to face. The participants who were far way within South Africa were interviewed via telephone and the participants who were located overseas were interviewed via Skype, which is computer assisted telephonic interview. Krishnaswamy et al. (2006:168) firmly states that where the representativeness of the sample is crucial for the study and the richness of the information is beyond what is implied in the questions planned is desirable, then direct interviewing is best suitable method for collecting data. With very few practitioners of TDD in South Africa and for the depth of information needed for this study unstructured interview proved to be the best research instrument.

3.8.1 Questionnaire construction

There were no predetermined question for this case study and the questions were open-ended and were adaptable according to the flow of the interview. The interviewer ensured using a checklist that the focused areas of the research that is mentioned in section 2.9 (Acclaimed benefits of TDD) and the attributes of TDD mentioned in section 2.13 (Conclusion) along with any noted disadvantages of TDD were covered in the interview. The open-ended questions allowed the participants to be engaged, express true feeling without restrictions, allowing exploring in detail the needed information for the research objectives. Krishnaswamy (2006:306) agrees that with open ended questions the interviewee is free to give any reply and thereby a wide range of answers is obtained and the interviewer in no ways influences the interviewee for a particular type of answer.

3.8.2 Interviews

An interview is a purposeful discussion between two or more people and the interviews can help the interviewer to gather valid and reliable first hand data that are relevant to the research objectives (Saunders et al., 2007:310). The type of interview used for this case study was unstructured interview because of the following benefits:

1. They are informal.
2. Helps to explore the data in-depth in which the interviewer is interested.
3. Interviewees are not restricted and are given an opportunity to speak freely about events, behaviour, and their beliefs. So keeps them engaged.
4. The interaction is non-directive.
5. The interviewer gets extra information from the interviewee along with the focused area of study, which can add value to the research.

3.9 Pilot study

Saunders et al. (2007:606) defines pilot study as a small-scale study to test a questionnaire, interview checklist or observation schedule to minimize the likelihood of the respondents having problem in answering the questions. Therefore 2 out of 12 respondents were interviewed ahead of the actual interview schedule to correct any of the problems in the interview process. The purpose of the pilot test is to refine the questions and to obtain some assessment of the questions for their validity and to ensure that the data collected will enable the investigation questions to be answered (Saunders et al., 2007:386).

3.10 Administration of Questions

The request for the interview with the research topic, areas of focus for the research and the expected outcome of the interview was sent to all participants via email and LinkedIn which is social network website. The average time set for each interview was between 30-45 minutes. The questionnaires were administrated by the researcher by the following methods:

1. ***Face to Face interviews*** – Participants who were within reach were interviewed at their working place.
2. ***Telephonic Interviews*** – Participants who were within South Africa and who were not reachable were interviewed using telephone.
3. ***Computer assisted telephonic interview*** – Participants who were overseas were interviewed via Skype.

3.10.1 Collection of questionnaire

The researcher personally conducted the interviews to collect the research data. 100% feedback was received from the 12 participants. 4 participants were interview face to face at their work place, 5 participants were interviewed via Skype and 3 participants using telephone. The time frame set for the interview was for 30 minutes but most of the interviews exceeded for another 10-15 minutes.

3.11 Data analysis

Data analysis is a process of evaluating data using analytical and logical reasoning to bring order structure and interpretation to a mass of collected data Marshall et al. (2006:154). The data analysis method that was used for this case study is thematic analysis which is a qualitative analytic method for identifying, analyzing, and reporting patterns (themes) within data. The phases of thematic analysis are described below (Braun et al., 2006:16):

1. ***Familiarizing yourself with your data* -** Transcribing data, reading, and noting down initial ideas.
2. ***Generating initial codes*** - Coding interesting features of the data in a systematic fashion across the collected data set, collating data relevant to each code.
3. ***Searching for themes*** - Collating codes into potential themes, gathering all data relevant to each potential theme.
4. ***Reviewing themes*** - Checking in the themes work in relation to the coded extracts generating a thematic map of the analysis.
5. ***Defining and naming themes*** - Analysis to refine the specifics of each theme, generating clear definitions and names for each theme.
6. ***Producing the report*** - Selection of vivid compelling extract, final analysis of selected extracts, relating back of the analysis to the research question and literature, producing a scholarly report of the analysis.

The researcher followed the above phases during the data analysis for this case study.

3.12 Validity and Reliability

Kruger (2011:10) defines that validity refers to the relevance of the data, whereas reliability refers to the accuracy of the data. The researcher claims that the selected population for the study represents an accurate representation of the total population referred to as reliability and the results of the study can be reproduced if the same research is done over time with the same sample population. The researcher also assures that research instrument allowed to truly measure the needed objectives and the results are accurate which supports the validity of the research.

3.13 Limitation of the Study

Out of the many benefits of TDD in extreme programming practices, only a few high impact benefits that are needed for evaluating and for recommendation to company XYZ, are measured in this case study. This case study does not consider the factors that could affect the benefits of TDD such as the size, number of teams, inter dependency between the departments, skill levels of team members with TDD and the complexity of the project on which TDD is being applied. The research solely studies the impact of TDD on the selected key beneficial factors from the participant's knowledge and experience, ignoring the other factors mentioned above.

3.14 Elimination of Bias

It had been important to the researcher to remain objective and to conduct the research according to sound scientific principles and practices at all times. The researcher ensured throughout the interview process that no personal bias in terms gender, attitudes, race, ethnic group, language, or reinforces stereotypes played a negative role against the participants in the research process.

3.15 Ethical Considerations

Saunders et al. (2007:178) refers ethics as the appropriateness of your behaviour in relation to the rights of these who become the subject of your work, or are affected by it. Research ethics therefore relates to questions about how we formulate and clarify our research topic, design our research, gain access, collect, analyse, write-up research findings in a moral and responsible way. The goal of ethics in research is to ensure that no one suffers or adverse consequences as a result of participating in research. Unethical activities include violating nondisclosure agreements, breaking participants confidentiality, misrepresenting results, deceiving people, invoicing regularities, avoiding legal liabilities and many more. The researcher made sure that all these obligations were met during the case study.

3.15.1 Ensuring participants have given informed consent

Prior to the distribution of the questionnaires, each participant was emailed with information and objective of the research and the benefits of taking part which is attached in appendix 7.2. The email also emphasized that the participant is not obliged to participate unless they wished to do so. After the interview the participants were requested to sign the consent form which is attached in appendix 7.1.

3.15.2 Ensuring no harm comes to participants

The form in appendix 7.1, clearly states that the participant’s answers in this research will be confidential. There was no risk involved in participation and the participant can withdraw from the research at any point of time and all the relevant data or information given by the participant will be deleted from the research.

3.15.3 Ensuring confidentiality and anonymity

To ensure that the confidentiality and anonymity are maintained, the researcher will make certain that the promises made to the participants in appendix 7.1, are kept at any cost. In addition the dissertation shall not bear the names of the participants.

3.15.4 Ensuring that permission is obtained

This research did not investigate internally into any organization/company for the case study. The research data crucial for this research was created from the experience and views of the participants. So the permission was obtained from the participants as a statement in the consent letter attached in appendix 7.1.

3.16 Conclusions

The combined research strategy was used for this case study. The primary data collected via direct interviews and the secondary data collected via the survey conducted by Ambler in 2008 was used in chapter 5 to arrive at the conclusion and recommendation.

The next chapter presents the findings of the research and correspondingly discusses and interprets the results from the findings.

Chapter 4 - Results, Discussion, and Interpretation of Findings

4.1 Introduction

Considering the sheer amount of textual and the length of the interviews, it is not possible to deal with case studies in their entirety in this research. Instead this research will be structured around the themes identified in the thematic analyses which discover the themes that are prominent and relevant to this research. These themes are presented in thematic network aiming to facilitate the structuring and depiction of these themes (Attride-Stirling, 2001:387). Thematic networks systematize the extraction of:

1. ***Basic Themes*** – These themes are the lowest-order premises evident in the text.
2. ***Organising Themes*** - These themes are categories of basic themes grouped together to summarize more abstract principles and
3. ***Global Themes*** – These are super-ordinate themes encapsulating the principal metaphors in the text as a whole.

The above themes are then represented as web-like maps describing the relevant themes at each of the three levels, and illustrating the relationships between them as shown in Figure 23. This technique provides practical and effective procedures for conducting a qualitative analysis. Attride-Stirling (2001:386) states that the advantages of using thematic network analysis are:

1. Enables a methodical systematization of textual data.
2. Facilitates the disclosure of each step in the analytic process.
3. Aids the organization of an analysis and its presentation.
4. Allows a sensitive, insightful, and rich exploration of a text's overt structures and underlying patterns.

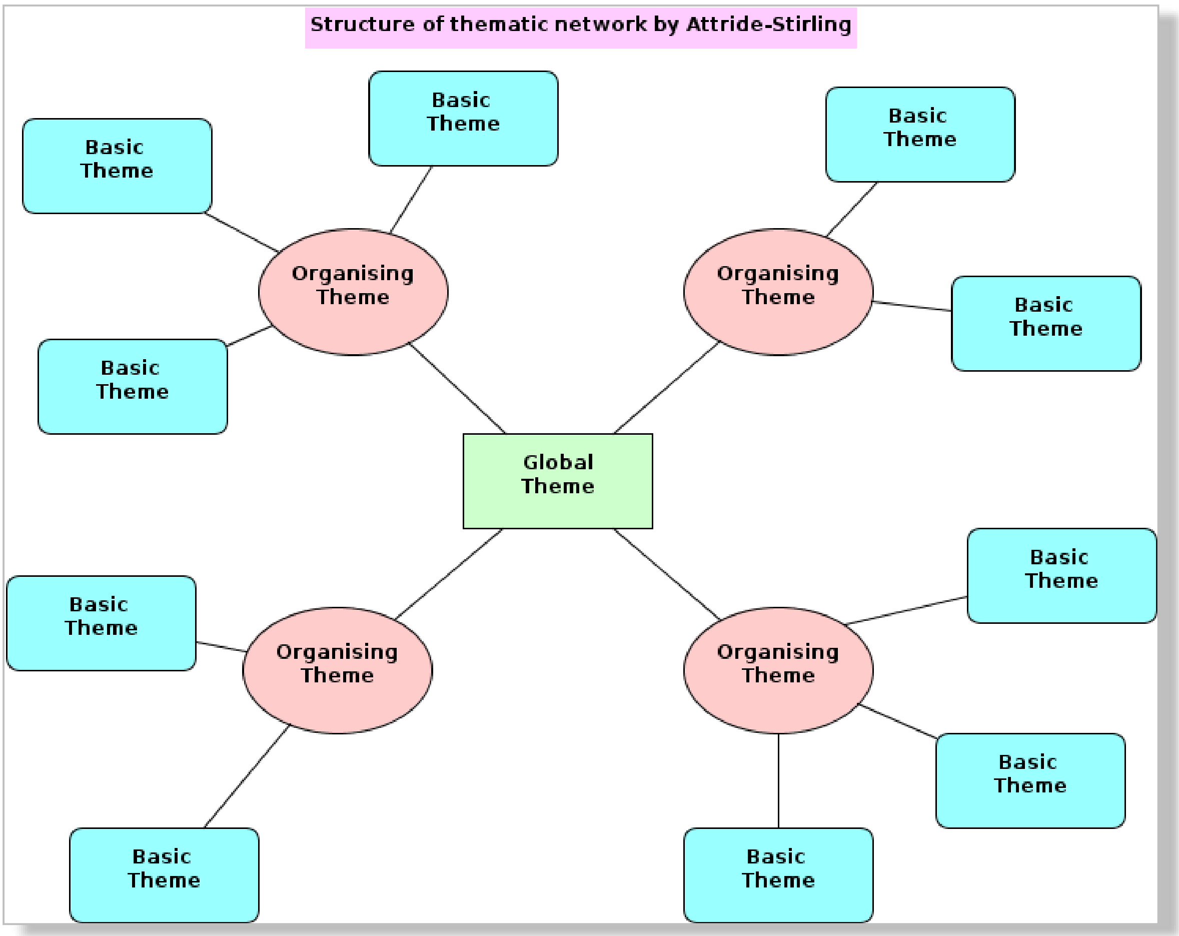


Figure 23: Structure of thematic network (Attride-Stirling, 2001:388)

4.2 Presentation

The results from the survey study of 121 participants conducted by Ambler (2008) and 12 one-on-one research interviews conducted by the researcher are presented in this chapter. The statistics of the participants in the survey conducted by Ambler (2008) and the background information of the participants in the interview are already mentioned in ***section 3.6 (Target population).***

4.2.1 Which best describes your current position?

Survey result

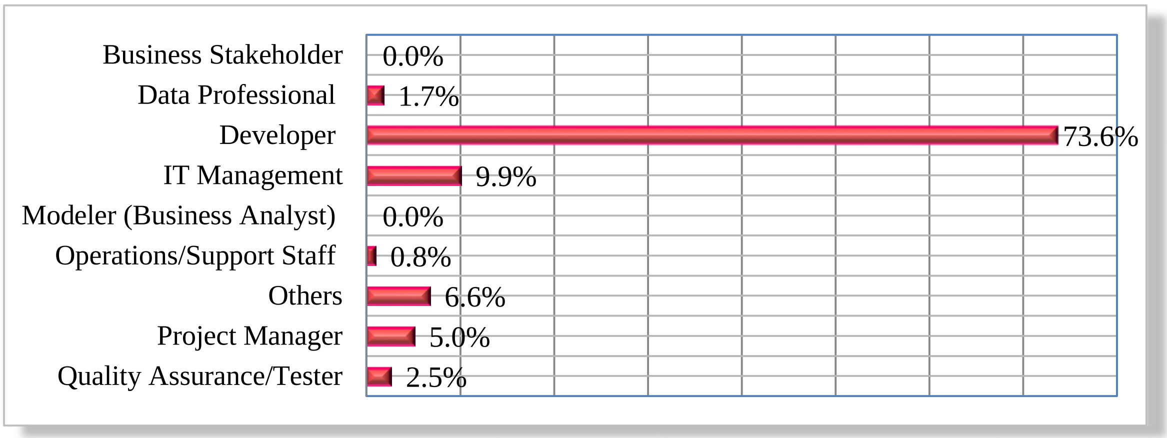
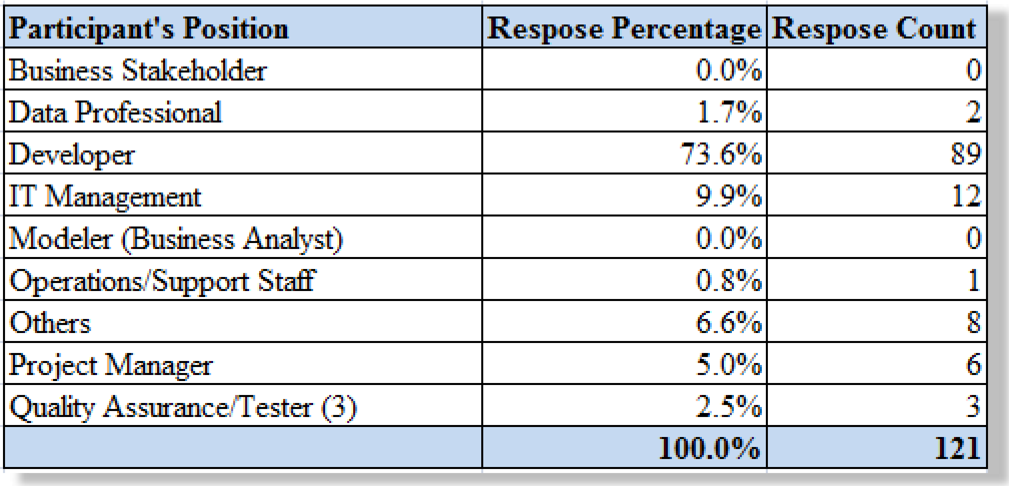


Figure 24: Position break down graph (Source: Self-generated)



Participant's Position	Respose Percentage	Respose Count
Business Stakeholder	0.0%	0
Data Professional	1.7%	2
Developer	73.6%	89
IT Management	9.9%	12
Modeler (Business Analyst)	0.0%	0
Operations/Support Staff	0.8%	1
Others	6.6%	8
Project Manager	5.0%	6
Quality Assurance/Tester (3)	2.5%	3
	100.0%	**121**

Table 7: Position break down (Source: Self generate)

Case study Result

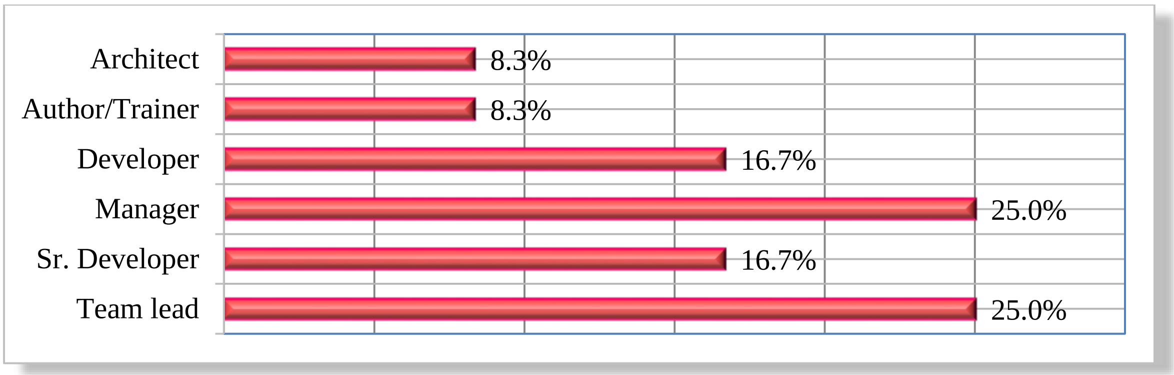
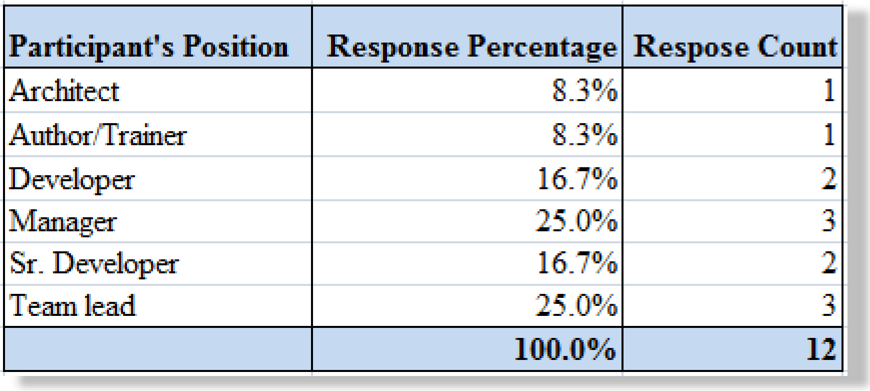


Figure 25: Position break down graph (Source: Self-generated)



Participant's Position	Response Percentage	Respose Count
Architect	8.3%	1
Author/Trainer	8.3%	1
Developer	16.7%	2
Manager	25.0%	3
Sr. Developer	16.7%	2
Team lead	25.0%	3
	100.0%	**12**

Table 7: Position break down (Source: Self generate)

4.2.2 How many years of experience in IT do you have?

Survey result

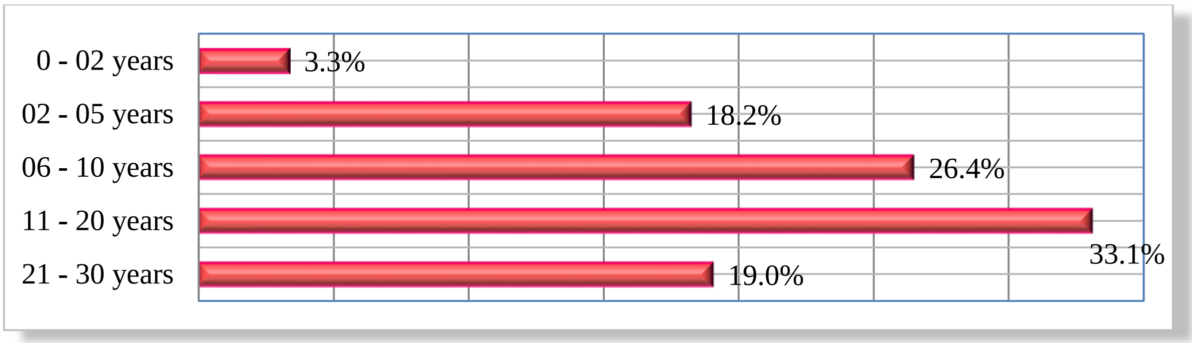
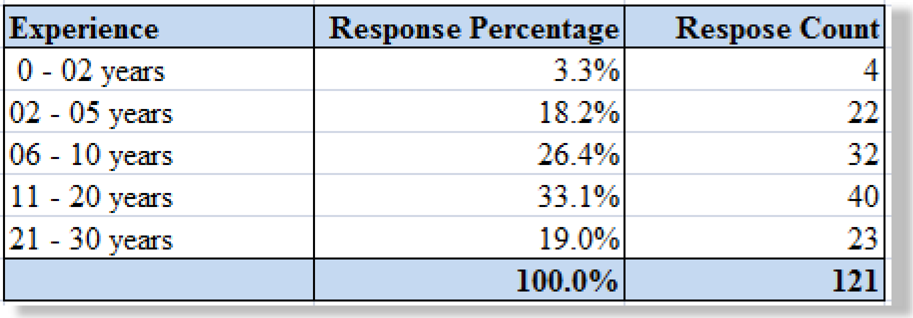


Figure 26: Experience bar graph (Source: Self-generated)



Experience	Response Percentage	Respose Count
0 - 02 years	3.3%	4
02 - 05 years	18.2%	22
06 - 10 years	26.4%	32
11 - 20 years	33.1%	40
21 - 30 years	19.0%	23
	100.0%	**121**

Table 8: IT Experience response statistics (Source: Self generate)

Case study Result

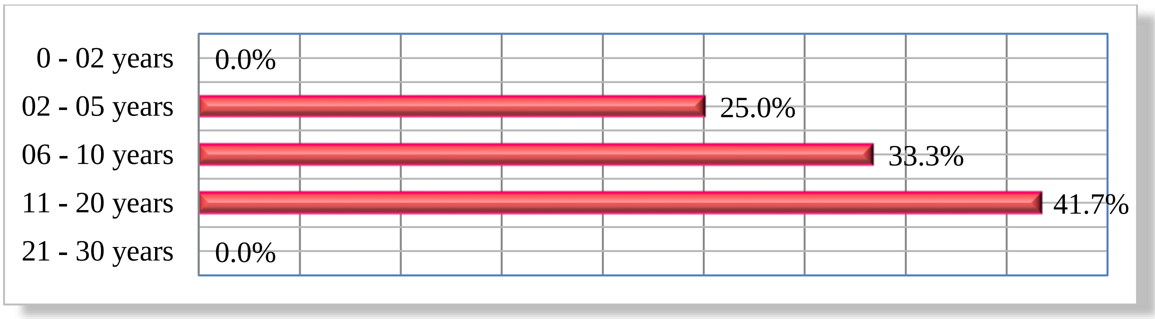
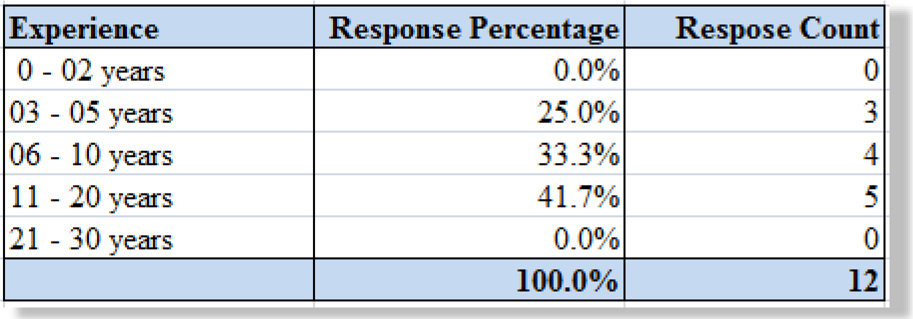


Figure 27: Experience graph (Source: Self-generated)



Experience	Response Percentage	Respose Count
0 - 02 years	0.0%	0
03 - 05 years	25.0%	3
06 - 10 years	33.3%	4
11 - 20 years	41.7%	5
21 - 30 years	0.0%	0
	100.0%	**12**

Table 9: Experience response statistics (Source: Self generate)

4.2.3 Where are you based?

Survey result

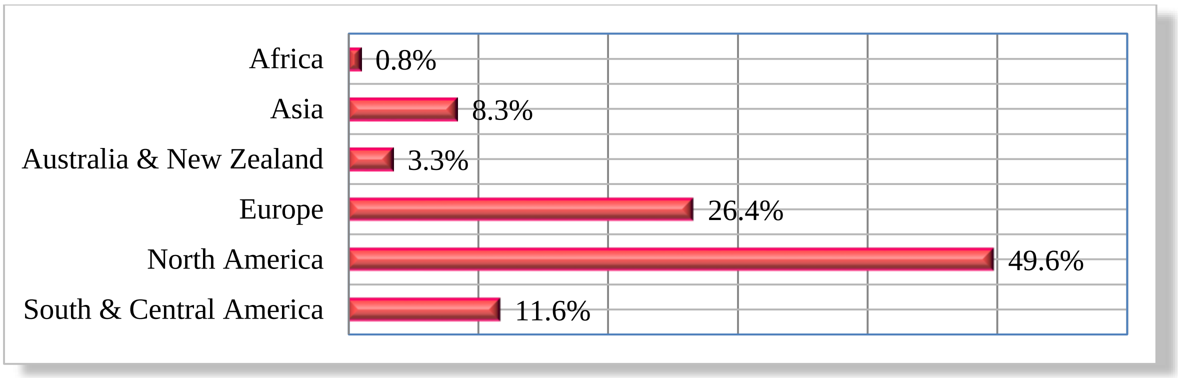
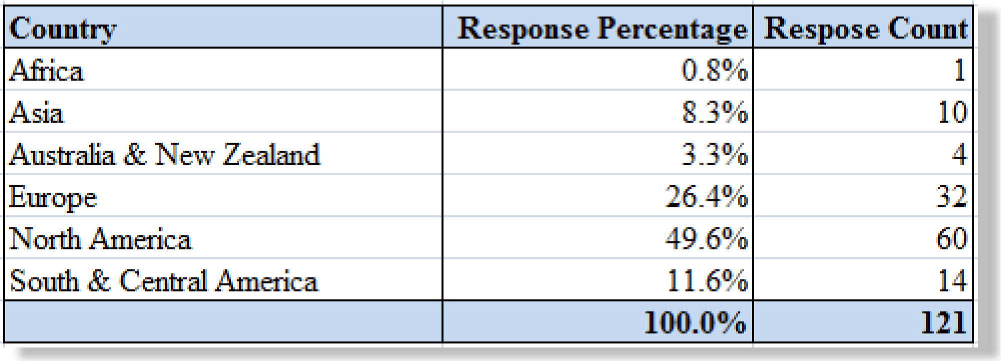


Figure 28: Location graph (Source: Self-generated)



Country	Response Percentage	Respose Count
Africa	0.8%	1
Asia	8.3%	10
Australia & New Zealand	3.3%	4
Europe	26.4%	32
North America	49.6%	60
South & Central America	11.6%	14
	100.0%	**121**

Table 10: Location statistics (Source: Self generate)

Case study

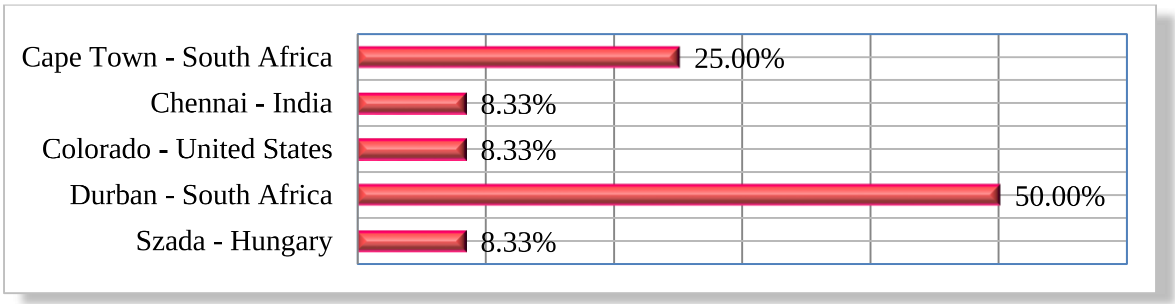
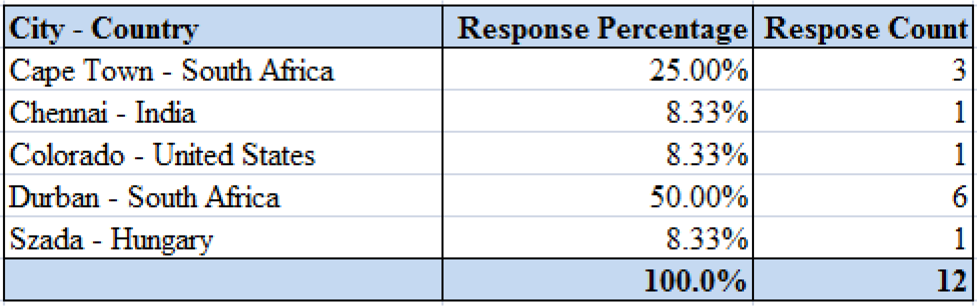


Figure 29: Location graph (Source: Self-generated)



City - Country	Response Percentage	Respose Count
Cape Town - South Africa	25.00%	3
Chennai - India	8.33%	1
Colorado - United States	8.33%	1
Durban - South Africa	50.00%	6
Szada - Hungary	8.33%	1
	100.0%	**12**

Table 11: Location statistics (Source: Self generate)

4.2.4. What is the total number of people in your organization?

Survey result

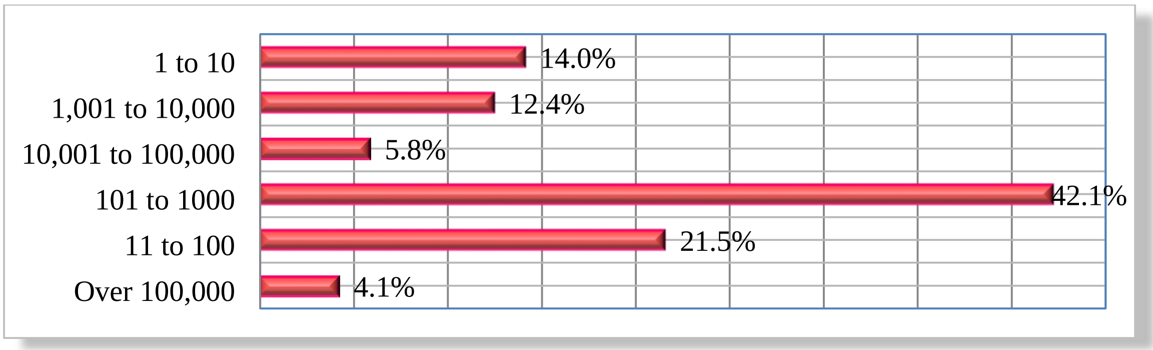
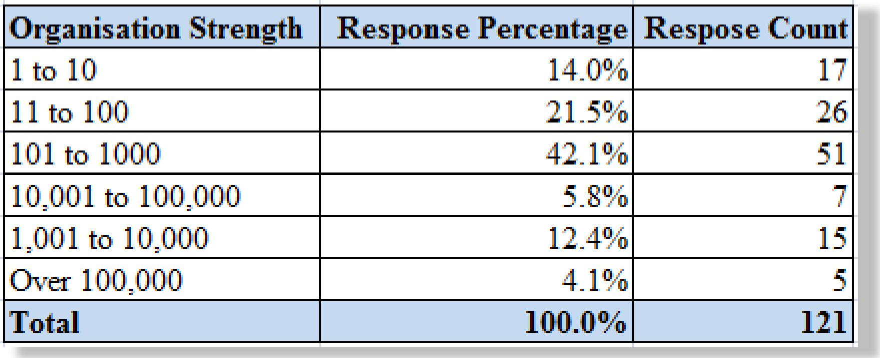


Figure 30: Organization strength break down graph (Source: Self-generated)



Organisation Strength	Response Percentage	Respose Count
1 to 10	14.0%	17
11 to 100	21.5%	26
101 to 1000	42.1%	51
10,001 to 100,000	5.8%	7
1,001 to 10,000	12.4%	15
Over 100,000	4.1%	5
Total	**100.0%**	**121**

Table 12: Organization strength break down graph (Source: Self-generated)

Case study Result

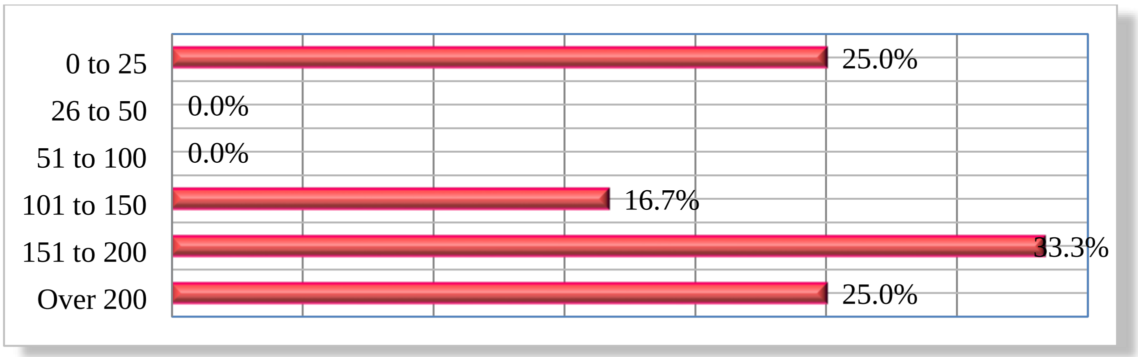


Figure 30: Organization strength break down graph (Source: Self-generated)

Organisation Strength	**Response Percentage**	**Response Count**
0 to 25	25.0%	3
26 to 50	0.0%	0
51 to 100	0.0%	0
101 to 150	16.7%	2
151 to 200	33.3%	4
Over 200	25.0%	3
Total	**100.0%**	**12**

Table 13: Organization strength break down graph (Source: Self-generated)

4.2.5 Which sector is your organization primarily in?

Survey result

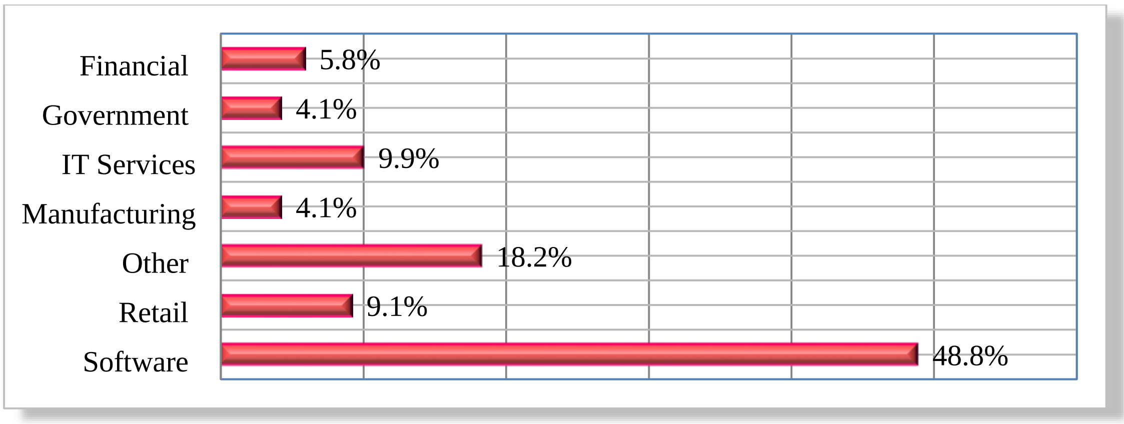


Figure 32: Organization Sector break down graph (Source: Self-generated)

Sector	Response Percentage	Response Count
Financial	5.8%	7
Government	4.1%	5
IT Services	9.9%	12
Manufacturing	4.1%	5
Other	18.2%	22
Retail	9.1%	11
Software	48.8%	59
Total	**100.0%**	**121**

Table 14: Organization Sector break down graph (Source: Self-generated)

Case study Result

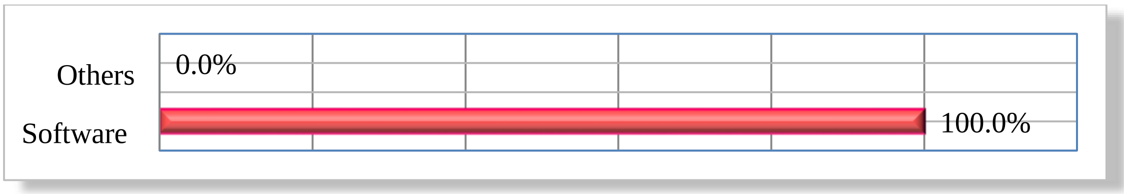


Figure 32: Organization Sector break down graph (Source: Self-generated)

Sector	Response Percentage	Response Count
Others	0.0%	0
Software	100.0%	12
Total	**100.0%**	**12**

Table 15: Organization Sector break down graph (Source: Self-generated)

4.2.6 What is your experience with developer TDD?

Survey result

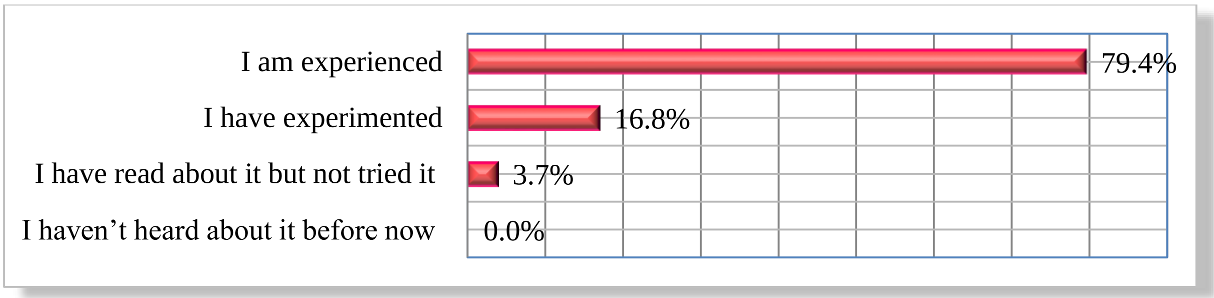


Figure 34: TDD Experience break down graph (Source: Self-generated)

Experience	Response Percentage	Response Count
I am experienced	79.4%	85
I have experimented	16.8%	18
I have read about it but not tried it	3.7%	4
I haven’t heard about it before now	0.0%	0
	100.0%	**107**

Table 16: TDD Experience break down graph (Source: Self-generated)

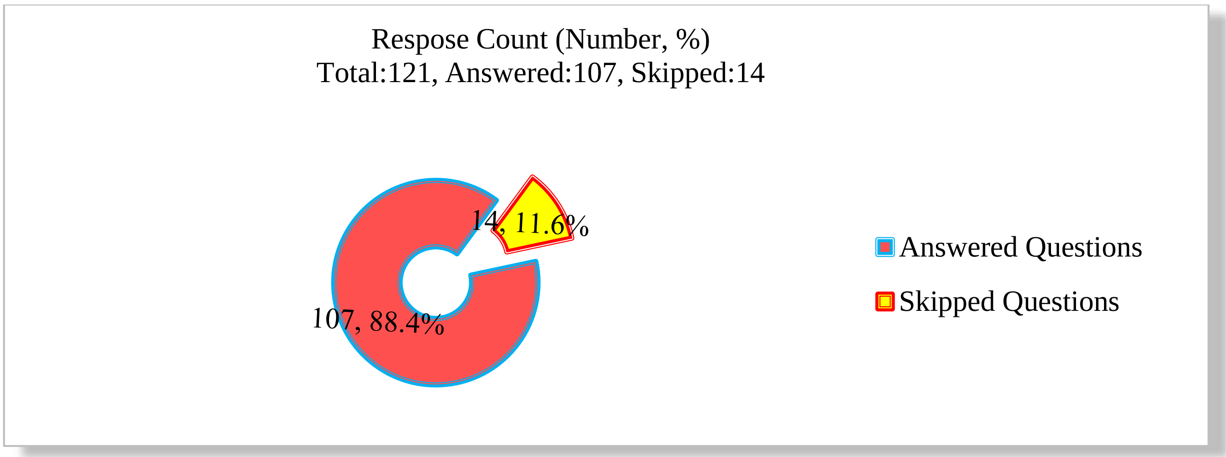


Figure 34: TDD Experience responses (Source: Self-generated)

Case study

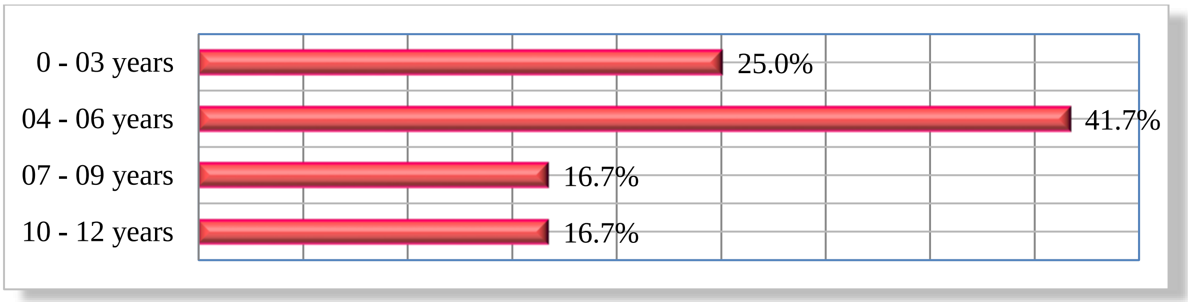
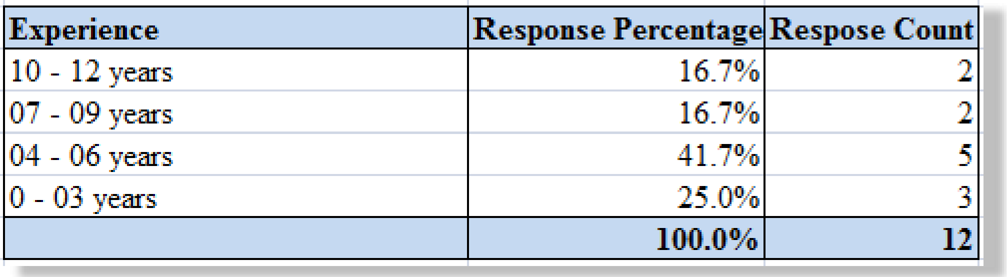


Figure 35: TDD Experience break down graph (Source: Self-generated)



Experience	Response Percentage	Respose Count
10 - 12 years	16.7%	2
07 - 09 years	16.7%	2
04 - 06 years	41.7%	5
0 - 03 years	25.0%	3
	100.0%	**12**

Table 17: TDD Experience break down graph (Source: Self-generated)

4.2.7 What is your personal belief in the effectiveness of Developer TDD?

Survey result

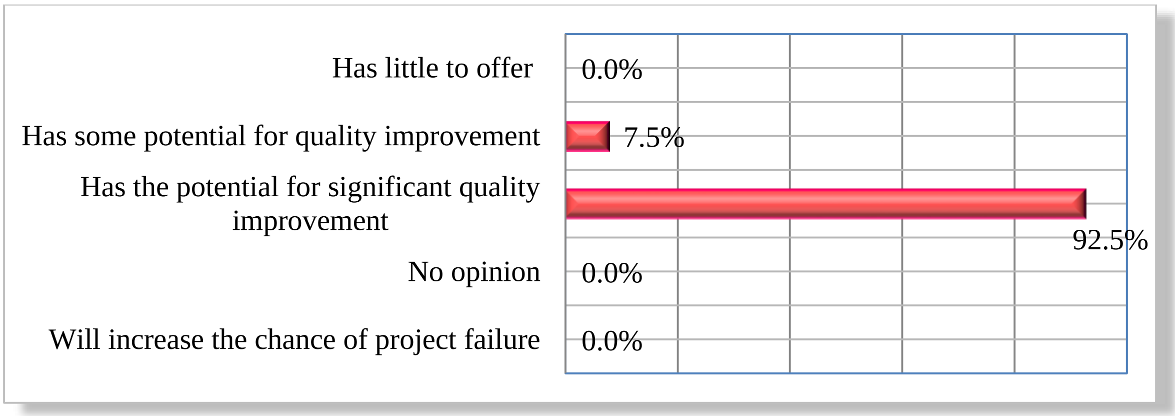


Figure 36: TDD Effectiveness break down graph (Source: Self-generated)

Experience	Response Percentage	Response Count
Has little to offer	0.0%	0
Has some potential for quality improvement	7.5%	8
Has the potential for significant quality improvement	92.5%	99
No opinion	0.0%	0
Will increase the chance of project failure	0.0%	0
	100.0%	**107**

Table 18: TDD Effectiveness break down graph (Source: Self-generated)

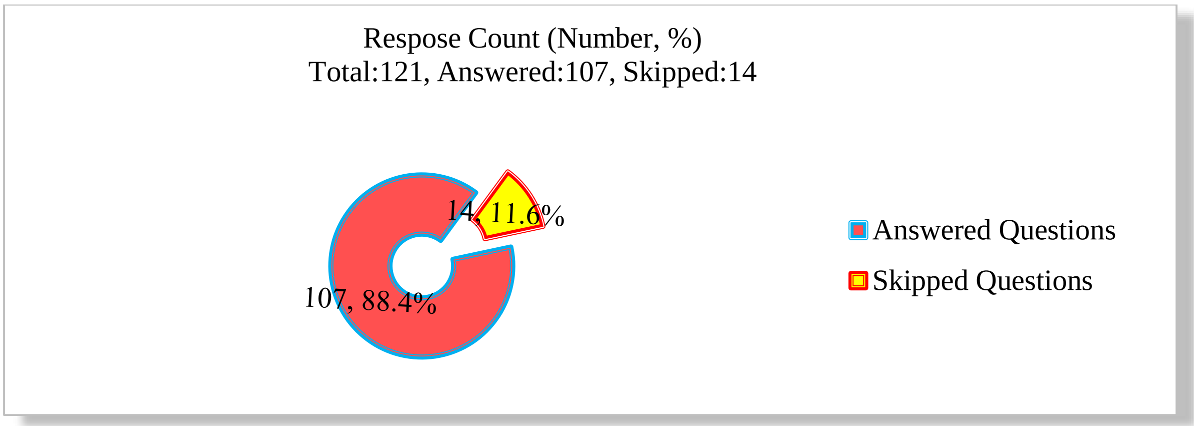


Figure 37: TDD Effectiveness Responses (Source: Self-generated)

4.2.8 What benefits of Developer TDD have you actually experienced?

Survey result

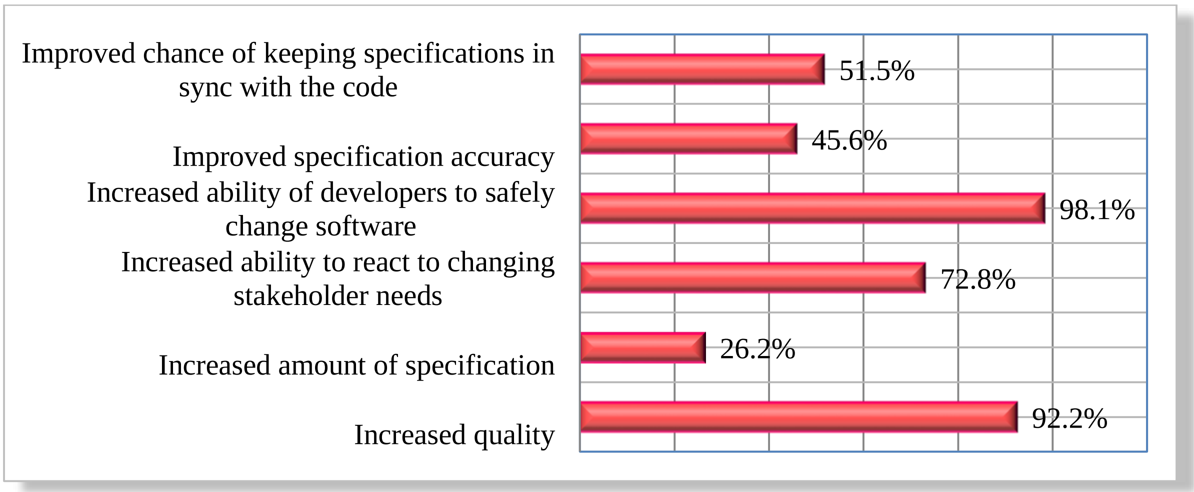


Figure 38: TDD Benefits break down (Source: Self-generated)

Benefits	Response Percentage	Response Count
Improved chance of keeping specifications in sync with the code	51.5%	53
Improved specification accuracy	45.6%	47
Increased ability of developers to safely change software	98.1%	101
Increased ability to react to changing stakeholder needs	72.8%	75
Increased amount of specification	26.2%	27
Increased quality	92.2%	95
Skipped Question	14.9%	18
Answered Question	85.1%	103
	100.0%	**121**

Table 19: TDD Benefits break down (Source: Self-generated)

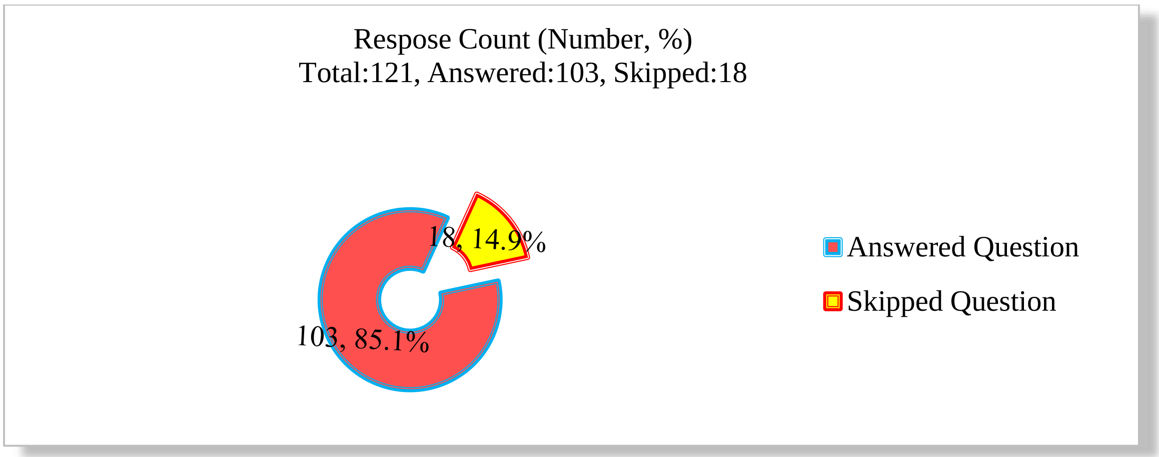


Figure 39: TDD Benefits responses (Source: Self-generated)

4.2.9 What is your opinion about the ease of learning Developer TDD?

<u>Survey result</u>

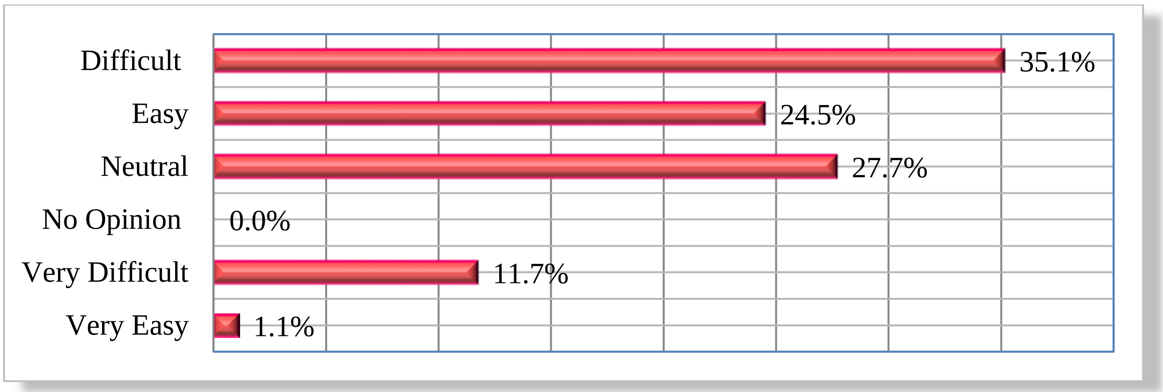
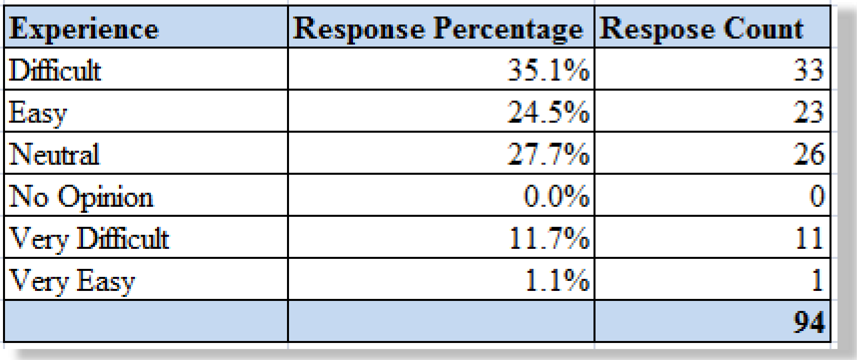


Figure 40: Ease of learning TDD break down (Source: Self-generated)



Experience	Response Percentage	Respose Count
Difficult	35.1%	33
Easy	24.5%	23
Neutral	27.7%	26
No Opinion	0.0%	0
Very Difficult	11.7%	11
Very Easy	1.1%	1
		94

Table 20: Ease of learning TDD break down (Source: Self-generated)

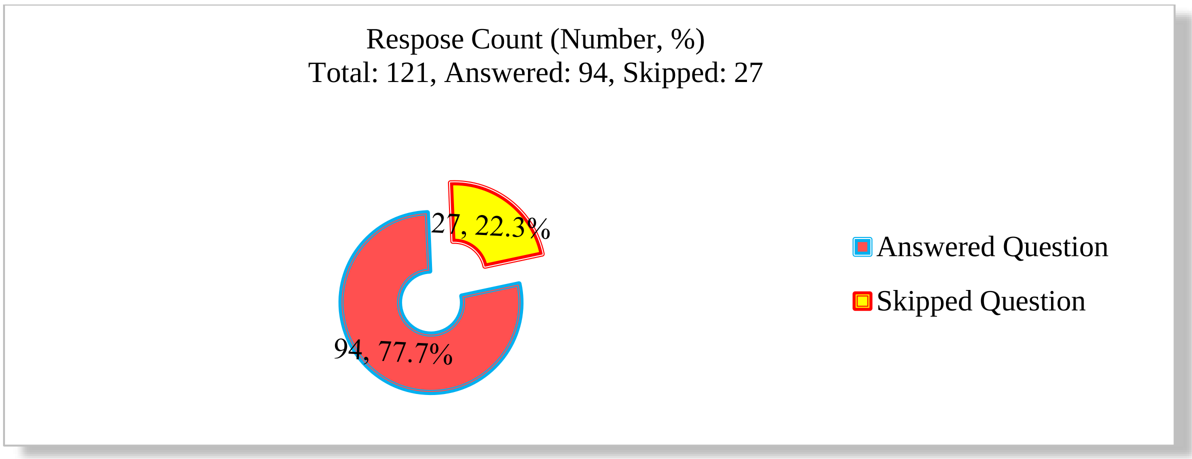


Figure 41: Ease of learning TDD responses (Source: Self-generated)

4.2.10 Based on your experience, how effective are the following strategies for learning Developer TDD?

Survey result

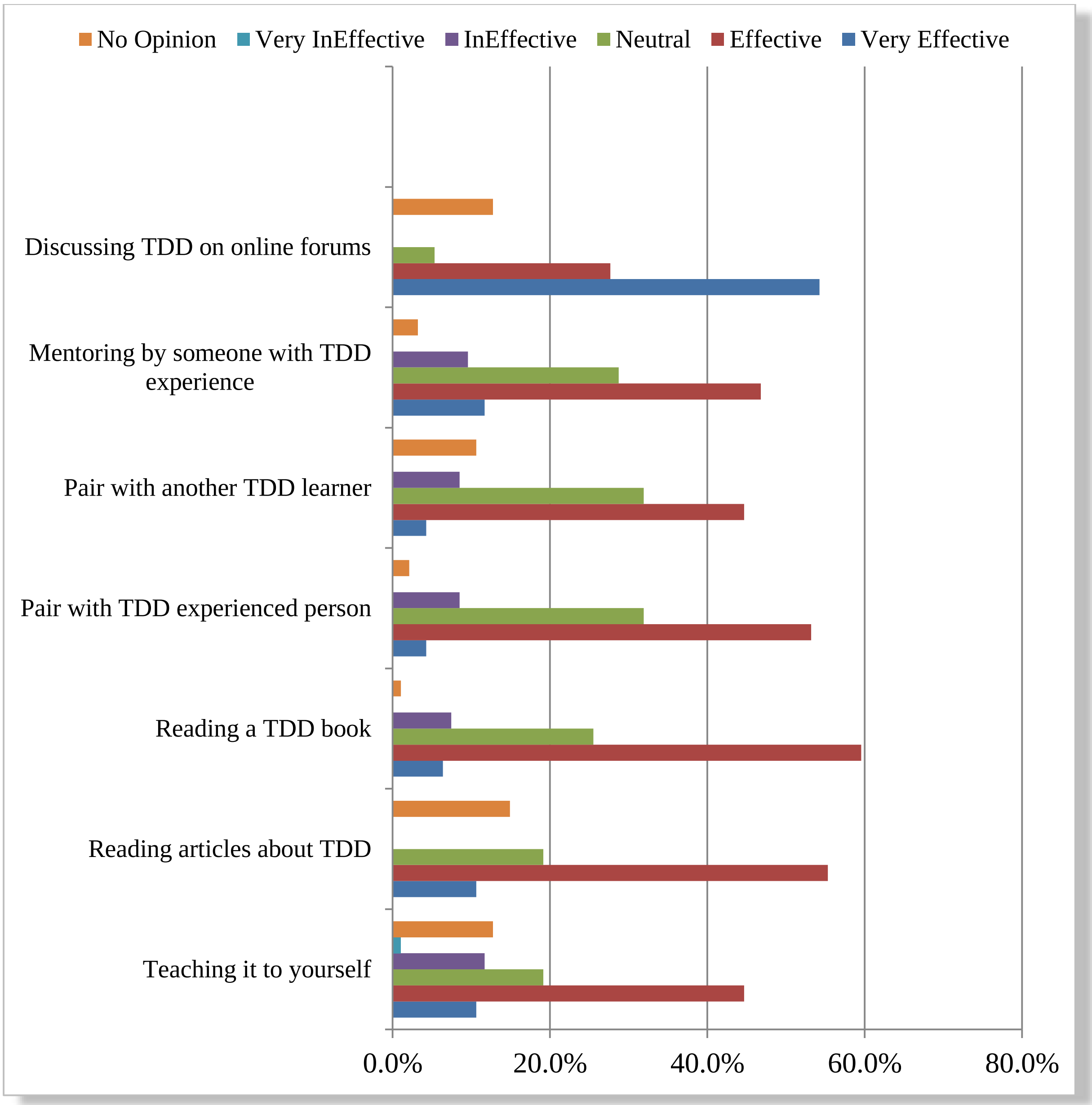


Figure 42: Strategies for learning TDD and its effectiveness (Source: Self-generated)

	Very Effective	Effective	Neutral	In Effective	Very Ineffective	No Opinion
Discussing TDD on online forums	73.4%	12.8%	2.1%	2.1%	0.0%	9.6%
Mentoring by someone with TDD experience	10.6%	44.7%	19.1%	11.7%	1.1%	12.8%
Pair with another TDD learner	10.6%	55.3%	19.1%	0.0%	0.0%	14.9%
Pair with TDD experienced person	6.4%	59.6%	25.5%	7.4%	0.0%	1.1%
Reading a TDD book	4.3%	53.2%	31.9%	8.5%	0.0%	2.1%
Reading articles about TDD	4.3%	44.7%	31.9%	8.5%	0.0%	10.6%
Teaching it to yourself	11.7%	46.8%	28.7%	9.6%	0.0%	3.2%
Training in TDD	54.3%	27.7%	5.3%	0.0%	0.0%	12.8%

Table 21: Strategies for learning TDD and its effectiveness (Source: Self-generated)

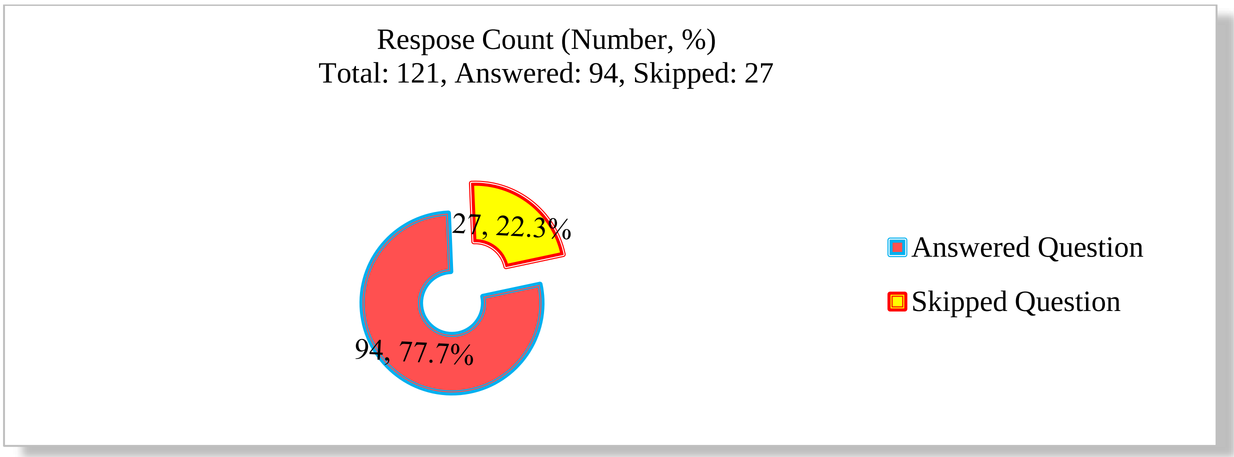


Figure 43: Strategies for learning TDD responses (Source: Self-generated)

4.2.11 When it comes to adopting TDD within your organization, how are the potential challenges affecting you?

Survey result

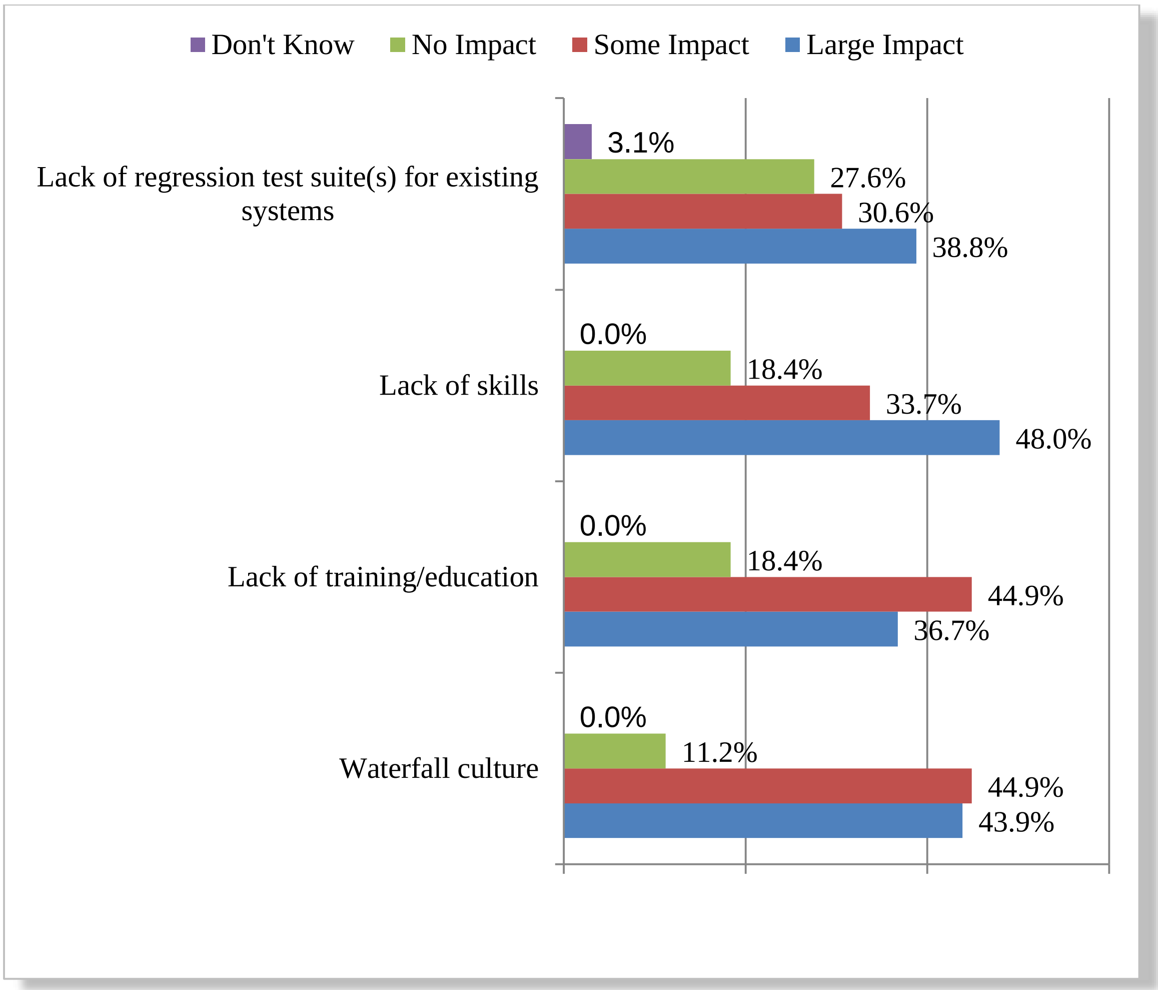
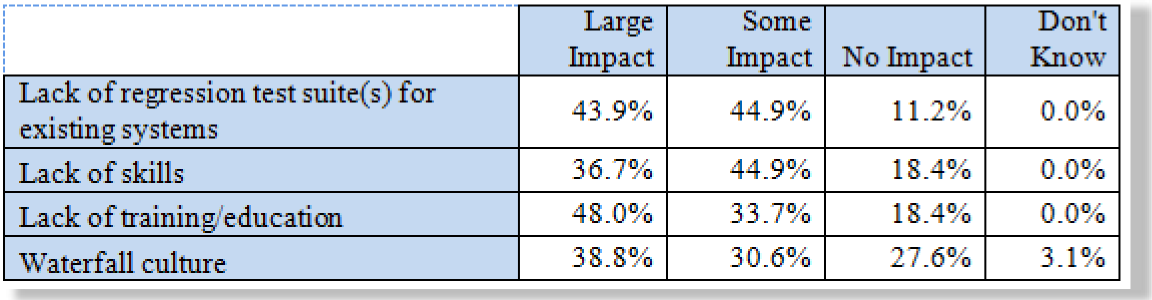


Figure 44: Adopting TDD challenges (Source: Self-generated)



	Large Impact	Some Impact	No Impact	Don't Know
Lack of regression test suite(s) for existing systems	43.9%	44.9%	11.2%	0.0%
Lack of skills	36.7%	44.9%	18.4%	0.0%
Lack of training/education	48.0%	33.7%	18.4%	0.0%
Waterfall culture	38.8%	30.6%	27.6%	3.1%

Table 22: Adopting TDD challenges (Source: Self-generated)

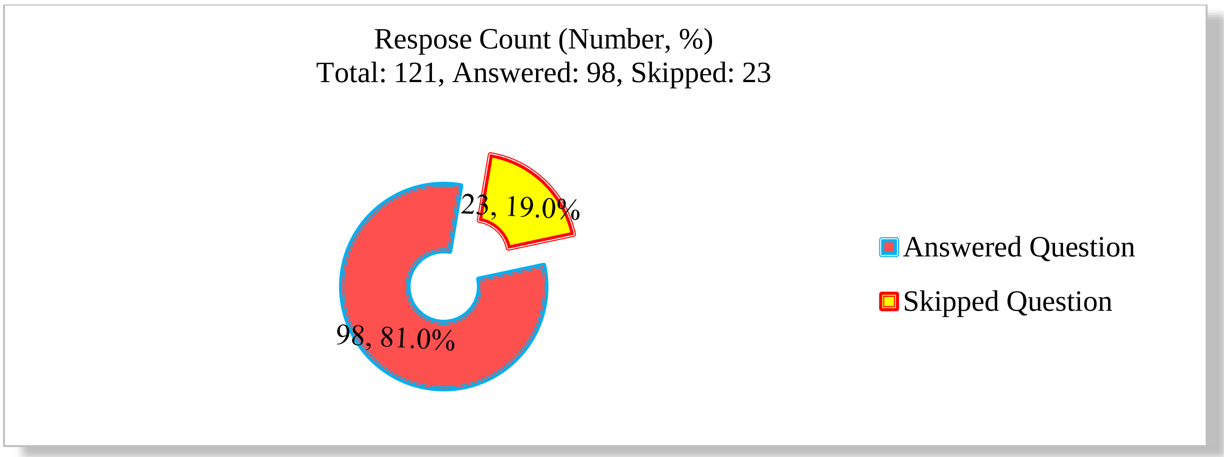


Figure 45: Adopting TDD challenges (Source: Self-generated)

4.2.12 What is the support for developer TDD within your organization?

Survey result

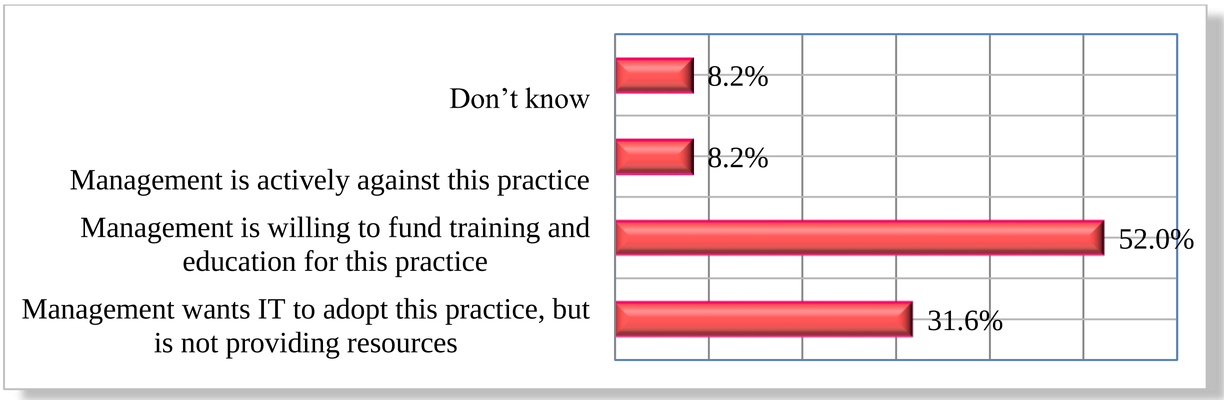


Figure 46: Support of TDD within the organization (Source: Self-generated)

Support	**Response Percentage**	**Response Count**
Don't know	8.2%	8
Management is actively against this practice	8.2%	8
Management is willing to fund training and education for this practice	52.0%	51
Management wants IT to adopt this practice, but is not providing resources	31.6%	31
	100.0%	**98**

Table 23: Support of TDD within the organization (Source: Self-generated)

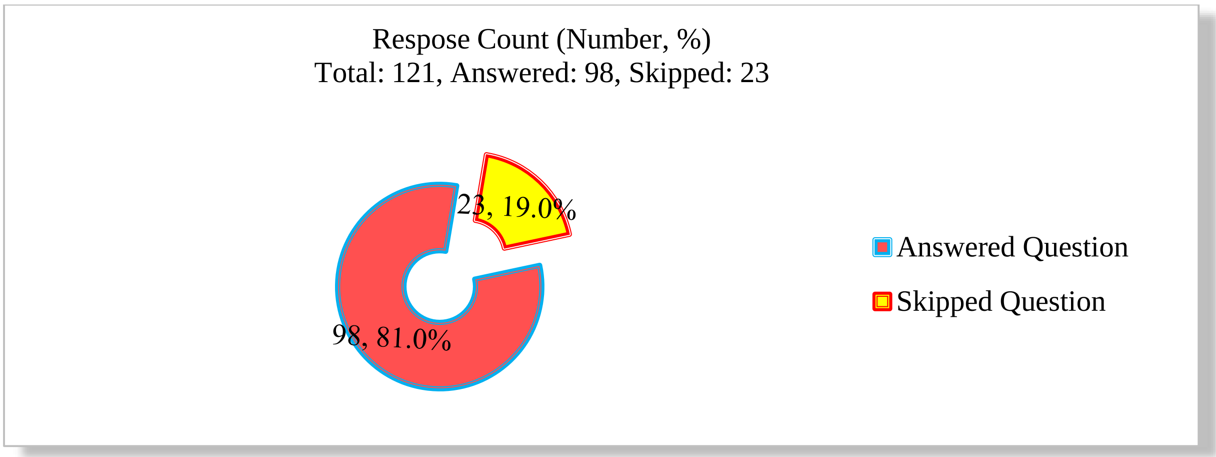


Figure 47: Support of TDD within the organization responses (Source: Self-generated)

4.2.13 What other forms of testing is your team doing?

Survey result

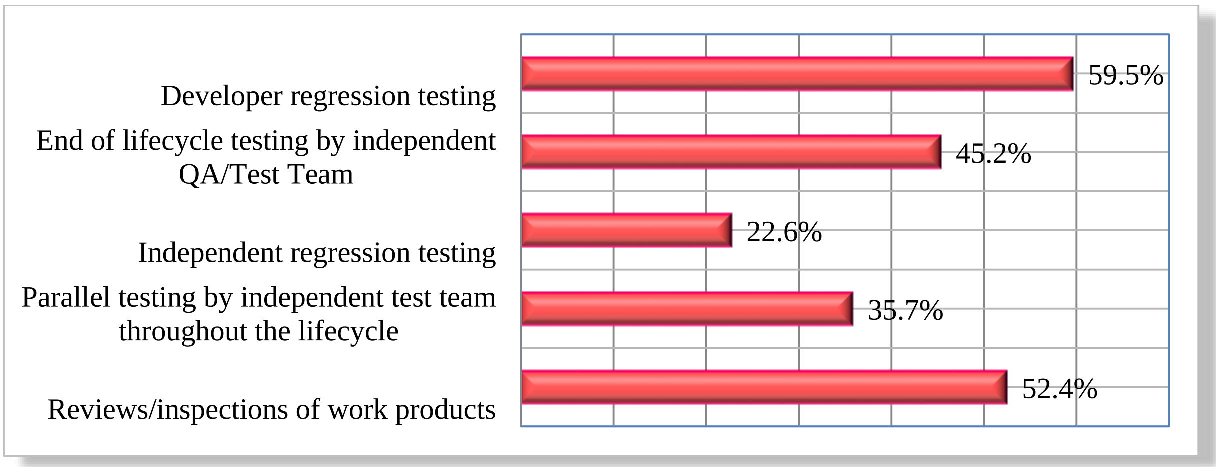


Figure 48: Other forms of testing (Source: Self-generated)

Support	**Response Percentage**	**Respons e Count**
Developer regression testing	59.5%	50
End of lifecycle testing by independent QA/Test Team	45.2%	38
Independent regression testing	22.6%	19
Parallel testing by independent test team throughout the lifecycle	35.7%	30
Reviews/inspections of work products	52.4%	44

Table 24: Other forms of testing (Source: Self-generated)

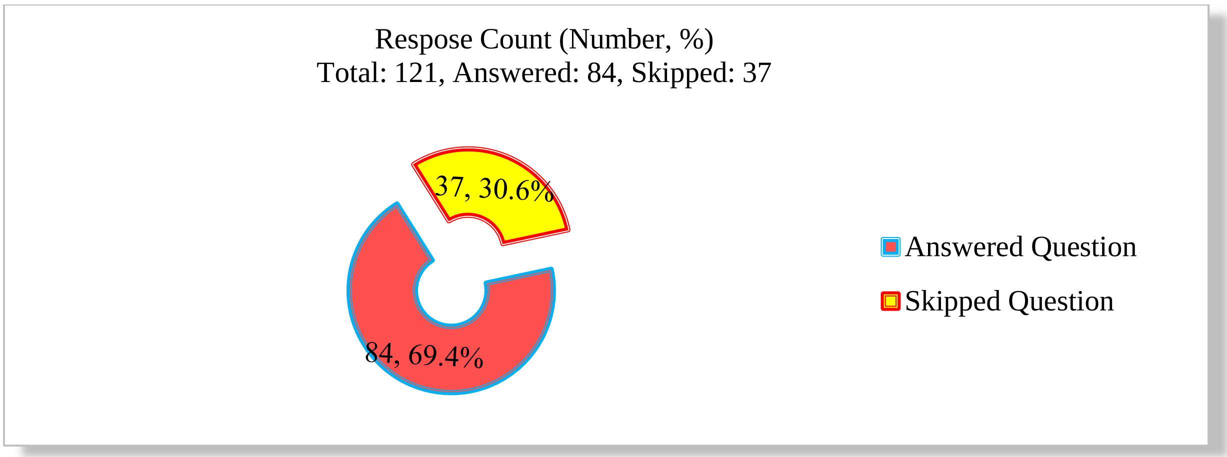


Figure 49: Other forms of testing responses (Source: Self-generated)

4.2.14 Who is writing tests and when are they doing it?

Survey result

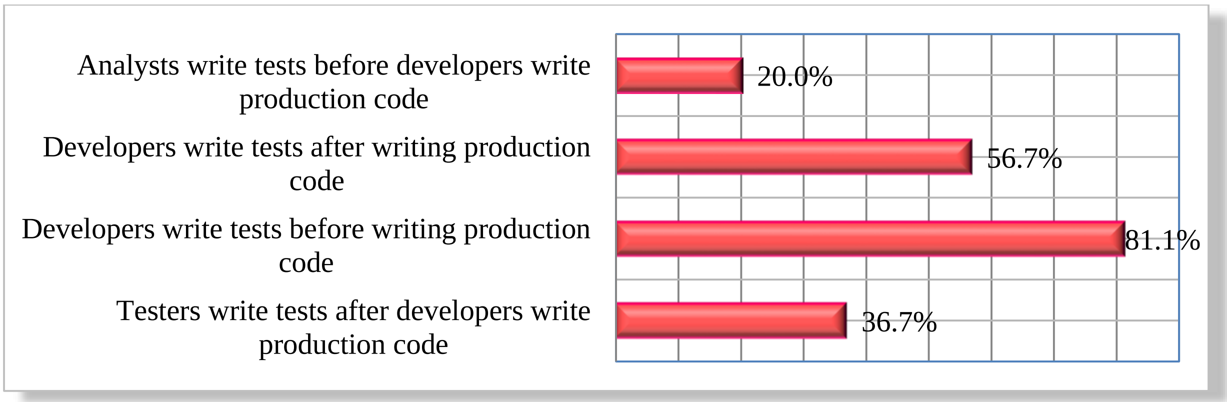


Figure 50: Who is responsible for writing test (Source: Self-generated)

Tests	Response Percentage	Response Count
Analysts write tests before developers write production code	20.0%	18
Developers write tests after writing production code	56.7%	51
Developers write tests before writing production code	81.1%	73
Testers write tests after developers write production code	36.7%	33

Table 25: Who is responsible for writing test (Source: Self-generated)

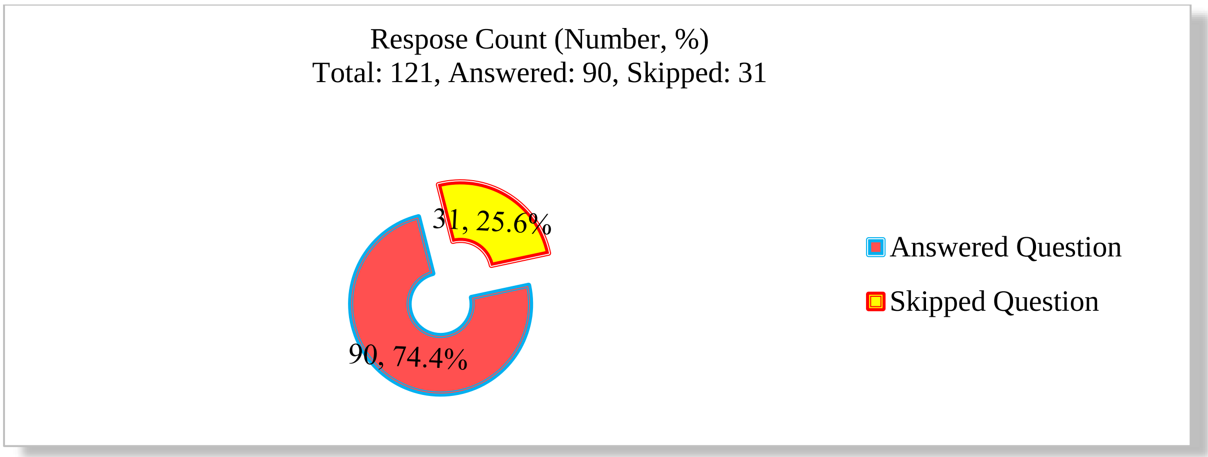


Figure 51: Who is responsible for writing test responses (Source: Self-generated)

4.2.15 How is your team capturing requirements specifications?

Survey result

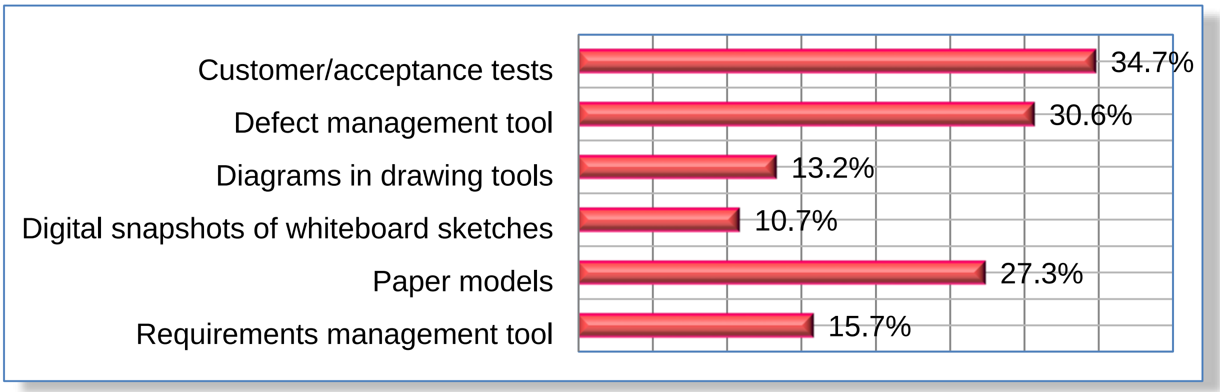


Figure 52: Capturing requirements specifications (Source: Self-generated)

Requirements specifications	Response Percentage	Response Count
Customer/acceptance tests	34.7%	42
Defect management tool	30.6%	37
Diagrams in drawing tools	13.2%	16
Digital snapshots of whiteboard sketches	10.7%	13
Paper models	27.3%	33
Requirements management tool	15.7%	19

Table 26: Capturing requirements specifications (Source: Self-generated)

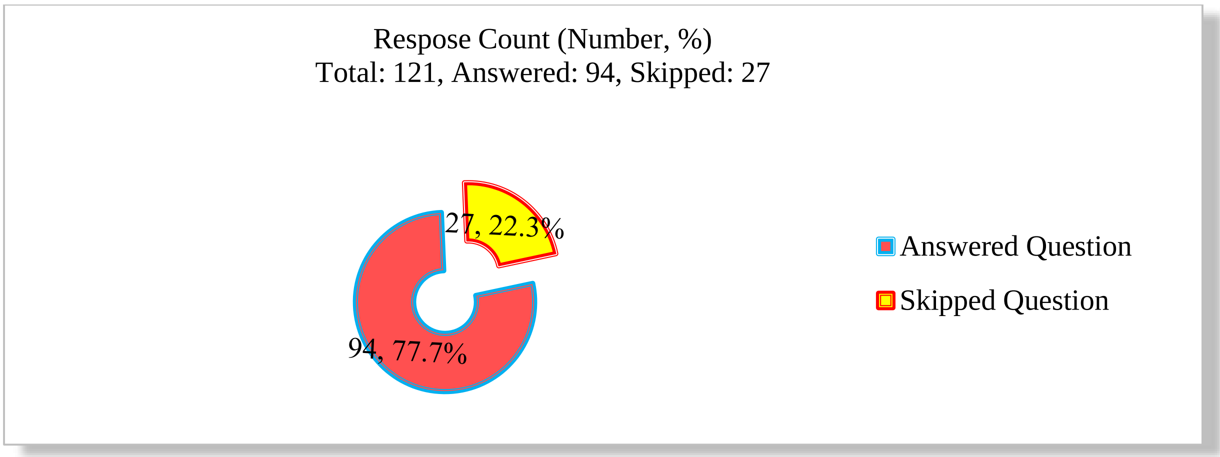


Figure 53: Capturing requirements specifications responses (Source: Self-generated)

4.3 Interpretation and Discussion

The full thematic network diagram derived from the case study is shown in Appendix 1. From the transcripts collected the most salient constructs in the discussions were identified and shaped into a finite set of codes that were discrete enough to avoid redundancy, and global enough to be meaningful. The transcripts were then dissected, classified and organised according to these codes. The basic themes were base lined from ***section 2.13 (conclusions)*** that are the key areas of this research. The discussions with the survey results, interview results and the support for the themes from the academic researchers are discussed under each basic theme.

4.3.1 Moving to TDD

The four basic themes that were identified for the organizing theme "Moving towards TDD" is shown in figure 54.

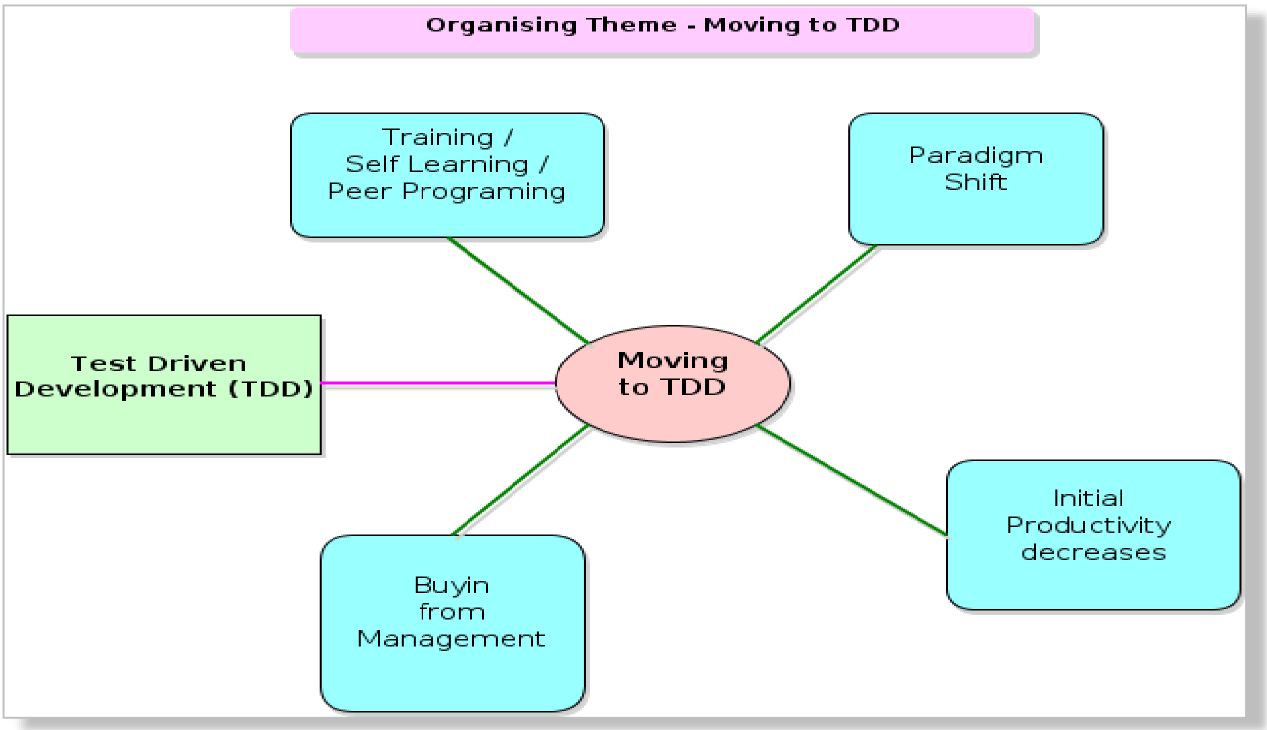

Organising Theme - Moving to TDD
Training / Self Learning / Peer Programing
Paradigm Shift
Test Driven Development (TDD)
Moving to TDD
Initial Productivity decreases
Buyin from Management

Figure 54: Organising theme for "Moving to TDD" with its basis themes and link to Global Theme. (Source: Self-generated)

4.3.1.1 Paradigm shift

Ambler (2008) survey results in section 4.2.9 indicate that the majority of the developers moving towards TDD had a difficult time in transition and the learning curve was hard. There were no arguments from the participants against this paradigm shift. All the participants agreed that the experience of moving to TDD was found challenging and the experience of throwing away old habits of programming and to learn the new TDD techniques that are well disciplined was tough. Gold et al. (2005:26) states that the role of the programmers with TDD is completely reversed. Instead of thinking about algorithm and structure the programmer has to conceptualize the situation more from the perspective of the end user. Therefore programmers coming from the traditional software development find this shift the hardest part to learn.

4.3.1.2 Training, Self-learning and Pair/Peer programming

Ambler (2008) survey results shown in section 4.2.10 shows that the most effective way of learning TDD is through discussing TDD on online forums, followed by reading a good TDD book and then mentoring and coaching. Most of the participants favoured formal training and mentoring by an experienced TDD practitioner as the most effective way to learn and train in TDD. One of the participant whose is also a manager states that for new learners peer or pair programming gets the desired results. One participant who is an author and trainer argues that with the experienced developers it's the unlearning of the accumulated habits in programming via the traditional mythology that takes longer to offload than learning TDD. Koskela (2008:441) points out that the common relationship of influence during a change is a senior staff member mentoring a junior staff member making their personal adoption processes easier because of the leadership brought by seniority.

4.3.1.3 Initial Productivity Decrease

All the participants pointed out the measurement of productivity cannot be accurate due to unclear definition of metrics measured. In terms of number of features released in a sprint, the initial productivity of the developers decreases due to learning curve. But when we look at the quality and design of the code the standards are high. Collier (2011:217) states that TDD like any practice change will feel awkward and frustrating in the beginning, as you will feel less productive. This is due to the fact that the developer is now doing detailed design, development, and testing combined in the same cycle. The initial shippable code may be less, but it will be production-ready when finished. Schmidkonz et al. (2007) in his study on the impact of TDD on projects indicate that the productivity drops to 60% initially and then the productivity raises as the programmers get comfortable with TDD. When compared to the team morale and quality the value are high in the initial stages and thereafter keeps on increasing as shown in figure 55.

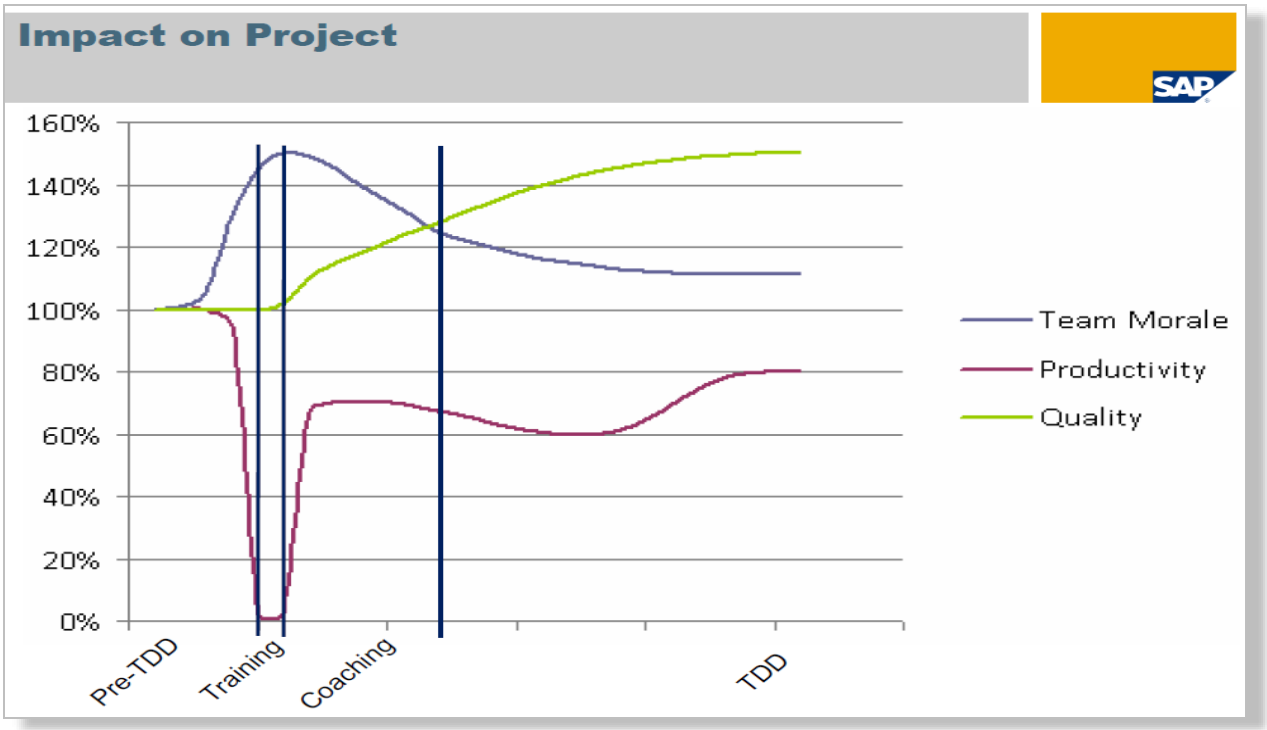


Figure 55: Impact of TDD on projects, adapted from Schmidkonz (2007)

4.3.1.4 Buy-in from the Management

Ambler (2008) survey result shown in section 4.2.12 indicates that majority of the managers are willing to fund for training TDD and educate their staff to help them to changeover to TDD. All the participants widely agreed that a strong buy-in is needed from management to implement TDD. Some participants mentioned that if the management team is technical enough to understand the benefits of TDD then it becomes easier for the buy-in. Koskela (2008:443) frames the term “managing upward” which means that the change we are leading must be managed and pushed upward otherwise there is a risk of failure resulting from the lack of understanding from the management. Also managing upward is not easy as it is related with position and power and also strongly depends on the culture of the company.

4.3.2 Better Code Quality

The case study revealed seven basic themes for the organising theme "Better Code Quality". The diagrammatic representation for this theme is shown in figure 56.

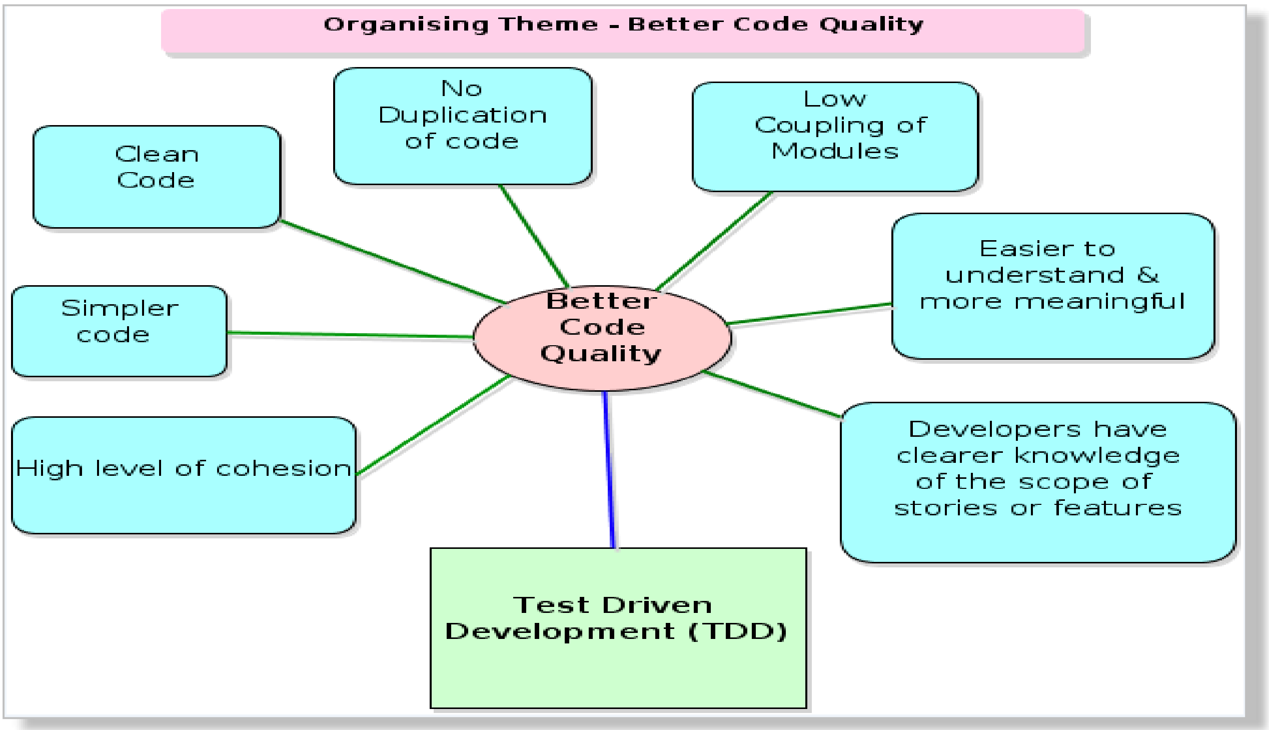

Organising Theme - Better Code Quality
No Duplication of code
Low Coupling of Modules
Clean Code
Easier to understand & more meaningful
Simpler code
Better Code Quality
High level of cohesion
Developers have clearer knowledge of the scope of stories or features
Test Driven Development (TDD)

Figure 56: Organising theme for "Better Code Quality" with its basis themes and link to Global Theme. (Source: Self-generated)

4.3.2.1 Clean Code

All the participants agreed that TDD promotes clean code. Williams et al. (2003) investigations in the industry revels that 92% of developers felt that TDD yielded high-quality code. Bergmann et al. (2011: 8) states that clean code can be read and enhanced by a developer other than its original author. Clean code has unit and acceptance tests and has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined and provides a clear and minimal API. Few participants stated that if you care about the code you are putting in then you won't leave it in a messy state. Crispin (2006:70) does provide supporting evidence for cleaner code with TDD in software development projects.

4.3.2.2 Simpler Code

Davis et al. (2009:46) argues that a simpler code structure is easier to test, can be self-defining, is smaller to store, and is easier to maintain. Where as clever code may only create future maintenance problems due to the code's length and complexity. Every one of the participants agreed that simpler code is easy to read, understand, modify, and maintain. Some of the participants also mentioned about the KISS and YAGNI principles that remind the developer to keep the code simple and to not use unnecessary code if you are not going to use it. These principles also help to keep the developer away from introducing complexity.

4.3.2.3 No Duplication of Code

Duplication is the primary enemy of a well-designed system as it represents additional work, additional risk, and additional unnecessary complexity. Creating a clean system requires the will to eliminate duplication (Martin, 2008:173). Majority of the participants experienced that continuous refactoring encourages removal of code duplicates and for the emergence of a good design. Some of the participants recommended that restricting the function to exhibit single behaviour, keeps the method simple, and avoids introducing duplicates. Tahchiev et al. (2003:96) claim that the use of TDD is effective at producing cleaner code and also that it "eliminates duplication of code".

4.3.2.4 Low Coupling of Code

Rathore (2001: 169) points out that code developed using TDD must be testable and subsequently the resulting design is often better in terms of low coupling as well. Galloway (2010:413) states that testability is also a form of verification that your code is easy to change and it's not brittle. These positive traits ensures loose coupling of the code. The participants indicated that low coupling of code was easy to change, reusable, and adaptable. One participant mentioned that the down side of low coupling is that the code has limited knowledge about its dependencies and so can introduce many indirections. Janzen et al. (2008:77) also demonstrated that TDD is better in producing simple, loosely coupled and highly cohesive code.

4.3.2.5 Easy Understanding and Meaningful Code

Participants agreed that with TDD you write more meaningful code that makes it easy to understand not only by the author but also other developers who inherit the code later on during maintenance. Code should be written to minimize the time it would take for someone else to understand it. Someone to fully understand your code, they should be able to make changes to it, spot bugs, and understand how it interacts with the rest of your code (Boswell et al., 2011:3). One participant stated that in any project understanding the code takes majority of the time when making a change, so by having unit tests the behaviour and purpose of the code existence is very clear. TDD has a live knowledge store in the form of unit tests that make the code east to understand

4.3.2.6 Clear Understanding of the Scope or Features

Williams et al. (2003) from his investigations found that 87.5% of developers reported better requirements understanding. Participants agreed that the method or the function is verified by the unit tests every time the developer touches the code. Ambler (2008) in his survey found out that TDD actually improved 51.5% of the chance of keeping the specification in sync with the code, as shown in section 4.2.8. Participants also found that when writing tests the developer exactly knew the behaviour and interface of the functions. So clear understanding and scope of the feature is a must before writing the test. Crispin (2006:70) indicated that TDD leads to the better requirement understanding and this consequently improves the correctness of the software.

4.3.2.7 High level of cohesion

Hilton (2009:54) experiments found that TDD improved the level of cohesion by about 21%. Tahchiev et al. (2011:79) argues that with TDD when you eliminate duplication, you tend to increase cohesion and decrease dependency. Participants agreed that they noted a high level of cohesion in their code using TDD and stated that the no code is implemented without testing so the code relation has to exist.

4.3.3 Better Application Quality

The research exposes nine basic themes for the organizing theme "Better Application Quality" is shown in figure 57.

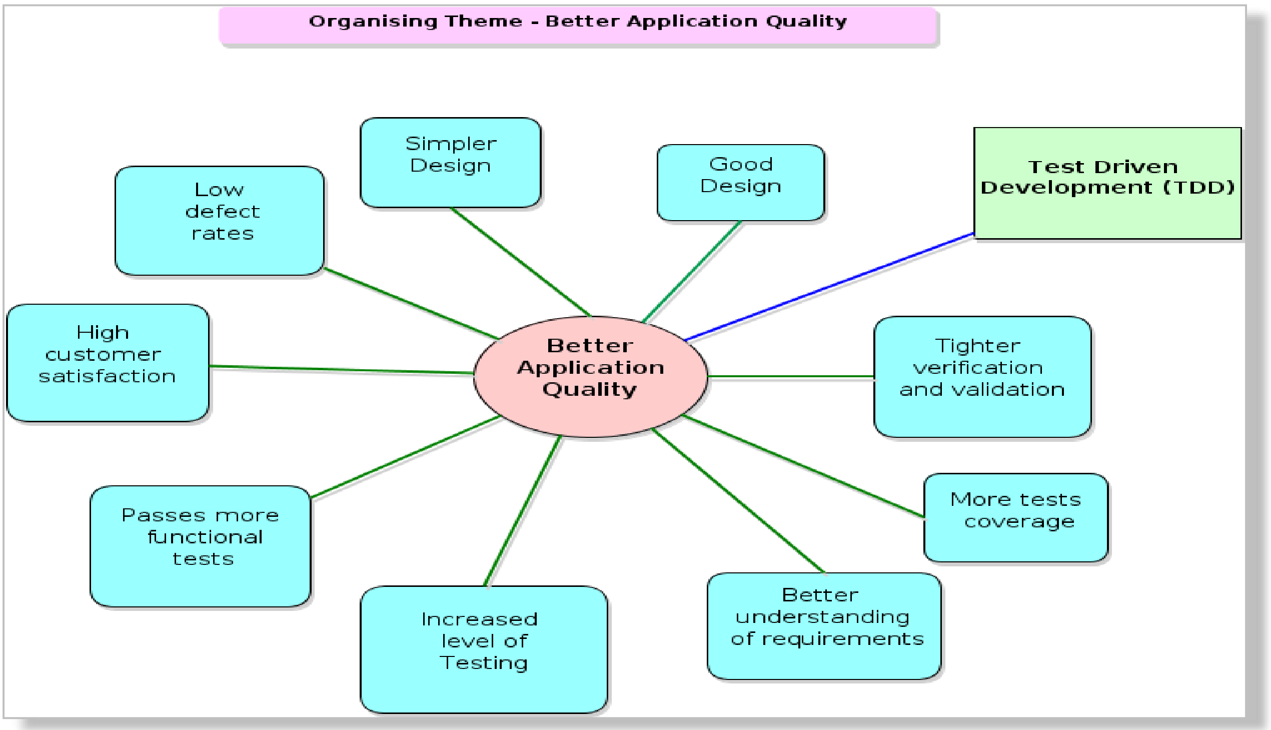

Organising Theme - Better Application Quality
Simpler Design
Good Design
Test Driven Development (TDD)
Low defect rates
High customer satisfaction
Better Application Quality
Tighter verification and validation
More tests coverage
Passes more functional tests
Increased level of Testing
Better understanding of requirements

Figure 57: Organising theme "Better Application Quality" with its basis themes and link to Global Theme. (Source: Self-generated)

4.3.3.1 Simpler Design

The principle of KISS and YAGNI discussed by the participants also apply here. Participants also stated that simple modules with clearly defined interfaces that interact with each of the other components make it simple design. Participants added that TDD makes it more extensible and makes it easier to maintain due to simple design. Williams et al. (2003) from his investigations found 79% of developers believed TDD promoted simpler design. Oram (2010:209) state that though TDD is considered as a design practice and simple design is expected to emerge when using the TDD principles. Simple design is achieved using TDD principles (Janzen et al, 2008:78).

4.3.3.2 Good Design

Pilone et al. (2008:287) points out that the critical part of getting to a good design is the refactoring step in TDD. Good designed classes are singularly focused and the don’t repeat yourself (DRY) principle is about having each piece of information and behaviour in your system in a single, sensible place (Pilone et al., 2008:163,153). Hayder (2007:129) argues that with TDD you have to foresee your application code before it is actually written which also helps you to design well, API and good code. Most of the participants agreed that TDD promotes good design. Some experienced participants pointed out that the developer must have good object oriented design skills to come up with a good design in TDD, as TDD is a not a design tool but it will guide you and lead you to good design.

4.3.3.3 Low Defect Rates

All the participants mentioned that they have experienced a significant drop of defects after implementing TDD. Most of the bugs are caught at the coding phase and the defects are fixed immediately by the developer. Bhat et al. (2006:238) experimentation revelled that there was a tremendous increasing the application quality due to reduced defects. Oram (2010:209) report also supports the fact that the reduction of bugs due to TDD is remarkable. Participants pointed out that TDD helps to implement the method or functionality using less code and less code means fewer chances of bugs. With TDD no production code is checked in with out testing which leads to reduced bug count. Some experienced participants pointed out that using tested libraries as much as possible will reduces bugs.

4.3.3.4 High Customer Satisfaction

Blankenship (2011:30) states that TDD provides feedback which is valuable when demoing working code to a customer. The customer’s feedback can be integrated into the code right away or to the next iteration. So the overall application is reviewed at the end of iteration by the customer that leads to high customer satisfaction. Participants mentioned that involving customers early in the project has a good customer satisfaction effect as the customer is involved and sees the working piece of code and thereby giving the customer a chance for the feedback. Participants also indicated that the developer not only thinks of the interface also about the software's behaviour from the customer’s point of view. The research from Crispin (2006:70) also shows that TDD can extend beyond unit test to customer facing tests that can contribute to more customer satisfaction.

4.3.3.5 Passes more Functional Tests

Participants indicate that with TDD you write good solid code that is tested and can direct to 100% code coverage. Crispin (2006:70) in her research points out that TDD developer wrote code that passed 18 percent more functional test cases. George et al. (2005:337) states that the developers using TDD produced code of a higher external quality that passed 18% more functional black box tests as compared to the codes developed by using the traditional test last approach. Ambler (2008) survey result shown in section 4.2.14 signify that 81.1% of developers write test before writing production code that also contributes to pass more functional tests.

4.3.3.6 Better Understanding of Requirements

Garofalo (2011:209) points out that with TDD process each cycle consists of a test case that defines the code requirements and the code that will satisfy these requirements. So you need to have a better understanding of the requirements before writing the test cases. Ambler (2008) survey result shown in section 4.2.15 indicates that 34.7% have customer acceptance test that promote better understanding of the requirements. Most of the participants mentioned that TDD forces you to think and act and you can’t think if you don’t have a good understanding of the requirements, the expected behaviour, and the needed interface for the method. Participants argued that if you are facing difficult in writing the test then either the requirements are not clear or the requirement is too large that it is time to break the requirements down to smaller pieces. Because the developer has already spent time thinking about the use case of how the feature is supposed to work, a better understanding of the requirements is already achieved that contributes to better implementation (Galloway, 2010:165).

4.3.3.7 More Tests Coverage

Participants agreed that the more the tests cover all the modules the more robust the application will be. Continuous integration also makes sure that more test coverage is achieved. Some participants commented that TDD tests can test and cover multiple classes. Bhat et al. (2006:293) observed in their case studies that the use of TDD contributes to more test coverage thereby lowering defect density. Cauldwell (2008:11) states that the more tests you have, the easier it is to practice refactoring and incremental changes to your code to improve its structure. Proper refactoring involves making very small organizational changes to your code, then making sure that your tests still pass. Without proper test coverage, refactoring is much more difficult.

4.3.3.8 Tighter Verification and Validation

Participants argued that the granularity of TDD encourages more frequent and tighter verification and validation. One of the participant mentioned that with TDD there is more communication with the business users and this improved communication allows verifying the requirements more thoroughly to write tests that can validate the methods more tightly. Dietrich et al. (2005:5) mentions that in TDD test cases are not only useful to validate but also to verify the correctness of the business rules.

4.3.3.9 Increased Level of Testing

Majority of the participants agreed that with TDD, it is impossible to write code without having that code covered by a test. With test first development you are not allowed to write any production code unless it is to make a failing unit test pass. (George et al., 2004:341) points out that TDD has the potential of increasing the level of testing and contributes to higher quality software. Scott (2003:171) states that an interesting side effect of TDD is that you achieve 100 percent coverage test and every single line of code is tested. The traditional testing doesn't guarantee this type of coverage. Therefore, TDD results in significantly better code testing and ensures increased level of testing.

4.3.4 Better Developer Productivity

The research revealed five basic themes for the organizing theme "Better Developer Productivity" which is shown in figure 58.

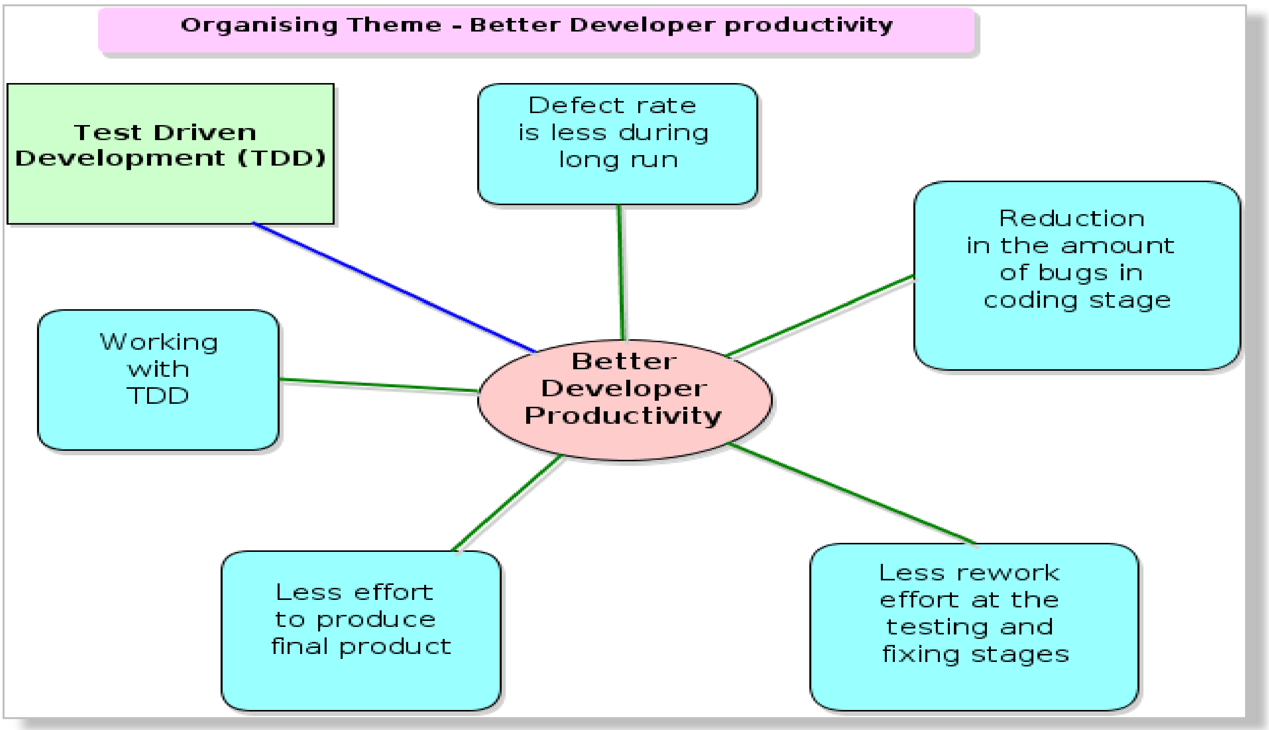

Organising Theme - Better Developer productivity
Test Driven Development (TDD)
Defect rate is less during long run
Reduction in the amount of bugs in coding stage
Working with TDD
Better Developer Productivity
Less effort to produce final product
Less rework effort at the testing and fixing stages

Figure 58: Organising theme "Better Developer Productivity" with its basis themes and link to Global Theme. (Source: Self-generated)

4.3.4.1 Working with TDD

Williams et al. (2003:4) investigation highlights that 78% of developers reported TDD improved overall productivity. Oram (2010:213) argues that the productivity dimension engenders the most controversial discussion of TDD. Although many admit that adopting TDD may require a steep learning curve that may decrease the productivity initially, there is no consensus on the long-term effects. Many of the participants pointed out that working with TDD is fun and once you are comfortable with TDD the practice become a muscle memory. Some participants stated that developing applications keeping testability in mind leads to robust, easy to change, and resistant to defects architecture. Participants also mentioned that TDD gives them confidence as their design is right and also gives them confidence to refactor carefully or recklessly.

4.3.4.2 Defect Rate Less During Long Run

Williams et al. (2003:4) case study reported up to a 40% reduction in defect density for the TDD group compared to the other group. Bhat et al. (2006:293), observed a similar result that the use of TDD reduces the number of defects. Ambler (2008) survey result in section 4.2.7 shows that 92.5% found that TDD has the potential for significant quality improvement due to less defect rate and during the long run the maintenance of the code becomes easy. All participants agreed that the defect rate diminishes drastically over the period of implementing TDD. Participants mentioned that the changes made to the code with TDD are verifiable and the developer has to fix the bug code now and can't leave it for later. The automated unit test in TDD is more reliable than a person as it will run the test exactly the same way, every time. So the defects are picked up on every run and the time needed to run the unit tests is much faster than manual testing.

4.3.4.3 Reduction of Bugs in the Coding Stage

Helper (2009) points out that TDD is well known practiced methodology that reduces the amount of defects introduced during the coding stage of the development process by the developers. Williams et al. (2003:4) case study states that 95.8% of developers reported reduced debugging efforts due to reduction in bugs because of TDD. Participants mentioned that earlier finding and fixing of the bug cost less and saves time. Some of the experienced TDD practitioners pointed out a strong test suite with TDD during the development is always better off as the speed of development will be maintained even as the project complexity increases when compared to non-TDD team.

4.3.4.4 Less Rework Effort

Erdogmus et al. (2005:11) experiment with TDD revealed that the developers wrote more tests but were more productive in terms of overall development effort, specifically less rework effort at the testing and fixing stages. Koskela (2007:346) proves a good point that producing good quality code with TDD guarantees the sustainability of the developer's productivity, whereas bad quality only creates more rework and grinds the progress to a halt. Participants stated that the code implementation with TDD is long lasting with reduced defects which contribute to less rework effort. The granularity achieved with TDD makes it easy to work on a single lose coupled method without affecting the other methods or modules, and this saves rework on other modules.

4.3.4.5 Less Effort to Produce Final Product

Ambler (2008) survey results showed in section 4.2.7 and section 4.2.8 highlights that 92.5% TDD practitioners has experienced the potential for significant quality improvement, 98.1% experienced increased ability to safely change software and 98.1% observed increased ability to react to changing. These factors contribute towards less effort to produce the final product. Williams et al. (2003:4) case studies also reveal that 50% of developers found a decreased development time overall. Participants in the research experienced that TDD helps to get the best final design in fewer attempts, compared to traditional development. With refactoring and testing good solutions are derived at the first attempt most of the time. Practitioners of TDD spend less time in the debugger and they can put the code into production much more often without disrupting customers due to the tested code with TDD.

4.3.5 Disadvantages of TDD

The organizing theme "Disadvantages of TDD" shown in figure 59 exposed five basic themes.

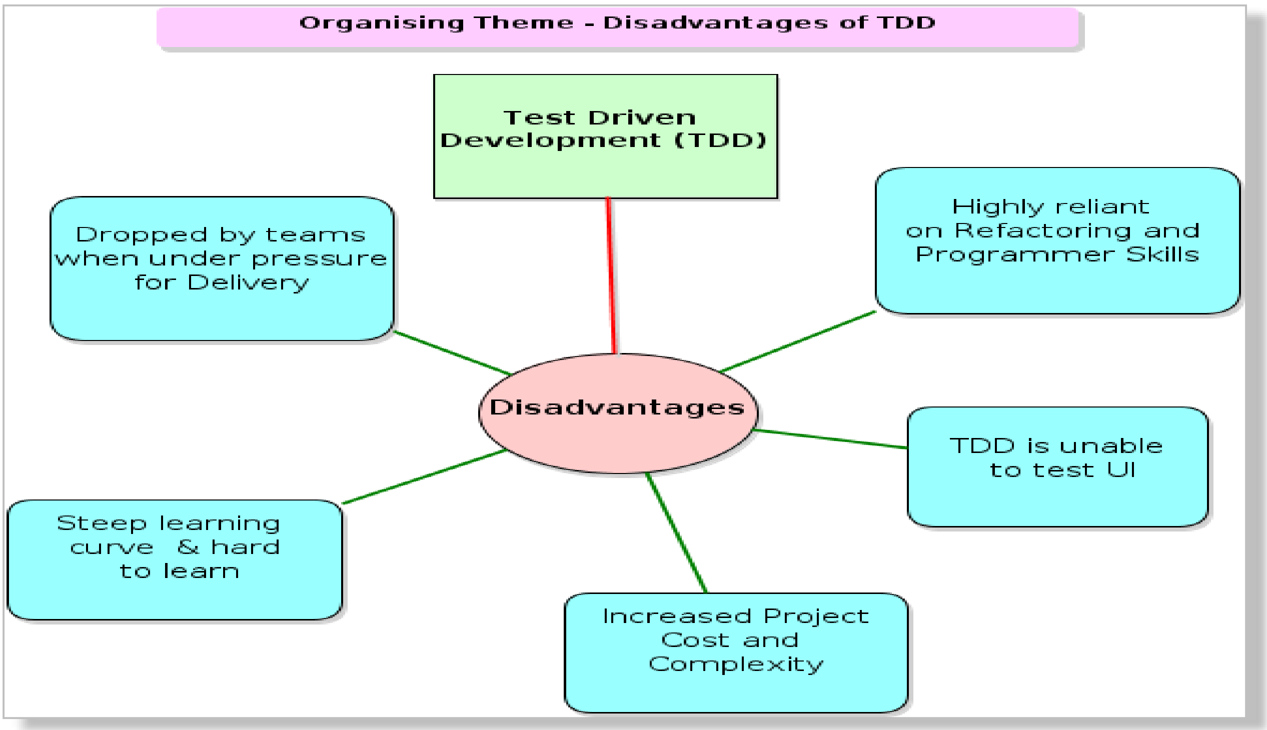

Organising Theme - Disadvantages of TDD
Test Driven Development (TDD)
Dropped by teams when under pressure for Delivery
Highly reliant on Refactoring and Programmer Skills
Disadvantages
TDD is unable to test UI
Steep learning curve & hard to learn
Increased Project Cost and Complexity

Figure 59: Organising theme "Disadvantages of TDD" with its basis themes and link to Global Theme. (Source: Self-generated)

4.3.5.1 Step Learning Curve

Participants agreed that TDD is hard to learn as the disciplined practice requires a mind shift from the traditional programming methodologies. Programming is also hard to learn especially on your own and so is TDD. Experienced participants commented that TDD makes you write code well and writing code well is hard. Ambler (2008) survey results shown in section 4.2.9 about the ease of learning Developer TDD indicated that 31.5% of the practitioners found it difficult learning TDD. Shore et al. (2008:287) mentioned that TDD even with green field systems it takes a few months of steady use to overcome the learning curve. Bender (2011:304) states that for a developer or development team learning TDD represents a steep learning curve that can make initial development much slower.

4.3.5.2 Dropped By Team When Under Pressure

Participants indicated that comprehensive testing is often dropped by teams when they are under pressure resulting in more bugs at the customer side and higher long term costs. A well-known issue in the software industry is that short term benefits are traded off for long term costs which lead to building of technical debts. Participants also mentioned that maintaining technical excellence in the heat of a delivery cycle is difficult for immature teams. Smart (2012) argues that sometimes it feels natural to drop the good habits you have painfully acquired and just hack under pressure. The counter argument by TDD practitioners is that, when you are under pressure to deliver, it is actually those good habits that allow you to progress at a fast, sustainable rate without being bogged down by slow feedback cycles and a continuous stream of obscure, difficult-to-diagnose and hard-to-fix bugs.

4.3.5.3 Increased Productivity Cost and Complexity

Participants objected to this disadvantage by arguing that during the initial few weeks the development with TDD is slower but by week 4-6 you are in the same speed with non-TDD team. When you compare the mid to longer term gains, then TDD yields 60% gains to speed and maintenance cost. Participants also agreed that once developer understand and trust TDD they begin to do less gold plating and crazy upfront design to ensure that the code is re-usable, flexible, and high. Maximilien et al. (2003:2) states that TDD is described as developing software in very short iterations with little upfront design. Oram (2010:213) controlled experiments suggests an improvement in productivity when TDD is used.

4.3.5.4 Unable to Test the User Interface

There was a mixed response from the participants. Some of them agreed with this fact and some practitioners disagreed with this fact saying that most of the time we choose not to test the UI, as this is a return on investment decision not a feasibility decision. Participants also state that the UI tests links to limitation of the particular technology so developers by pass UI test. It is easy to do TDD with loosely coupled UI architectures such as MVVM (Presentation Model) and MVC. An Alexander et al. (2001:178) point out that TDD has a challenge for graphical user interface (GUI) applications. The primary focus of TDD is functional tests and with an advanced GUI application, there is a portion of the application that is user driven and outside the scope of a functional test. Walsh (2006:45) also states that TDD works great on the internal aspects of an application but not suited for testing the user interface.

4.3.5.5 Highly Reliant on Refactoring and Programming Skills

Participants agreed with this fact that if developers cannot write good code they will not be able to do TDD. If developers can't master TDD then they should not be able to write good code. Experience developers mentioned that developers need design knowledge to do good refactoring. Bender (2011:307) advises that practice is important for building the skills required to be a productive TDD developer. Developers must master new technical skills such as dependency injection, MVC/MVVM, unit-testing frameworks, and supporting frameworks such as NBehave. Developers adopting the SOLID principles and practices can become comfortable with the idea of TDD development.

4.3.5.6 Database or External File Not Directly Tested

Most of the participants agreed to this disadvantage saying that it is hard to test external files or database. Participants indicated that tests must test at least some code down to your DB Layer. If there are no automated system or acceptance tests then manual test must be conducted which will cover the database tests. TDD tests a single class or method in isolation from its dependencies, therefore mocks and stubs help us isolate these codes under test and make our tests more focused and expressive. Good integration tests validate the features of the external system that you use in your application. McWherter (2010:4) points out that a unit tests is a method used to verify that a small unit of source code is working properly. Unit tests should be independent of external resources such as databases and files. Kent Beck's 10 minute rule forces the developer to keep the tests under 10 minutes, so testing databases or web servers will delay running of tests.

4.4 Conclusion

This chapter discussed the survey results, interview results, the basic themes that were dissected from the interview transcripts and correlated the results to the literature findings. Most of the advantages of TDD in the literature review were proved in practice, but only some of the disadvantages discussed in the literature review were found true in practice.

The next chapter which is the most import chapter reveals the conclusions and recommendations from this research.

Chapter 5 - Conclusions and Recommendations

5.1 Introduction

The aim of this chapter is to provide an overall picture of the research, its findings, and the way forward. The findings of the primary and secondary research will be reviewed to come to a final conclusion. Possible recommendations for the implementation of TDD studies will be discussed.

5.1.1 Findings from the Study

The literature review in chapter 2 will be discussed from the perspective of the research query.

1. The literature review in chapter 2 started from the definition of software, the software development process, and the software development life cycle. The discussions were followed by the comparison of the traditional waterfall software development methodology with the new agile software development methodologies.
2. The study looked in-depth into the concepts of TDD and the theories behind them. Sterling (2010:68) points out that TDD is about creating a supportable structure for imminent change and this is achieved by writing tests to drive the software design in an executable fashion.
3. The three laws of TDD by Sterling et al. (2010:68) states that:
   i. You must write a failing unit test before you write production code.
   ii. You must stop writing that unit test as soon as it fails; and not compiling is failing.
   iii. You must stop writing production code as soon as the currently failing test passes.
4. The three simple steps of refactoring using the red-green-refactor steps by Garofalo (2011:210) are:
   i. Write the test and the assertions and fail them (red).
   ii. Implement the code to pass the assertions (green).
   iii. Refactor the code and test it again.
5. Sterling et al. (2010:70) points out the benefits achieved by using the three simple laws of TDD which are:
   i. Reduction in debug time dramatically.
   ii. Significantly increase in flexibility of your system, allowing you to keep it clean.
   iii. Creation of suite documents that fully describe the low level behaviour of the system.
   iv. Creation of a system design that has extremely low coupling.

6. Hilton (2009:7) in his research found that by following TDD you achieve predictability, reliability, speed, confidence, scope limiting, manageability and automatic documentation.
7. The four key attributes of TDD by Janzen et al. (2005:43) and Erdogmus et al. (2005:226) were studied in the literature which were:
    i. Small unit focus.
    ii. Test-orientation.
    iii. Frequent regression testing and
    iv. Automated testing.
8. The findings also points out the skills need in TDD as suggested by Bhadauria (2009: 12) in his research.
9. The literature review also investigated the acclaimed benefits of practicing TDD which are:
    i. *Simpler Software Design* - Oram (2010:209) claims that an incremental and simple design is expected to emerge when using the TDD principles.
    ii. *Better Software Quality* - Bhat et al. (2006) case study and Oram (2010:209) clinical trials concluded that lower complexity, better cohesion, removal of duplication and reduced number of defects led to better internal and external software quality.
    iii. *Better Software Application Quality* – Research form Crispin (2006:70) and George et al. (2004:341) supported the claims of better application quality due to TDD. TDD leads to the better requirement understanding and this consequently improves the correctness of the software and produces high quality software products with fewer bugs.
    iv. *Developer Productivity* - Bulajic et al. (2012:174) from his experiment found that TDD increases the development time from 16% to 25%. But due to better requirements understanding and reduction in debugging time the overall productivity is high. Erdogmus et al. (2005:11) and Oram (2010:213) experiments also supported developer's productivity.
10. The research finally covered the software developer's paradigm shift problems as mentioned by Gold (2005:26) about the programmers thinking when switching over to TDD from the traditional software development methodology. Burns

5.1.2 Findings from the Secondary Research

1. Survey indicates that many practitioners are from North America with 49.6%, Europe 26.4% and then from South and Central America with 11.6%. The *popularity of TDD* in Sub Saharan Africa was last with only 0.8%.
2. The TDD survey participants are primarily from the *software industry* with 48.8%.
3. 79.4% of the participants were experienced with TDD. The *years of experience* with TDD were 41.7% participants with 4-6 years, 16.37% with 7-9 years, and 16.7% with 10-12 years.
4. 92.5% of the participants agreed that TDD has the potential for *significant quality improvement.*
5. 98.1% of the participants were *confident to make changes* to the code with TDD as it safe guarded the developers.
6. 72.8% of the participants indicated that TDD Increased ability *to react to changing stakeholder needs*.
7. 51.5% of the practitioners accepted that TDD improved the chance of keeping specifications in *sync with the code.*
8. 35.1% of the participants indicted that it is *difficult to learn* TDD and 11.7% agreed that it was very difficult, to learn TDD.
9. 73.4% of the participants mentioned that learning TDD, by discussing via *online forums* is the *most effective way* followed with 54.3%, who said that formal training and coaching is the effective way to learn and implement TDD.
10. 48.0% participants mentioned that *lack of training* and education as the potential challenge to get on with TDD. 43.9% indicated that *lack of regression test suits* for existing products could be a hurdle.
11. 52.0% of the participants pointed out that *management are keen to fund training* for TDD and 31.6% people indicated that management wants to *adopt TDD practice without providing resources*.
12. *Developer regression* test topped the survey for other forms of test the team is practicing with 59.5%. Reviews and inspection of the produced followed next with 52.4%. *Test-last by QA teams* was last with 45.2%.
13. 81.1% of the practitioners agreed that *developers write tests before writing production code*. 56.7% also agreed that *developers write test after writing production code*. 36.7% participants indicated that testers write test for developers *after developers write the production code*.

5.1.3 Findings from the Primary Research

The primary research provided very informative data regarding the claims of the disadvantages of TDD in the industry and the effort needed to move to TDD, along with the research conducted on the benefits of TDD. The factors that contribute to each benefit are discussed below.

5.1.3.1 Moving to TDD

1. *Paradigm shift* - Moving to TDD is challenging and the experience of throwing away old habits to learn the new TDD techniques that is well disciplined is tough. Instead of thinking about algorithm and structure the programmer has to conceptualize the situation more from the perspective of the end user.
2. *Training, Self-learning and Pair/Peer programming* – Formal training and mentoring by an experienced TDD practitioner is the most effective way to learn and practice TDD. Unlearning of the accumulated habits in traditional programming takes longer to offload than learning TDD.
3. *Initial Productivity Decrease* - Measurement of productivity cannot be accurate due to unclear definition of metrics measured. The number of features released in a sprint is less due to learning curve, but the quality and design of the code are high right from the beginning.
4. *Buy-in from the Management* - Strong buy-in is needed from management. Managing upward change or managing together is needed from management and the development team.

5.1.3.2 Better Code Quality

1. *Clean Code* - Clean code can be read and enhanced by a developer other than its original author.
2. *Simpler Code* - The principle of KISS and YAGNI are some of the principles that lead to simpler design. Simpler design can make the code more extensible and makes the maintenance easier.
3. *No Duplication of Code* - Continuous refactoring encourages removal of code duplicates and helps the emergence of a good design.
4. *Low Coupling of Code* - Low coupling of code is easy to change, reusable, and adaptable. The down side is that low coupling code has limited knowledge about its dependencies.
5. *Easy Understanding and Meaningful Code* - TDD has a live knowledge store in the form of unit tests that make the code easy to understand. With TDD you write expressive code that is only needed for the functionality, which makes code meaningful.
6. *Clear Understanding of the Scope or Features* – Writing tests before production code helps the developer to exactly know the behaviour and interface of the functions.

5.1.3.3 Better Application Quality

1. *Simpler Design* - The principle of KISS and YAGNI help to keep the developer away from introducing complexity.
2. *Good Design* – Refactoring steps and keeping the classes singularly focused, following DRY principle and having each piece of information and behaviour in your system in a single, sensible place promotes good design.
3. *Low Defect Rates* – Most of the bugs are caught and fixed at the development stage. With TDD you can implement functionality with less code. So less code leads to fewer chances for bugs.
4. *High Customer Satisfaction* – Short iterations and workable releases allows getting immediate feedback from customers, keeping customers in loop leads to higher customer satisfaction.
5. *Better Understanding of Requirements* – TDD forces you to think and act and you can’t think if you don’t have a good understanding of the requirements, the expected behaviour, and the needed interface for the method.
6. *More Test Coverage* – Continuous integration and TDD tests spanning multiple classes contributes to more test coverage.

7. *Tighter validation and verification* - Granularity encourages more frequent and tighter verification and validation and more communication with the business users allows verifying the requirements more thoroughly for writing tests.
8. *Increased Level of Testing* - It is impossible to write code without having that code covered by a test. Tests should be written, passed, and built into an automated testing framework before a story can be considered done.

5.1.3.4 Better Developer Productivity

1. *Working with TDD* - TDD is fun to work with and developing applications keeping testability in mind leads to robust, easy to change, and resistant to defects architecture. TDD gives the developer the confidence to make changes to the code.
2. *Defect Rate Less During Long Run* - Changes made to the code with TDD are verifiable and the automated unit test in TDD is more reliable than manual testing.
3. *Reduction of Bugs in Coding Stage* - Earlier finding and fixing of the bug cost less and saves time. Bugs are fixed and killed at the same stage where it originates.
4. *Less Rework Effort* – Code implementation with TDD is long lasting with reduced defects along with granularity contribute to less rework effort.
5. *Less Effort to Produce Final Product* - TDD helps to get the best final design in fewer attempts and the developers spend less time debugging due to refactoring and testing good solutions. Best design consumes less effort to add new features.

5.1.3.5 Disadvantages of TDD

1. *Steep Learning Curve* – Learning TDD required mind shift and discipline, so upfront investment is more.
2. *Dropped by Teams under Pressure* - Maintaining technical excellence in the heat of a delivery cycle is difficult for immature teams. But continuing with good habits under pressure allows faster delivery.
3. *Increased Productivity Cost and Complexity* – Initial productivity cost is high but mid and longer terms are 60% benefit with speed and maintenance cost.
4. *Unable to Test the User Interface* - UI tests links to limitation of the particular technology so developers by pass UI test. Easy to do TDD with loosely coupled UI architectures such as MVVM (Presentation Model) and MVC.

5. *Highly Reliant on Refactoring and Programming Skills* - Developers need design knowledge to do good refactoring. Developers adopting the SOLID principles and practices can become comfortable with TDD.
6. *Database or External File Not Directly Tested* – It’s hard to test external files or database. TDD tests a single class or method in isolation from its dependencies, so good integration tests validate the features of the external system.

5.1.4 Conclusion

The aim of this research was to find the effectiveness and benefits of TDD in software development process. The study of the analysed data collected from the primary research and secondary research strongly supports the benefits achieved on code quality, application quality, and developer’s productivity. Even though implementing TDD has high initial investment cost and it is difficult to learn which are very short term, the mid and the long terms benefits of TDD on projects is very high. The objectives of this research and the results had provided the necessary data to make a scientific decision on whether to move over to TDD or not. The research greatly recommends that implementing TDD will be beneficial to any software development organization.

5.2 Recommendations

From the research study and information collected during the course of this dissertation, the following recommendations are made trusting that they could make a contribution to the implementation of TDD as a software development methodology at company XYZ.

1. Awareness of the benefits of TDD and the changes that each team members have to make must be presented to the whole development team, including the stake holders and the product owners.
2. The existing software development methods and process must be evaluated and weighed against TDD. Plan for the needed changes must be drafted along with the short, mid and long terms roadmap.
3. Test Driven Development can be very hard to learn. Initial training and then coaching with an experienced TDD practitioner within each team will make the move towards TDD faster. The industry figures show that it takes approximately 8 weeks to be comfortable with TDD, and 24-30 weeks to break even .i.e. to be where you were, pushing the same amount of features as before TDD.

4. Management support is essential, commitment from management to provide all the necessary technology, tools and time to implement TDD is a must. Management must define the productivity matrix clearly as the traditional measurements can’t apply to TDD. Productivity must be based on how quickly the team is able to deliver quality features.
5. The development team must commit to be disciplined and to stick with TDD all the time. Extra effort to help fellow team members is needed from each team member.
6. Check point on the progress with TDD in regular interval will keep everyone in loop.
7. TDD refers legacy code as code written without tests and absence of regression test suits for existing products could be a hurdle.
8. Management and the development team must agree upon what happens to the legacy code. If automated tests are available then it is an advantage as we can start test first development going forward.
9. Code coverage in TDD is a must. The acceptable level of code coverage must be decided.
10. Code review in the initial stage is a must and should be carried on going forward.
11. Good practice and implementation of TDD in pairs or groups makes learning fast and easier.
12. Regular group discussions on the TDD regarding the mistakes, corrections, and experience will fasten up the learning and switch over.
13. One responsible person to monitor the TDD progress, and for removing the impediments from the teams and to associate with the management to provide whatever is needed to keep the teams moving forward.
14. Implementation of TDD must start with one team and a small project, with an expert absorbed into the team guiding the developers. The end results after a sprint/iteration must be reviewed and any lessons learnt must be implemented and should be continued in the next sprint/ iteration until we are satisfied with the results before proceeding to the next team.
15. Software Quality measurement in every sprint/iteration must be in place, which is the key indicator for the success of TDD.

5.3 Conclusion

1. TDD is a very powerful and beneficial approach to developing software. TDD is a disciplined software development practice and when TDD is followed correctly, it offers many benefits to the whole development team and the organisation.
2. Developers can produce a software product with good design, more quickly, confidently and cost effectively, due to high developer's productivity.
3. With TDD the software product has high quality tested code that leaves very little room for bugs, so the overall application quality ends up high.
4. Customer's involvement after every iteration and high satisfaction are achieved with TDD.
5. The only noted down sides of TDD are steep learning before implementing TDD and the non-testability of database layers and external files.
6. TDD does not replace end-to-testing or acceptance testing.
7. Benefits of TDD are not experienced immediately. The benefits are experienced during mid and long terms of the project.
8. As TDD can accommodate quick changes more frequently and easily, TDD reduces the cost of maintenance of software products by 80%.
9. TDD shortens the time that is needed to deliver the product to the market.
10. Regression test being an integral part of TDD, projects of bigger size, multiple teams working on a single project and teams who are inter dependent will be more beneficial due to instant feedback if someone is breaking the code. The fixing of the code takes place immediately saving time.
11. TDD is a paradigm shift and will change the way developers develop software and it's a safety net for the developer to make change and implement ideas confidently and fearlessly.
12. TDD will keep any software development organisation and its product to be competitive in market.
13. TDD will challenge the developers to produce good software design and will keep the developers more satisfied with what they producing.
14. TDD is a design tool and not a testing tool that will drive the developer to a better software design.

5.4 Areas and Scope for Further Research

This research investigated the benefits of TDD from the literature with the experiences of the TDD practitioners in the software development industry and proved that the productivity of the software development and the quality of the software can be increased to a larger percentage by implementing the principles of TDD. But in reality there are other factors such as requirement specification, team size, environment, tools, skills etc. that can affect the productivity of the software development. The results from this research can contribute to any further research to investigate the above mentioned factors on TDD. The Behaviour Driven Development (BDD), which is a new methodology and the next level to TDD in the software development, is catching up the industry very fast. The findings and the results from this research can also be used to carry forward the investigations into BDD.

Bibliography

Agarwal, B., Gupta, M. and Tayal, S. P. (2009). Software Engineering and Testing: An Introduction (Computer Science). 1st Edition. London: Jones & Bartlett Publishers.

Alexander, B., Dillon, J. B. and Kim, K. Y. (2011). Pro iOS5 Tools: Xcode Instruments and Build Tools. Edition Unknown. New York: Apress.

Ambler, S. (2005). The Agile System Development Lifecycle. Ambysoft online article: http://www.ambysoft.com/essays/agileLifecycle.html . [Accessed 28 April 2012].

Ambler, S. (2008). Test Driven Development (TDD) Survey Results: October 2008 [online]. Available at: http://www.ambysoft.com/surveys/tdd2008.html . [Accessed 1 June 2012].

Ambler, S. W. and Lines, M. (2012). Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise. 1st Edition. Boston: Pearson Education.

Arthur, L. J. (1983). Programmer Productivity: Myths, Methods and Murphology - A Guide for Managers, Analysts and Programmers. Edition Unknown. John Wiley & Sons Inc.

Attride-Stirling, J. (2001). Qualitative Research Thematic networks: an analytic tool for qualitative research. Sage Journals. Vol. 1(3), p. 385-405.

Beck, K. (2001). Aim, Fire. IEEE Software 18(5), pp. 87-89.

Beck, K. (2003). Test-Driven Development by Example. 1st Edition. Addison-Wesley Professional.

Bender, J. and McWherter, J. (2011). PROFESSIONAL Test-Driven Development with C# - Developing real world applications with TDD. 1st Edition. Indiana: Wiley Publishing, Inc.

Bergmann, S. and Priebsch, S. (2011). Real-World Solutions for Developing High-Quality PHP Frameworks and Applications. 1st Edition. Indianapolis: Wiley Publishing, Inc.

Bhadauria, V. S. (2009). To test before or to test after - An experimental investigation of the impact of test driven development. Unpublished Doctorate of philosophy dissertation. Arlington: The University of Texas.

Bhatt, T. and N. Nagappan (2006). Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies. Available from: http://blogs.msdn.com/cfs-file.ashx/__key/communityserver-components-postattachments/00-01-07-76-74/isese_2D00_fp17288_2D00_Bhat.pdf [Accessed 11 February 2012].

Blankenship, J., Bussa, M. and Millett, S. (2011). Pro Agile .NET Development with Scrum. 1st Edition. Apress.

Boswell, D and Foucher, T. (2011). The Art of Readable Code. 1st Edition. California: O'Reilly Media, Inc.

Braun, V. and Clarke, V. (2006) Using thematic analysis in psychology. Qualitative Research in Psychology, 3 (2). pp. 77-101. [Online]. Available at: http://eprints.uwe.ac.uk/11735/ . [Accessed 10 July 2012].

Bulajic, A., Sambasivam, S. and Stojic, R. (2012). Overview of the Test Driven Development. Proceedings of Informing Science & IT Education Conference (InSITE) 2012. [Online]. Available from: http://proceedings.informingscience.org/InSITE2012/InSITE12p165-187Bulajic0052.pdf . [Accessed 10 July 2012].

Burns, R. (2000). Introduction to research methodology. 4th Edition. London: Sage publications.

Business Dictionaries. (2012). Skill definition [online]. Available from: http://www.businessdictionary.com/definition/skill.html . [Accessed: May 21, 2012].

Cannon, L. D. (2011). CISA Certified Information Systems Auditor Study Guide. 3rd edition. Indiana: Wiley Publications.

Cauldwell, P. (2008). Code Leader: Using People, Tools, and Processes to Build Successful Software (Programmer to Programmer). 1st Edition. Indiana: Wiley Publishing, Inc.

Chromatic. (2003). Extreme Programming Pocket Guide. 1st Edition. O'Reilly Media.

Collier, K. W. (2011). Agile Analytics: A Value-Driven Approach to Business Intelligence and Data Warehousing. 1st Edition. New Jersey: Addison-Wesley.

Condelli, L. and Wrigley, H. S. (2004). Real World Research: Combining Qualitative and Quantitative Research for Adult ESL [online]. Available at: www.leslla.org/files/resources/RealWorldResearch.doc . [Accessed 2 June 2012].

Crispin, L. (2006). Driving Software Quality: How Test-Driven Development Impacts Software Quality. IEEE Software 23(6), pp.70-71.

Davis, B. and Tuker, H. (2009). 97 Things Every Project Manager Should Know. 1st Edition. California: O’Reilly Media, Inc.

Denscombe, M. (2003) The Good Research Guide for small-scale social research projects. 2nd Edition. Berkshire: Open University Press.

Dietrich, J. and Paschke, A. (2005). On the Test-Driven Development and Validation of Business Rules. In: ISTA, Vol. 63GI (2005), p. 31-48.

Downey A. B., (2012). Think Complexity: Complexity Science and Computational Modelling. 1st Edition. O'Reilly Media.

El-Haik, B. and Shaout, A. (2010). Software Design for Six Sigma: A Roadmap for Excellence. 1st Edition. New Jersey: John Wiley & Sons.

Erdogmus, H., M. Morisio and M. Torchiano (2005). On the Effectiveness of the Test-First Approach to Programming. IEEE Transactions on Software Engineering 31 (3), pp. 226-237. Available from: http://nparc.cisti-icist.nrccnrc.gc.ca/npsi/ctrl?action=rtdoc&an=5763742&lang=fr [Accessed 12 March 2012].

Esposito, D. and Saltarello, A. (2010). Microsoft¨ .NET: Architecting Applications for the Enterprise. Washington: Microsoft Press.

Galloway, J., Hanselman, S., Haack, P., Guthrie, S. and Conery, R. (2010). 1st Edition. Indianapolis: Wiley Publishing, Inc.

Garofalo, R. (2011). Applied WPF 4 in Context. 1st Edition. Apress.

George, B., Williams, L. (2004). A structured experiment of test-driven development, Information & Software Technology, vol. 46, pp. 337-342.

Gold, R., Hammell, T. and Snyder, T. (2005). Test-Driven Development: A J2EE Example (Expert's Voice). 1st Edition. Apress.

Groenewald, T. (2004). A phenomenological research design illustrated. International Journal of Qualitative Methods [online]. Available at: http://www.ualberta.ca/~iiqm/backissues/3_1/pdf/groenewald.pdf3(1) . Article 4. [Accessed 1 June 2012].

Gunvaldson, E., David, J.M. and Gousset, M. (2007). Installing Team Foundation Server. 1st Edition. Indianapolis: Wiley Publishing, Inc.

Gupta, A. and P. Jalote (2007). An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development. First International Symposium on Empirical Software Engineering and Measurement, pp. 285-294.

Hayder, H. (2007). Object-Oriented Programming with PHP5. 1st Edition. Birmingham: Packt Publishing Ltd.

Helper, D. (2009). Type Mock, the unit testing company: The cost of test driven development. Available from: http://www.typemock.com/blog/2009/03/05/the-cost-of-test-driven-development/ . [Accessed 20 April 2012].

Hilton, R. (2009). Quantitatively Evaluating Test-Driven Development by applying object-Oriented Quality Metrics to Open Source Projects. Unpublished Master of Science in Engineering thesis. Regis University.

Humphrey, W. S. and Thomas, W. R. (2010). Reflections on Management: How to Manage Your Software Projects, Your Teams, Your Boss, and Yourself. 1st Edition. Boston: Pearson Education, Inc.

Huttermann, M. (2011). Agile ALM: Lightweight tools and Agile strategies. Pap/Psc Edition. New York: Manning Publications.

IT Job watch. (2012).TDD Demand Trend. [Digital Image].Available from: http://www.itjobswatch.co.uk/jobs/uk/tdd.do [Accessed 27 July 2012].

Janzen, D. & Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction - Computer, vol. 38, no. 9, pp. 43-50.

Janzen, D. and H. Saiedian (2008). Does Test-Driven Development Really Improve Software Design Quality? IEEE Software 25 (2), pp. 77 – 84.

Jeffries, L. R. (2001). What is extreme programming? [Online]. Available from: http://xprogramming.com/book/whatisxp/. [Accessed 10 May 2012].

Jeffries, R., Anderson, A. and Hendrickson, C. (2000). Extreme Programming Installed. 1st Edition. Addison-Wesley Professional.

Jones, C. and Bonsignour, O. (2011). The Economics of Software Quality, Video Enhanced Edition. 1st Edition. New Jersey: Addision-Wesley.

Koskela, L. (2007). Test Driven: Practical TDD and Acceptance TDD for Java Developers. 1st Edition. Greenwich: Manning Publications Co.

Koskela, L. (2008). Test Driven: Practical TDD and Acceptance TDD for java developers. Edition Unknown. Greenwich: Manning Publications Co.

Krishnan, M.S., Kriebel, C.H., Kekre, S. & Mukhopadhyay, T. (2000). An Empirical Analysis of Productivity and Quality in Software Products, vol. 46, no. 6, pp. 745-56.

Krishnaswamy, K. N., Sivakumar, A. I. and Mathirajan, M. (2006). Management research methodology: Integreating of principals, methods and techniques. First impression. New Delhi: Dorling Kindersley (India) Pvt. Ltd.

Kruger, E. R. (2011).Top Market Strategy: Applying the 80/20 Rule. 1st Edition. New York: Business Expert Press.

Kuhn, D.R., Wallace, D.R. & Gallo Jr., A.M. (2004). Software Fault Interactions and Implications for Software Testing, IEEE Transactions on Software Engineering, vol. 30, no. 6, pp. 418-421.

Lui, K. M. and K. C. C. Chan. (2007). Software Development Rhythms: Harmonizing Agile Practices for Synergy. New Jersey: John Wiley & Sons, Inc.

Lyons, N. and Wilker, M. (2012). Interactive Project Management: Pixels, People, and Process (Voices That Matter). 1st Edition. California: New Riders.

Maliandi, G. L. (2010). Red/Green/Refactor: A pattern for clean code that works. [Digital image]. Available at: http://blogs.southworks.net/gmaliandi/2010/02/redgreenrefactor-a-pattern-for-clean-code-that-works-part-i-what-why/ [Accessed 27 May 2012].

Marchewka, J. T. (2012). Information Technology Project Management: Providing Measurable Organizational Value. 4th Edition. New Jersey: John Wiley & Sons.

Marshall, C. and Rossman, G. B. (2010). Designing Qualitative Research. 5th Edition. California: Sage Publications Inc.

Martin, R. C. (2009). Clean Code: A Handbook of Agile Software Craftsmanship. Edition Unknown. New Jersey: Prentice Hall.

Maximilien, E. M., Williams, L. and Vouk, M. (2003). Test-Driven Development as a Defect-Reduction Practice. Proceedings of the 14th International Symposium on Software Reliability Engineering [online]. Available from: http://maximilien.org/publications/papers/2003/Williams+Maximilien+Vouk03.pdf. [Accessed 10 March 2012].

McWherter, J and Hall, B. (2010). Testing ASP. NET Web Applications. 1st Edition. Indiana: Wiley Publishing, Inc.

Mishra, J. and Mohanty, A. (2011). Software engineering. 1st Impression. New Delhi:Pearson Education India.

Muller, M., & Hagner, O. (2002). Experiment about test-firrst programming. Software, IEE Proceedings- [see also Software Engineering, IEE Proceedings], 149 (5) pp, 131-136.

Nachiappan, N., & Maximilien, M. E., Bhat, T. & Williams, L. (2008). Realizing quality improvement through test driven development: results and experiences of four industrial teams. Available from: http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf [Accessed 13 February 2012].

Oram, A. and Wilson, G. (2010). Making Software - What Really Works, and Why We Believe It. 1st Edition. Sebastopol, CA: O’Reilly Media, Inc.

Oxford Dictionaries. (2012). Skill definition. [Online]. Available from: http://oxforddictionaries.com/definition/skill . [Accessed: 20 May, 2012].

Pilone. D, and Miles, R. (2008). Head First Software Development. 1st Edition. California: O’Reilly Media, Inc.

Rathore, A. (2011). Clojure in Action. 1st Edition. New York: Manning Publications Co.

Robson, C. (2002). Real World Research: A Resource for Social Scientists and Practitioner-Researchers (Regional surveys of the world). 2nd Edition. Oxford: Blackwell.

Roodyn, N. (2004). Extreme .NET: Introducing extreme programming techniques to .NET Developers. 1st Edition. New Jersey: Pearson Education, Inc.

Shajahan, J. (2005). Research methods for management. 3rd Edition. Mumbai: Jaico publishing house.

Saunders, M., Lewis, P. and Thornhill, A. (2007). Research methods for business students. 4th Edition. Essex: Pearson education limited.

Schmidkonz, C., and Staader, J. (2007). Piloting of Test Driven Development in Combination with Scrum: An experience report. [Online]. Available at: http://www.scrumalliance.org/resource_download/267. [Accessed 1 July 2012].

Scott, A. (2003). Agile Database Techniques: Effective Strategies for the Agile Software Developer. 1st Edition. Indianapolis: Wiley Publishing, Inc.

Scott, A. (2009). The Agile Scaling Model (ASM): Adapting Agile Methods for Complex Environments. [Online]. Available from: ftp://ftp.software.ibm.com/common/ssi/sa/wh/n/raw14204usen/RAW14204USEN.PDF . [Accessed 1 June 2012].

Shore, J. and Warden, S. (2008). The Art of Agile Development. 1st Edition. California: O'Reilly Media, Inc.

Simply Hired. (2012). Test Driven Development Job Trends. [Digital Image].Available from: http://www.simplyhired.com/a/jobtrends/trend/q-test+driven+development .[Accessed 27 July 2012].

Smart, J. F. (2012). Integration Test Driven Development - the Vietnam of TDD. [Online]. Available from http://weblogs.java.net/blog/johnsmart/archive/2011/05/15/integration-test-driven-development-vietnam-tdd . [Accessed 3 July 2012].

Spinellis, D. (2006). Code Quality: The Open Source Perspective. 1st Edition. Upper Saddle River, NJ: Addison-Wesley.

Sterling, C and Barton, B. (2010). Managing software debt: Building for inevitable change. 1st Edition. New Jersey: Addision-Wesley.

Tahchiev, P., Leme, F., Massol, V. and Gregory, G. (2003). JUnit in Action. 2nd Edition. Greenwich: Manning Publications.

Walliman, N. (2011). RESEARCH METHODS: THE BASICS. 1st Edition. Oxon: Routledge.

Walsh, B. (2006). Micro-ISV: From Vision to Reality. 1st Edition. Apress.

Williams, L., Maximilien, E. M. and Vouk, M. (2003). Test-Driven Development as a Defect-Reduction Practice. Proceedings of the 14th International Symposium on Software Reliability Engineering [online]. Available from: http://maximilien.org/publications/papers/2003/Williams+Maximilien+Vouk03.pdf [Accessed 10 March 2012].

Yin, R. K. (2006). Case study methods. New Jersey: Lawerence Erlbaum Associates, Inc.

Appendices

Appendix A - <u>Participant's Consent</u>

Research Title: Understanding the effectiveness and benefits of implementing Test Driven Development (TDD) in Software Development Process.

Research Supervisor: **Miss Trishana Ramluckan**

Researcher: **Yoganand Aiyadurai** (Student No: 111602)

1. I understand that the data collected from this interview will be used only for the purpose of MBA dissertation.
2. I had an opportunity to ask questions and to have them answered.
3. I understand that notes will be taken during the interviews and that they will also be audio-taped and transcribed.
4. I understand that I may withdraw myself or any information that I have provided for this project at any time prior to completion of data collection, without being disadvantaged in any way.
5. If I withdraw, I understand that all relevant information including tapes and transcripts, or parts thereof, will be destroyed.
6. I agree to take part in this research.
7. I am aware that my confidentiality will not be disclosed without my approval.
8. I wish to receive a copy of the report from the research (please tick one):

YES	NO

Participant's signature: ____

Participant's Name: Xyz ____

Participant's Contact Details: Phone - ________________________________

Email – ____

Signed Date: ____

Interviewed on Date: Time: **to** (SA Time)

Approved by MANCOSA (**www.mancosa.co.za**)

Note: The participant should retain a copy of this form.

Appendix B – Email Request for Participation in the Dissertation

Hi Xyz,

I am student at MANCOSA, Durban and I am currently doing my MBA dissertation on the following topic:

Understanding the effectiveness and benefits of implementing test driven development (TDD) in Software Development Process.

The claimed benefits of TDD which are crucial for this research are:

1. Simpler design.
2. Better code quality.
3. Better application quality.
4. Developer's productivity.

Drawbacks if any:

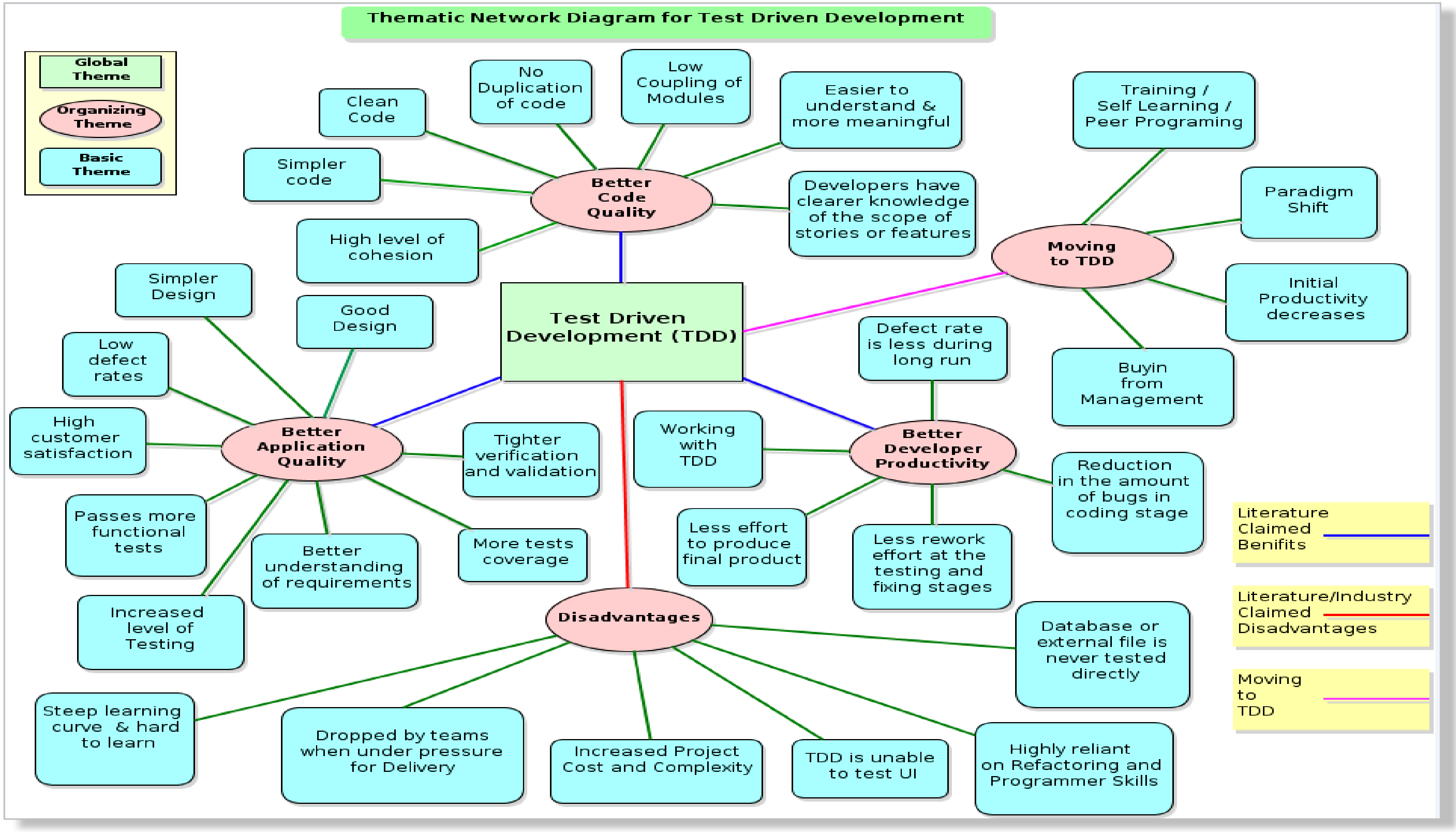
5. Any noted disadvantages using TDD.

I will be grateful to you if you can contribute your knowledge and experience to my dissertation, by giving me 30 minutes of your time for the interview. This would be a free flow interview covering the above benefits and you can interpret and clarify any doubts before or during the interview.

Please let me know if you are willing to participate and if so please let me know your preferred data and time.

Thanking you,
Yoganand Aiyadurai
Cell: +27 845667791
Work: +27 31 5801732
Email: yoganand.aiyadurai@derivco.com , y.aiyadurai@gmail.com

Appendix C - Full Thematic Network Diagram for Test Driven Development


Thematic Network Diagram for Test Driven Development
Global Theme
Organizing Theme
Basic Theme
Test Driven Development (TDD)
Better Code Quality
Clean Code
No Duplication of code
Low Coupling of Modules
Easier to understand & more meaningful
Simpler code
Developers have clearer knowledge of the scope of stories or features
High level of cohesion
Moving to TDD
Training / Self Learning / Peer Programing
Paradigm Shift
Initial Productivity decreases
Buyin from Management
Simpler Design
Good Design
Low defect rates
High customer satisfaction
Better Application Quality
Tighter verification and validation
Passes more functional tests
Better understanding of requirements
More tests coverage
Increased level of Testing
Defect rate is less during long run
Working with TDD
Better Developer Productivity
Reduction in the amount of bugs in coding stage
Less effort to produce final product
Less rework effort at the testing and fixing stages
Disadvantages
Database or external file is never tested directly
Steep learning curve & hard to learn
Dropped by teams when under pressure for Delivery
Increased Project Cost and Complexity
TDD is unable to test UI
Highly reliant on Refactoring and Programmer Skills
Literature Claimed Benifits
Literature/Industry Claimed Disadvantages
Moving to TDD

Appendix D - Codes to Themes for TDD learning (Moving to TDD)

Organising Theme - TDD Learning			
Basic Themes	**Codes**	**Issues/Advantages discussed**	**Themes identified**
TDD initiation	1. Team/organizational Effort.	Team Participation is a must.	1. All members of the team must be proficient with TDD to be successful.
	2. Huge learning curve.	Time Consuming.	2. Training and self-learn requires time.
	3. Preparation	Planning is required.	3. Developing using TODD is completely different from the traditional software development.
Buy in from Management	4. Push from the top to bottom.	Support in the Organization.	1. Buy in from Management is needed.
	5. Upfront cost.	Big investment initially.	2. The cost of implementing TDD is high in the beginning.
	6. Bad code &legacy software.	Cost a lot to maintain up to 90% of the total cost.	3. Management must realize this and prevent bad code upfront with TDD.
Paradigm Shift	7. Shift to TDD.	Change in mind set.	1. Every developer following TDD for the first time experiences Paradigm Shift, feel like going back.
	8. Trust	Trust unit test.	2. You have to trust the unit test more than yourself.
	9. Senior developers.	Difficulty to unlearn old methods, takes time.	3. Experienced programmers find it hard to bypass their traditional way of diving into writing code first.
	10. Fresh/Junior developers.	Easy to train/learn.	4. As they start blank, nothing to unlearn the old methods.
	11. Taking baby steps.	Small incremental development & Test.	5. Senior developer find this concept hard as they code everything and then test.
	12. Discipline.	TDD Must be followed by everyone	6. Development Practice changes as more focus is on unit test and you cannot trust your own code until proven not guilty.
	13. No production code.	Writing unit test first.	7. You can't write production code, until you have written the test and the test passes.

	14. Small releases.	potentially shippable	8. Develop, test, and checked in clean shippable small releases, compared to one big release.
Initial Productivity	15. Exercise and learning.	No. of features developed decrease in initial stages.	1. Productivity with respect to pushing out features decreases due to learning curve.
	16. More unit tests.	Slows down development.	2. Slows down development, but you think more about the function/method and clear implementation.
	17. Good unit test.	Takes time to master.	3. Good developers write good effective unit test, take time for average developer to master.
	18. Team acceptance.	All members must accept.	4. TDD is Effective only when all team members adopt and practice it otherwise it slows down and looses the effects.
Training / Learning TDD	19. Self-learning.	Hard and takes time depends on the person.	1. Time taken to learn also depends on person to person and their ability to learn new methodologies.
	20. Online materials/blogs.	Views from different people, real life experiences.	2. Can't adapt all as it is as it may or may not work for you. Eliminates errors repetition as it is already experienced by people.
	21. Pair/ Peer programming.	Easy way to learn.	3. Quicker 8-12 weeks learning, due to instant feedback from the trainer. You follow the right way.
	22. Formal training.	Is an Advantage.	4. Makes the learning process quicker as you gain prior knowledge before practice.
			5. Training in TDD encourages learning TDD onto the right and effective way.
	23. To be comfortable.	12-15 weeks.	6. To be comfortable to be working with TDD after training and coaching. To come close to the productive. Quality picks up.
	24. Proficient and implement.	28-32 weeks.	7. The team morale, productive is very close to pre TDD implementation. But high quality is achieved.

Appendix E - Codes to Themes for Better Code Quality

Organising Theme - Code Quality			
Basic Themes	**Codes**	**Issues/Advantages discussed**	**Themes identified**
TDD enables Clean Code	1. No unwanted code.	Write no production code other than to pass a failing test.	1. Minimum quality code to pass the test.
	2. Test only want is needed right now.	Write no more a test then it is sufficient to fail.	2. You unit test the method to check only the desired behavior.
	3. Minimal code	Easier to Read and modify later.	3. Implement simplest thing that could possibly work.
	4. Refactoring	Remove duplication.	4. Cleans code and removes duplicates.
	5. Simplify code	Red-Green-Refactor cycle.	5. You simplify your code all the time. Keeps you away from complexity.
	7. Caring	Code written by someone who cares.	6. If you really care about the code you write you won't leave it in a messy state but continue to refactor it until you are satisfied.
	8. Code smell indicator	Time to revisit and remove duplicates.	7. Clean up code when the test get longer and refactoring is possible.
Low Coupling	9. Keeps changeable.	Adaptability	1. Can accommodate quick changes.
	10. Production code.	Red-Green-refactor principle.	2. No production code is written without following Red-green-refactor principle
	11. Limited knowledge.	Modules have limited knowledge of its dependencies.	3. Refactoring and changes are easy to make.
	12. Refactoring	Part of TDD cycle.	4. Refactoring has more purposes that removing duplication
	13. Pros of low coupling.	Highly changeable.	5. Loosely couple modules are highly changeable.
	14. Cons of low coupling.	Difficult to understand.	6. Loosely coupled code leads to increased complexity.

	15. More in-depth investigations.	Introduce many indirections	7. You need to look down a level in order to see all that happens.
	16. Pieces of Modules.	Easy to swap	8. Modules or pieces of code are easy to swap/change when they aren't dependent on one another.
	17. Reuse, extensibility.	Modules are reusable	9. Independent modules are reusable and are easily extendable.
	18. Decoupled code.	Easy to understand	10. Code is decoupled which makes the code easier to understand as a unit and much safer to modify.
	19. Insured and protected.	Much safer to modify.	11. You are confident to make changes and are guaranteed to work without break other methods.
Simple Design	20. Test First Design.	Think and act.	1. Latest First Design forces you to really think about what you are going to do
	21. Writing test.	Stops adding extraneous capabilities	2. When a developer writes test then it really puts a damper on developer driven scope creep.
	22. Refactoring	Keep Safe.	3. Running all the tests makes refactoring safe.
	23. Clear code.	Benefit of tests.	4. Running all the tests makes refactoring safe.
	24. New smaller classes.	Removing duplicates.	5. Removing duplication leads to creation of new, smaller classes and methods.
	25. Meaningful names.	Expressing ideas.	6. Expressing ideas gives meaningful names to the classes and methods we have and to those that are created by the duplication rule.
	26. Refactoring	Breaking Complex code.	7. Always make use of refactor to make the complex code/method simpler leading to simple design.

	27. Communicate the design.	Developers understanding.	8. Having all the tests help make the code clear and communicate the design.
Clear Knowledge of Scope and features	28. Tighter verification.	Method is verified every time you touch the code.	1. This controls the feature scope and you work only with the needed requirements.
	29. Clear understanding.	Output and usage of the function.	2. When you write test you exactly know the behavior and interface of the functions
	30. Interface	Clear requirements and expected input/output.	3. You need to have all knowledge of the interface and the expected behavior, input, output of the method, if you want to write a good test.
No Duplication of Code	31. Refactoring	Remove duplication.	1. Cleans code and removes duplicates.
	32. Simplify code.	Red-Green-Refactor cycle	2. You simplify your code all the time.
	33. One behavior.	One behavior per function	3. Restricting the function to one behavior keeps the method simple and avoids introducing duplicates.
Easy understanding and more meaningful	34. Self-documentation.	Unit test for a method.	1. The unit tests explicitly show how the developer/programmer expected the code to be used
	35. Understanding the Code	Most time spend.	2. Writing new code 5%, Modifying existing code 25%, Understanding Code 70%. Saves understanding time.
	36. Knowledge store.	Actively updated.	3. Unit test runs all the time any failing test is updated and the tests are always live.
	37. Knowledge transfer.	Unit test code purpose.	4. Easy for a new developer to take over the project as unit test describes the behavior of the method and are self-explanatory.
	38. Statement & Hypothesis.	Unit test briefs.	5. Tests capture assumptions, theory, and subtleties.

	39. Easy accessible.	Unit test and production code in one project.	6. Capture the knowledge you put into building the system into a very accessible form at one place.
	40. Readability	Unit test is not only testing	7. Provides information about the responsibility of each element of code and the design of that code.
	41. Single source of truth.	The unit tests are the specifications.	8. Requirements can be written in the form of tests.
Good Design /quality	42. Good code structure and expressiveness.	Unit test names should reflect intent.	1. Unit Test names should be expressive.
	43. Unit Test.	Treat it as a technical specification	2. The design and the behavior of the production code should be explained by Unit test.
	44. Evolutionary development.	Design not static.	3. The design of the method or class always changes making it better.
	45. Unit Test contain.	Thoughtfully designed code.	4. Unit test proves its functionality at a unit level and the method gains better test coverage.
	46. TDD is Not about tests.	It’s about good design.	5. Your make your design better through good tests.
	47. Code coverage.	Code that is tested better.	Produce better quality code. Not just code that is tested better, but well-designed code.
	48. Collective ownership.	Responsibility	6. All developers are responsible for the integrity of the code.
	49. Interface/API changes	Behavior preserving.	7. All callers of that interface/API must change as well along with their tests.
	50. Keeping healthy.	Refactoring	8. When we refactor the code remains healthy.

High level of cohesion	51. Cohesive class.	Measurement	1. Strength of the association of variables and methods within a class.
	52. Few Tasks	Responsibility	2. Class is responsible for a few related logical tasks and so it has high cohesion.
	53. Methods and variables	Relationship	3. Methods and variables closely related.
	54. Methods	Functionality	4. Methods have only one function to perform.
	55. High cohesion	Understanding	5. Easier to understand what a class or method does.
	56. Cohesion classes	Reusable	6. Easier to reuse the methods and classes.

Appendix F - Codes to Themes for Better Application Quality

Organising Theme - Application Quality			
Basic Themes	**Codes**	**Issues/Advantages discussed**	**Themes identified**
Simple Design	1. KISS	Principles Used.	1. Keep It Simple Stupid.
	2. YAGNI	Principles Used.	2. You Aren't Going to Need It.
	3. Modular approach.	Individual components.	3. Simple modules with clearly defined interfaces that interact with each of the other components make it simple design.
	4. Testing earlier.	More bugs caught upfront.	4. Increase the application quality.
	5. Design and test.	How your class will be used by other classes.	5. TDD helps to flesh out a clean and efficient design for your classes.
	6. Improves design.	TDD makes it more flexible.	6. TDD makes it more extensible, it makes it easier to maintain.
Good Design	7. Good architecture.	Support & Benefit.	1. Testing, reusing, maintenance, and adaptability to changes that contributes to good design.
	8. Design activity.	Good design leads to loose coupling and highly cohesive code.	2. TDD assumes that the software design is either incomplete, or at least open to evolutionary changes.
	9. Individual components.	Passes through pre-defined tests.	3. Regression test are easy and all modules are tested, which increases application quality.
	10. Helping the team.	Understanding the features that the users want.	4. TDD improves the quality of the system dramatically especially the internal quality– How easy to understand, change the code base, simplicity, size and coupling.
	11. Recurring bugs.	Stops	5. With TDD when a defect is reported you have to write a new test to expose it. When this test passes and continues to pass the defect is gone for good.

	12. DRY	Don’t repeat yourself principle.	6. Avoid duplicate code by abstracting or separating out things that are common and placing those things in a single location. 7. DRY is about having each piece of information and behaviour in your system in a single, sensible place.
	13. Interactions and interfaces.	Developers are forced to think.	8. Developers think about their own objects and the objects they depend on and wiring them to test and code.
Low Defect Rates	14. Earlier detection.	Tests conducted from beginning.	1. Lows defects rates in QA and negligible defects in production.
	15. Direct lines of code reduction.	Due to refactoring.	2. Less code less defect.
	16. Unit tests.	More important than the code.	3. The code has to pass the unit test to be accepted. Bugs eliminated in the code.
	17. Code coverage.	All code covered by unit test.	4. Tested code in production eliminated bugs.
	18. TDD tests	Built-in regression testing.	5. Ensures that tomorrow's changes do not damage today's functionality.
	19. Libraries	Use of tested libraries.	6. Use of extensive exposure to tested libraries reduces bugs.
High Customer satisfaction	20. Continuous design.	Balance technical excellence with delivering value.	1. Keeps the customers highly satisfied with what they want and how the application behaves.
	21. Software’s behavior.	Correct Interface and behavior.	2. Programmer thinks of not only the interface but also about the software's behavior.
	22. Frequent visibility.	Features to the customer.	3. Early and frequent visibility of implemented features to the customer is highly likely to increase the probability that the software implemented.
	23. Concrete working.	Visibility to customers.	4. Customers always see concrete working software.

Passes more functional test	24. Optimum functionality Test.	Methodical nature of TDD.	1. Ensures that all the units in an application have been tested for optimum functionality, both individually and in synergy with one another.
	25. Code coverage data.	Proves effectiveness of the tests.	2. Aiming at 100% code coverage covers all functional tests.
	26. Business requirements.	Functional test.	3. Functional tests verify the business requirements and the specifications.
Better understanding of requirements	27. Think and act.	Test First Design forces you to really think about what you are going to do.	1. Implements good expected features as the requirements are clear.
	28. Documentation of design.	Test artifacts.	2. Leads to better understanding and smother maintenance.
	29. Expectations.	Concrete understanding.	3. Success of TDD include concrete understanding of the expectations of TDD,
	30. Code and business requirements.	High degree of fidelity.	4. Requirements are written as tests and if all tests pass then there is a high degree of confidence that the code meets the needs of the business.
	31. Shared language.	No Ambiguity.	5. System conforms to each particular aspect of the specification and fewer problems with ambiguity and less room for interpretation on behalf of developers.
More tests coverage	32. Improve robustness.	All modules are covered by test.	1. Changes to an application are easy because the tests will catch errors that such changes might introduce.
	33. Continuous integration.	Rebuilt and tested	2. Application is rebuilt and tested each time a change is made to the code base.
	34. TDD test.	Multiple classes.	3. TDD test can test and cover multiple classes.
	35. Regression testing.	Triggers off multiple tests.	4. TDD tests can be used to regression test to see the validity across the project.

Tighter verification and validation	36. Loosely coupled components.	Tested separately.	1. TDD encourages developing loosely coupled components, so they can be tested in isolation and integrated together.
	37. Business users.	Encourages communication.	2. To create these tests, you have to interact with the business users that lead to tighter verification and validation of business requirements.
	38. High quality of code.	Granularity of TDD.	3. Encourages more frequent and tighter verification and validation.
Increased Level of Testing	39. Defining test first.	Test the design	1. Defining tests first you get to test the design at a very granular level before the production code is built.
	40. Code covered.	Production code.	2. With TDD, it is impossible to write code without having that code covered by a test.
	41. All tests.	Test run all times.	3. Test can be run any time and you run test to see noting breaks.

Appendix G - Codes to Themes for Better Developer Productivity

Organising Theme - Developers Productivity			
Basic Themes	**Codes**	**Issues/Advantages discussed**	**Themes identified**
Less defect rate	1. Having unit test.	Extra time to code/push features.	1. Each time you run your unit tests, you save yourself the amount of time it would have taken to manually test your code.
	2. Reliable	Automated tests.	2. The automated test is more reliable than a person; it will run the test in exactly the same way, every time.
	3. One of the principles of Agile.	Maximising the code not written.	3. In TDD there is no need to get it right the first time that leads to overdesign, before sufficient information is available to make the design decision.
	4. Fixing complexity with refactoring.	Fix it now and don't leave it later	4. If you do not refactor messy code now and decide to leave it later because you fear breaking it then problems start to pile on to each other as time progresses creating a backlog.
	5. Changes verifiable.	Refactor advantage.	5. You can refactor the underlying implementation of a method and the unit tests will verify that the functionality of the method has not changed
Reduction of bugs in the coding stage	6. Sooner cheaper.	Bug finding.	1. Earlier finding and fixing the bug cost less and saves time.

	7. Development speed.	Strong test suite.	2. Creating a strong test suite with TDD during the development is always better off. The speed of development will be maintained even as the project complexity increases, compared to non-TDD team.
	8. Writing test.	Stops adding extraneous capabilities.	3. When a developer writes test then it really puts a damper on developer driven scope creep.
Less rework	9. Clear understanding.	Output and usage of the function.	1. When you write test you exactly know the behavior and interface of the functions
	10. Lifetime of your project.	Long lasting.	2. The more hours you plan to work on it, the better the case for tests. Not writing tests always comes back and bites you later during rework or maintenance phase.
	11. Code is More Understandable.	Handy set of API documentation.	3. Capture the knowledge you put into building the system into a very accessible form as tests capture assumptions and subtleties.
	12. Unused code	Out of the system.	4. Developers design interfaces and write methods based on what might happen in future. TDD helps keep this parasite code out of your system.
	13. No over engineering.	Code is simple understandable.	5. TDD prevents over engineer your code that leads to complex facets that serve little purpose other than elegant or esoteric code.
Less effort to deliver final product	14. Iterative development.	Flexible to changes & customer’s requirements.	1. Fast delivery of features and customers involvement in each phase.

	15. Final Design.	Fewer attempts.	2. TDD helps to get the best final design in fewer attempts, compared to traditional development.
	16. Complex problem.	More time saved.	3. Refactoring and testing as we go and getting everything right the first time is the key as we exercising the code as going along complex problem becomes simple.
	17. Team dependence.	Experience and Tools used.	4. It depends largely on the experience of your team with TDD and the test framework(s) you choose to determine the learning curve, and the size of your project.
	18. Continuous delivery.	Code into production.	5. Can put code into production much more often without disrupting customers.
	19. Debugger	Less time spend.	6. You spend less time in the debugger writing System.out or printing statements, to figure out the behavior of you own code.
	20. Cost, Quality, Time triangle.	Relation with TDD.	7. Software project management will tell you that you can achieve only 2 items in a triangle compromising 1, But with TDD all 3 are possible in mid and long terms.
	21. No unwanted extra code.	Precise code.	8. You only write code that moves you closer to your final goal quickly.
Working with TDD	22. TDD is fun.	Interesting and challenging.	1. Manual testing is boring. Learning how to build your product using a series of repeatable tests is a good challenge and has a good pay off

	23. Scientifically minded developers.	Logical and fun technique	2. TDD keeps the developers occupied making coding and designing enjoyable and fun.
	24. Better task focus.	One test case at a time.	3. When developing the functionality for a single test, the cognitive load of the programmer is lower.
	25. No way back.	Must never discontinue.	4. No developer can go back to traditional development away from TDD.
	26. TDD is muscle memory.	Must not think when doing TDD.	5. TDD must feel comfortable, doing that something in the real work, without needing to think about.
	27. TDD formula	Refactoring and Test first design.	6. TDD = Refactoring + Test First Design.
	28. Testability in mind.	Architecture that is Open, extensible and flexible.	7. Developing applications keeping testability in mind leads to robust, easy to change, and resistant to defects architecture.
	29. Without TDD	Zero confidence in design.	8. TDD gives me confidence that my design is right and also gives me confidence to refactor carefully or recklessly.
	30. Productivity	Matrix measurement.	9. Delivery or creativity of a quality software product in a limited period with limited resources.

Appendix H - Codes to Themes for Disadvantages of TDD

Organising Theme - Disadvantages of TDD			
Basic Themes	**Codes**	**Issues/Advantages discussed**	**Themes identified**
Steep learning curve	1. Learning is hard.	Not only TDD but programming too.	1. This is true. Programming well is hard to learn especially on your own. TDD makes you write code well and writing code well is hard.
	2. Intelligent refactoring.	Code isolation	2. Learning how to isolate code by intelligently refactoring your code to use the appropriate design patterns takes time.
	3. Tough sell to managers.	Concerned with timelines.	3. Software managers are generally only concerned with timelines. Asking for more time for TDD make lead to disagreement.
Dropped by teams when under pressure for delivery	4. More coding up front.	Can't skip tests.	1. Test-first means you can't skip tests, so you'll end up writing more code up front which takes a long time.
	5. Dropping TDD.	More Bugs, more after testing.	2. Comprehensive testing is often dropped by teams when they are under pressure resulting in more bugs at the customer and higher long term costs.
	6. Well-known issue.	Short term benefits.	3. Making a decision to trade of short term benefits for long term costs is a well-known issue in the industry and is known as building up technical debt.
	7. Immature teams.	Maintaining technical excellence.	4. This is an issue in our business. Maintaining technical excellence in the heat of a delivery cycle is difficult for immature teams.
	8. Development team.	Management discipline.	5. Management team that cannot install TDD discipline and attitude then any practices that you implement as part of any improvement program will fail.
Increased project complexity and cost	9. Cost	Not true. TDD reduces cost.	1. Initial few weeks its slower but by week 4-6 you are in the same speed with non-TDD team. Mid to longer term gains are 60% w.r.t to speed and maintenance cost.

	10. Complexity	Understand and trust TDD.	2. Once developer understand and trust TDD they begin to do less gold plating and crazy upfront design to ensure that the code is re-usable, flexible, and high.
Unable to test User Interface	11. ROI decision.	Simply not true.	1. The fact is that we often choose not to test the UI but this is a Return on investment decision not a feasibility decision.
	12. Technology used dependent.	UI is generally harder.	2. Limitation of the particular technology so developers by pass UI test. Easy to test with loosely coupled UI architectures such as MVVM (Presentation Model) and MVC.
Highly reliant on refactoring and programming skills	13. God coding skills.	Mastering TDD	1. If developers cannot write good code they will not be able to do TDD. If developers can't master TDD then they should not be able to write good code.
	14. Refactoring	Design skills	2. Yes developers need design knowledge to do good refactoring.
Database or external files are never tested directly	15. Layers in testing. Hard to test external dependency.	Integration/System tests.	1. Tests must test at least some code down to your DB Layer. If you are not doing automated system/acceptance tests then you should be doing manual QA. This will definitely be hitting the Database. Kent Beck’s 10 minute rule says that the tests must run under 10 minutes, so testing databases or web servers will delay running of tests.
	16. TDD unit test.	Test in isolation.	2. TDD tests a single class or method in isolation from its dependencies. Mocks and stubs help us isolate these codes under test and make our tests more focused and expressive.
	17. Test are order independent.	Fast, atomic, isolated, and conclusive.	3. TDD unit tests are order independent and they are executed automatically by the continuous integration server on each commit to the source control system.
	18. Use integration tests.	Validate code for external system.	4. Good integration tests validate the features of the external system that you use in your application.

End to End Testing	19. TDD tests.	Not End to End tests.	1. TDD test is not an end-to-end test. A TDD test does not interact with a live database or web server.
	20. Acceptance tests.	Customer’s requirements.	2. Acceptance test is for customer’s benefit and TDD tests are for developer’s benefit.