WHY DO SOFTWARE DEVELOPERS PRACTICE TEST-DRIVEN DEVELOPMENT?

PATRICK KAYONGO (KYNPAT001)
DEPARTMENT OF INFORMATION SYSTEMS
University of Cape Town

ABSTRACT

This dissertation describes research that has been undertaken to understand factors influencing software developers' intention to perform test driven development (TDD). Unit tests are a form of testing, where tests are written for small units of software being developed. TDD is a practice where these tests are written before the functionality is written, so as to guide the design of the code for the functionality, as well as to ensure test coverage for all functionality. There has been some research conducted to understand TDD by looking at its effects on both the outcomes and the practice of software development. It has been found to increase quality by decreasing defects, while also increasing the maintainability and the changeability of the code. On the other hand, some research has also found it to increase time spent on completing tasks. Despite this, to the best of the researcher's knowledge, there hasn't been research done to understand the behavioural components of TDD, and in particular, why developers choose to practice TDD.

A conceptual model based on the Theory of Planned Behaviour (TPB) is described and used as a lens to understand intention. TPB proposes that intention to perform a behaviour (TDD in this case) is influenced by three factors: attitude towards the behaviour, subjective norm, and perceived behavioural control. This dissertation seeks to build onto this model for the purpose of understanding TDD, and proposes the following determinants of the influences of intention: attitude is influenced by attitude towards time taken, differences in quality, maintainability and developer efficiency; subjective norm is influenced by the perceived perception of the environment regarding changes in quality, time taken, and maintainability of the code. Lastly, perceived behavioural control is posited to be made up of perceived difficulty of TDD, and how much experience a developer has.

This model is then tested based on data collected from an online survey distributed around the world. 779 responses were collected from developers in various countries around the world. The majority of the respondents to practice TDD, allowing us to gain greater insight into why those that practice TDD actually do so. Because the study is a psychographic study, perceptions were understood from the developers using an ordinal Likert scale. To analyse this data in order to prove the hypotheses, Chi-square tests with contingency tables, Kruskal-Wallis tests and ordinal logistic regression were used as statistical methods. It is found the data collected does not conform to the model, and recommendations are made for a future study to form a more comprehensive model.

PLAGIARISM DECLARATION

1. I know that plagiarism is wrong. Plagiarism is to use another's work and to pretend that it is one's own.
2. I have used the APA convention for citation and referencing. Each significant contribution to, and quotation in, this dissertation from the work, or works, of other people has been acknowledged through citation and reference
3. This dissertation is my own work. I have not used the material in this paper in any of my other essays/reports/projects.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.
5. I have included internet article references used for this dissertation.

Patrick Kayongo Signed by candidate

Signature Removed

17 May 2016

ACKNOWLEDGEMENTS

First of all, I would like to express my gratitude to my supervisor, Dr. Wallace Chigona, who patiently supported me through the uncertainty of topics and delays with completion. Without his continuous feedback and support, this would not be possible. I would also like to thank Robert C. Martin, with whose assistance, a diverse range of responses from around the world could be attained.

I would like to thank my wife, Sisanda Kayongo, who persistently encouraged me to complete this thesis, despite disillusionment and discouragement on my side. I would also like to thank my parents, Dr. Dan Mwesigwa-Kayongo and Mrs. Christine Kayongo, who created an environment where learning and the pursuit of knowledge was the norm, and consistently encouraged me towards the attainment of this Masters degree. Also, to friends of mine, specifically Sabatha Mhlanga, Mylo Mannya and Lubabalo Luayba whose encouragement spurred me on towards the completion of this.

Lastly and most importantly, I would like to thank YHWH, who is the source of all truth and rest, and by whose pleasurable grace knowledge can be sought. *Soli Deo Gloria*.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

CAS – Complex Adaptive Systems

RMSEA – Root mean square error of approximation

TDD – Test Driven Development

TPB – Theory of Planned Behaviour

TRA – Theory of Reasoned Action

XP – eXtreme Programming

1. INTRODUCTION

Software is becoming increasingly ubiquitous. From smart watches to smart homes, every part of the modern person's life is becoming more reliant on software and the devices that run this software. An example of this rise in the ubiquity of software is the growing trend of interconnected devices, or what is popularly known as the Internet of Things (IoT). Technology and research advisory company Gartner has predicted that by the year 2020, there will be 25 billion connected devices across consumer devices, business and automotive devices (Gartner, 2014).

With the growing prevalence of software, the way software packages change over time has also evolved. Traditionally, when updates needed to be made to software, the old software was stopped and removed from the device, after which the new software was added. In more recent times, software updates are done dynamically, sometimes while the software is still running (as is the case with many modern operating systems), giving the user new features or security and stability updates (Jhanwar & Yaryan, 2012).

With the rising use and dynamism of the software, it is becoming increasingly critical for software development projects to produce quality software, that meets the constantly changing requirements of users, while at the same time prevents defects from being introduced into the software.

A difficulty that many software development teams face is the increased volatility of the requirements from customers and organisations while a project is still in progress. These changes in requirements come about for reasons such as changing customer needs over the lifecycle of a project, and increased understanding of the domain by the development team (Nurmuliani, Zowghi, & Powell, 2004). It has been found that increased volatility of requirements lead to an increased amount of defects introduced into the software, especially when requirements are changed closer to the release date (Javed, Maqsood, & Durrani, 2004; Malaiya & Denton, 1999). The changes in requirements result in changes made to code for which its extended use is not fully understood. Because of the cohesive and coupled nature of code, changes in one part of the application can easily affect the functionality of another part of the application, introducing defects.

An additional challenge that is brought about by the cohesive and coupled nature of code, especially in large teams, is that defects that may be introduced by other team members to the parts of the application that they may not be working on. Studies have shown that, especially towards the end of a project, larger team sizes may lead to increased defects (Pendharkar & Rodger, 2009). This may be due to communication bottlenecks and a lack of understanding of the code that is being worked on. This risk is becoming greater with the rise of teams which are not co-located, and are distributed across geography and organizations (Jacobs et al., 2005).

With this increased dynamism of software development projects, volatility of requirements and changing nature of teams, it is becoming increasingly difficult for traditional software development practices to effectively deliver quality software that meets and continues to meet customer requirements. Traditional waterfall development practices had large upfront design, long periods of coding and testing, and finally a release to the customer of the software. Such practices did not cater well for the changes in customer requirements or changes in the code due to improved developer understanding of the domain.

1.1. AGILE SOFTWARE DEVELOPMENT

Agile software development methodologies have come about to address some of the adverse consequences of traditional waterfall software development practices, such as unfinished projects, long and extended timelines, analysis-paralysis without working software and dissatisfied and disengaged customers. The contexts in which software is developed for can be seen as complex adaptive systems, which are constantly changing due to changes in the environments. Agile methodologies try and foster software development practices which allow flexibility and adaptability to changing environments (Anderson, 2008; Dybå & Dingsøyr, 2008; Erickson, Lyytinen, & Siau, 2005; Highsmith, 2002).

The agile manifesto, which undergirds agile software development was written in 2001 by seventeen software developers from different areas of development. The manifesto states:

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more." (Beck et al., 2001)

Undergirded by these principles, many methods of agile software development have been created, such as eXtreme programming (XP), lean, scrum, Kanban and feature driven development. These methods differ in what they prescribe, and what they leave to the discretion of the software development team. For example, XP is prescriptive on actual software development practice and less on how the project is managed, whereas scrum has a focus on project management and introduces 'ceremonies' such as daily stand-ups, backlog grooming, etc., but doesn't have much to contribute regarding actual writing of code (Bowes, 2015).

As agile software development was a response to volatile requirements and the increasing dynamism of software projects, various practices were introduced to aid in this. Regarding project management, practices such as backlog grooming and the iterative practice of dividing timelines into 'sprints' were used to manage and prioritise changing requirements. Regarding development, distributed source control systems such as git were used to handle the distributed working of software teams. Continuous integration was a practice developed to merge changes from different developers, and tests created to ensure that the software integrates correctly, and code does not introduce defects into pre-existing code.

1.2. TEST DRIVEN DEVELOPMENT

A practice that was introduced to aid in the development of these tests, and is the focus of this dissertation, is test-driven development (TDD), which involves writing tests for functionality to be developed, followed by writing the functionality (Fowler, 2005). These tests allow the code to be more flexible or 'agile' as changes can be made where the risk of introducing regression defects in the existing functionality is mitigated.

Because tests are written first before the actual functionality is written, it ensures that all the functionality has tests written to test that it works, and the functionality continuously tested whenever the tests are run. This is a proactive approach to quality, compared to teams who intend to write tests after the functionality has been written, but may never do so due to time constraints, or functionality that has been written in a manner that makes it difficult to write tests for.

These tests are unit tests and form a part of a larger range of tests which are done on software during development. Unit tests are the most low-level of the tests, testing small pieces of units while ignoring external parts they integrate with, such as databases. A higher level of tests are integration tests, which test functionality with all parts integrated together such as the database, and the user interaction, with acceptance tests testing walkthroughs of the system as a user would (Mathur & Malik, 2010).

TDD first started becoming popular as part of eXtreme programming, one of the various agile methods. Several benefits have been found when using TDD, such as improved quality of the software due to reduced defects, improved maintainability of the codebase, and less regression defects introduced with the development of new software. There have also been a few challenges that have been discovered through different case studies, such as increased time taken to develop the software (Bhat & Nagappan, 2006; George & Williams, 2003; Williams, Maximilien, & Vouk, 2003).

Despite the increased popularity of TDD in some circles, there has also been criticisms of the practice from influential software development practitioners in the software development industry worldwide. One of the most notable criticisms of TDD came from David Heinemeier Hansson, founder of the popular Ruby on Rails web framework for the Ruby language, who wrote in a blog post:

"The current fanatical TDD experience leads to a primary focus on the unit tests, because those are the tests capable of driving the code design (the original justification for test-first).

I don't think that's healthy. Test-first units leads to an overly complex web of intermediary objects and indirection in order to avoid doing anything that's "slow". Like hitting the database. Or file IO. Or going through the browser to test the whole system. It's given birth to some truly horrendous monstrosities of architecture. A dense jungle of service objects, command patterns, and worse." (Hansson, 2014)

His argument was that in attempting to make the code more testable, a developer may increase the complexity of the software, making it more difficult to work with in future, and therefore actually decreasing the agility of the codebase. He posited to focus more on another form of testing called integration testing (see Section 2.2.2 Types of Software Testing) to ensure that applications functioned as intended.

Yet, his views were not without objection. Another influential figure in software development and an advocate of TDD, Robert C. Martin, responded in a follow up blog post, where he argued that Hansson's suggestions of how to ensure quality and trust in the code base through integration testing would not suffice in ensuring trust by the team that the software is working as it is supposed to. When additions and changes are made to the code base, one would not be sure that the existing functionality and the new functionality works as it is supposed to, as the suggested integration tests do not cover all scenarios (Martin, 2014). Her argues that:

"… integration tests have very little chance of meeting my two predicates.

First I doubt they can attain the necessary trustworthiness because they operate through the GUI; and you can't reach all the code from the GUI. There's lots of code in a normal system that deals with exceptions, errors, and odd corner cases that cannot be reached through the normal user interface.

Indeed, I reckon you can only cover a bit more than half the code that way. It seems unlikely to me that anyone would be willing to deploy a system based on tests that leave such a large fraction of the code uncovered." (Martin, 2014)

Many other practitioners have written articles and had talks about the benefits and the drawbacks of TDD. From the diversity of comment from practitioners, it is evident that it is a contentious issue within software development, with opinions ranging from complete dislike to evangelistic praise of the practice.

1.3. RESEARCH QUESTIONS

Despite the increased commentary on the topic by practitioners, to the best of the researcher's knowledge, there hasn't been much academic research done to understand the behavioural aspects of the practice of TDD. What has been done are case studies in companies that have adopted the practice for a project, to understand some of the benefits or the challenges of TDD. Also, controlled experiments with developers, especially students have been done in an attempt to empirically measure some of the effects of TDD. Yet, there haven't been any studies that have been done to produce a generalizable understanding of the phenomenon.

In addition, despite the little research done on the outcomes of practicing TDD, there hasn't been much research about the behavioural and psychographic aspects which result in individuals adopting the practice. From the diversity of practice and opinion about the topic from practitioners, one can deduce that there are a number of factors that lead an individual to performing TDD. This research was undertaken to understand why developers intend to practice TDD.

The intention to perform TDD can be understood through the Theory of Planned Behaviour (TPB) (Ajzen, 1991) which posits that intention to perform a behaviour is influenced by three factors, namely (i) attitude towards the behaviour, (ii) subjective norms, and (iii) perceived behavioural control. The attitude towards the behaviour is made up of the beliefs about the outcomes of performing the behaviour together with the strength of those beliefs. Subjective norm is made up of the perceived beliefs of those around the developer and the perceived strength of those beliefs. Lastly, perceived behavioural consists of a developer's belief in whether they can actually perform the behaviour (self-efficacy), as well perceived locus of control.

Using the TPB, coupled with the benefits and drawbacks of TDD that can be found in the literature, this dissertation seeks to answer the following questions about TDD among software developers around the world:

What are the motivational factors that affect a software developer's intention to perform TDD?

In answering this main question, the thesis seeks to answer the following sub-questions:

1. Does the software developer's attitude towards the outcomes of TDD affect their intention to perform TDD?
2. Do the beliefs held by others in the software developer's environment affect their intention to perform TDD?
3. Do experience and skill level affect a developer's intention to perform TDD?

Such research would be important to software development practitioners and teams. Though limited, the research that has been done has shown that there are benefits to TDD. It has shown to decrease the number of defects appearing, increase the maintainability of the codebase, and in some cases

increase developer efficiency, all which contribute to better software produced in this dynamic environment (Bhat & Nagappan, 2006; George & Williams, 2003; Williams et al., 2003). By understanding the psychographic motivations behind the practice of TDD, software team managers and other organizational leaders can motivate their teams to perform TDD as part of their daily patterns of software development. The better produced software would not only be beneficial for the end customer through less defects, but also to the organization that develops the software as they would have more confidence in their codebase, allowing them to handle continuously changing requirements, increasing their agility.

1.4. OUTLINE OF THE THESIS

The next chapter will review the existing literature surrounding the topic. As TDD forms part of the practices that emerged from agile methodologies, agile software development will be explored with the theoretical foundations of complex adaptive systems that help define 'agile' software development. Followed from this, the sub-field of software testing will be explored to understand its history and its different components, such as integration testing, acceptance testing and user testing. The specific testing practice of unit testing and TDD will then be discussed. Chapter 3 will analyse of the Theory of Planned Behaviour, the conceptual model that will be used to understand the influences that can determine a software developer's intention to perform TDD.

Chapter 4 will be an explanation of the actual research conducted. First, the philosophical presuppositions in the research will be outlined. A realist ontological perspective and positivist epistemological perspective has been taken, as this aligns with both the assumption of an objective. Second, the survey that was used as a research instrument will be discussed. Most of the developers who responded practice TDD, which may not be an accurate reflection of the world software development population. Despite this, the results still provide valuable insight into why those who do practice TDD actually do so. The data is described further in Chapter 5. In Chapter 6, the hypotheses that test the data against the conceptual model are then explored. Though there are limited statistical analysis techniques for ordinal data which was used throughout the research instrument, three different techniques were used, which confirmed the relationships described in the conceptual model for describing the attitude and subjective norm components. The perceived behavioural control components were shown not to support what was described in the model. Even with the data and the statistical results showing a conceptual model support for most of the components, there appeared to the potential for some statistical error in the techniques used, which keeps the paper from generalizing the results to the population. More of this will be discussed in the final chapter, as well as the limitations and future studies.

2. LITERATURE SURVEY

Test-driven development is a practice which finds itself within two sub-fields of research within computer science and information systems: software testing and software development practice. Software development has to do with the translation of requirements into a format that can be understood and processed by a computer. TDD is concerned with this as the practice affects how the software is developed. On the other hand, software testing has to do with the verification of the software that has been created by the programmer to ensure that it matches the specified requirements.

Unit testing has formed part of software testing for decades, but TDD gained popularity both in practice and literature with the rise of agile software development methodologies and various methods that arose from it, especially Extreme Programming (XP) (George & Williams, 2003; Madeyski, 2006; Mathur & Malik, 2010). To understand this background, first agile software development and software testing will be explored, and these two will be brought together by further understanding unit testing..

2.1. AGILE SOFTWARE DEVELOPMENT

Agile software development is an approach to software development projects that has aimed to overcome some of the challenges posed by traditional process-driven software development approaches such as the waterfall method of software development (Highsmith, 2002).

The traditional, process-driven approaches of software development assumed a level of consistency in the operating environment over time. This assumption resulted in long periods of planning large releases of software to be developed, followed by the analysis of user requirements and the implementation of these user requirements (Figure 1). Yet, if there were changes in the environment that the software was being developed in, or changes in requirements, such a fixed approach found difficulty adjusting to these changes, or being 'agile'. (Dybå & Dingsøyr, 2008)

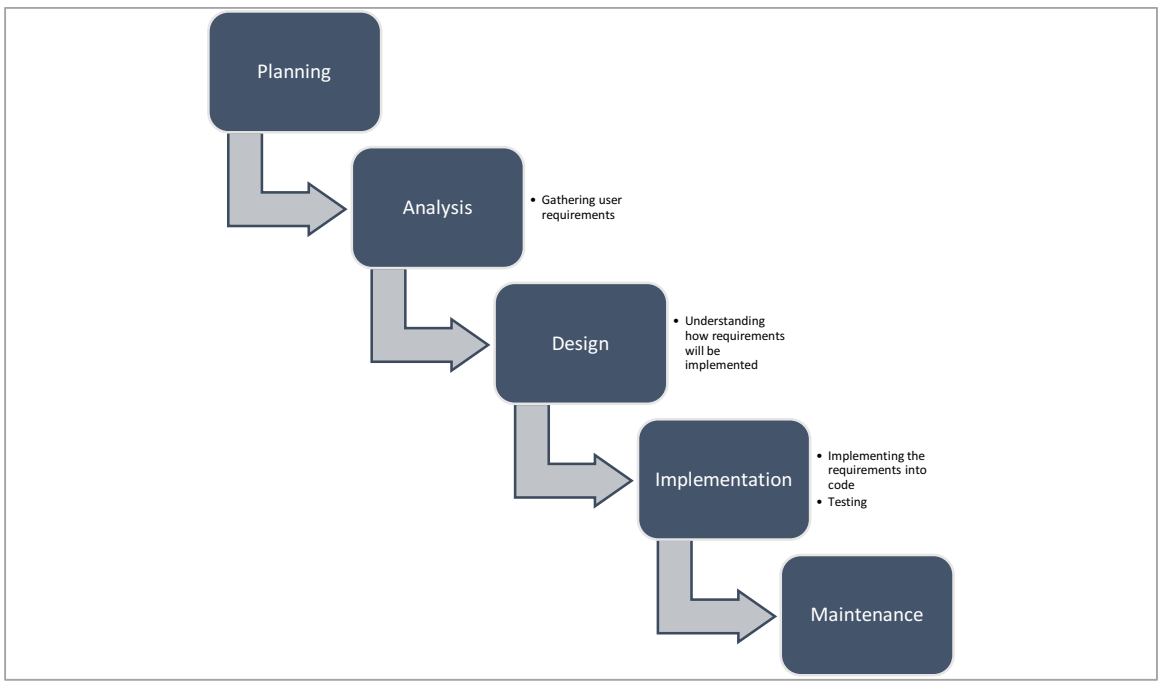


FIGURE 1 TRADITIONAL SOFTWARE DEVELOPMENT LIFECYCLE (SDLC) (BOEHM, 1988)

Software development firms sought to remove a lot of the scaffolding that came with the traditional approaches, to make it easier to quickly respond to changes in requirements within the environment they are developing for. (Erickson et al., 2005). A major shift in approach was incremental releases of smaller, working feature sets rather than large releases of complete applications with all the features.

Although not under the banner of 'Agile', there is a long history of iterative and incremental software development. The notion of iterative and incremental development (IID) emerged in the 1930s from a plan proposed by Walter Shewhart of Bell Labs, which consisted of "plan-do-study-act" (PDSA) cycles, or iterations which assisted in quality improvement. In 1969, Brian Randell and F.W. Zucher of the IMB T.J. Watson Research Centre sent an internal memo to management advising on the development approach which stated the following (Larman & Basili, 2003):

"The basic approach recognizes the futility of separating design, evaluation and documentation processes in software-system design. The design process is structured by an expanding model seeded by a formal definition of the system, which provides a first, executable functional model. It is tested and further expanded through a sequence of models, that develop an increasing amount of function and an increasing amount of detail as to how that function is to be executed. Ultimately, the model becomes the system" (Larman & Basili, 2003)

In Winston Royce's 1970s article, in which he describes what eventually became the waterfall approach to software development, he too advocated for shorter iterations with feedback to inform further development. The 1990s was when the ideas of agility started becoming more popularly with the development of concepts such as rapid application development (RAD) and XP. Finally, in February 2001, 17 individuals from separate strands which shared the same ideals met together and formed the Agile Alliance, producing the Agile Manifesto from it which described the values and principles of agile methodology (Larman & Basili, 2003).

2.1.1. THEORETICAL FOUNDATIONS OF AGILE SOFTWARE DEVELOPMENT

Agile methodologies were not necessarily developed through academic research, yet there are existing academic theories which can provide a foundation for the research of agile software methodologies.

COMPLEX ADAPTIVE SYSTEMS

Complex Adaptive Systems (CAS) draws from theories of both system theory and complexity theory (Stacey, 1995). Because different strands of research have brought about the understanding of complexity within systems, there is no universally accepted definition of CAS, yet there are common themes which run through the different research areas (Anderson, 2008).

The theories draw from system theory and agree that a system consists of agents, which interact with each other within the system. A system is also within an environment, and therefore there is interaction between the system and the environment. Traditional strategic approaches to this theory would assume that much of the interaction between the agents within the system, as well as the interaction between the system and the environment can be predicted. On the other hand, CAS theory understands that the result of many of these outcomes is unpredictable, and therefore cannot be planned over the long term (Stacey, 1995). There may be a common pattern of behaviour because the agents exist within an institution which can be seen as a system with its own rules and norms for interactions. But the heterogeneity of the agents means that they may act in ways which are different to the predictable norms and rules of the system, based on their individual rules, or schemata. Other agents who interact with the changing agent can respond either positively or negatively, and

depending on the responses and adaptations of other agents, there can be a change in the system as a whole (Anderson, 2008; Chiva-Gomez, 2004; Stacey, 1995).

In a similar way, proponents of agile methodologies acknowledge that the contexts in which they are creating software for are always changing. These can be changes from business requirements because of a changing business, to changes in priorities of the various requirements. They therefore seek to be flexible and 'agile' so as to adapt to this change and continue to effectively deliver the software that will provide the adapting business with value (Dybå & Dingsøyr, 2008).

2.1.2. CHARACTERISTICS OF AGILE METHODOLOGIES

The agile manifesto states four values for software development:

"Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan" (Beck et al., 2001)

These will be discussed in more detail below:

INDIVIDUALS AND INTERACTIONS OVER PROCESSES AND TOOLS

Traditional software development practices come from an engineering background in which processes are defined and optimized in an effort to achieve the best results. Agile approaches take a different approach. Because software is developed by people and not machines that can be optimized, to get the best software, competent people are required to understand the business domain and requirements (which form part of the client team) and to create the software (which form part of the development team). In addition to this, interaction is to be encouraged among individuals to allow for knowledge sharing and free information flow. (Cockburn & Highsmith, 2001)

WORKING SOFTWARE OVER COMPREHENSIVE DOCUMENTATION

Traditional software development processes value documentation to capture and analyse user requirements. The analysis phase to capture and document these requirements precedes the development of the user requirements into working functionality, and can often take much time, delaying the actual implementation of this software. Agile methods prioritise the actual development of the requirements into working software over the time taken to document these requirements within analysis documents. Knowledge remains tacit rather than explicit (Paulk, 2002).

Practitioners have discussed how unit tests can also act as documentation. Because traditional documentation produced in analysis phases of software development can become outdated when requirements change, a 'documented' way to find out how software actually works is by looking at the actual code. Because code may be difficult to understand, looking at tests which pass provide an easier way to decipher what the software is supposed to do (Farcic, 2014; Perpignand, 2010).

CUSTOMER COLLABORATION OVER CONTRACT NEGOTIATION

Many traditional software development cycles and methods advocate for written contracts indicating the exact work to be performed, and other details over that work such as the cost and the time to be taken. Agile methods prioritise a collaborative relationship with the customer in which requirements

are continually discussed and refined. This may cause problems if perceptions change over time and a disagreement on what was agreed upon, but the user stories advocated by many agile methods can be used as documentation and snapshots of thinking at certain points in time (Paulk, 2002).

More customer collaboration and less fixed contracts can result in more volatile agreements. Because this aspect of the agile manifesto embraces the fact that requirements do change, and seeks to create processes that allow for this change, the risks associated with this increased volatility have to be managed.

RESPONDING TO CHANGE OVER FOLLOWING A PLAN

This value underpins all the other values of the agile software development. As has been mentioned previously, agility can be seen as flexibility in responding and adapting to a changing environment. While traditional methods advocate for documentation of requirements, long term project plans that detail the implementation of those requirements, agile methods advocate shorter iterations with short term planning for an iteration. As has been mentioned earlier, it is difficult to ascertain the long term state of the context that the software is developed for, and therefore agile practitioners may view this long term planning as futile and not credible. Responding to this changing context therefore takes priority (Anderson, 2008; Dybå & Dingsøyr, 2008; Erickson et al., 2005).

As will be discussed later, TDD allows for responding to change by ensuring that defects aren't introduced into existing functionality, thus allowing for more confidence when releasing the new functionality.

2.2. SOFTWARE TESTING

Software testing involves ensuring that software which has been developed performs and functions as required and intended, as well as to discover potential defects in the developed software. It involves many different activities which occur at different stages from the analysis and implementation stages of development to the monitoring of production environments. (Bertolino, 2007; Gelperin & Hetzel, 1988).

2.2.1. HISTORY OF SOFTWARE TESTING

From the definition of software testing, it can be assumed that software testing originated in its most elementary form at the beginning of programming, but literature on software testing began around the 1970s, with its earliest conference on testing software applications occurring in 1972 (Bertolino, 2007; Hetzel, 1991).

Around the 1980s, testing and its practices started becoming a more formalized process and engineering discipline, and moved from a reactive activity in the software development process, to a more proactive activity. As the waterfall model for software development gained maturity, a V-model developed which linked different stages of the software development lifecycle to different testing activities, in terms of level of detail (Bertolino, 2007; Mathur & Malik, 2010). More information on the activities within this model can be seen in the next sub-section (Types of Software Testing).

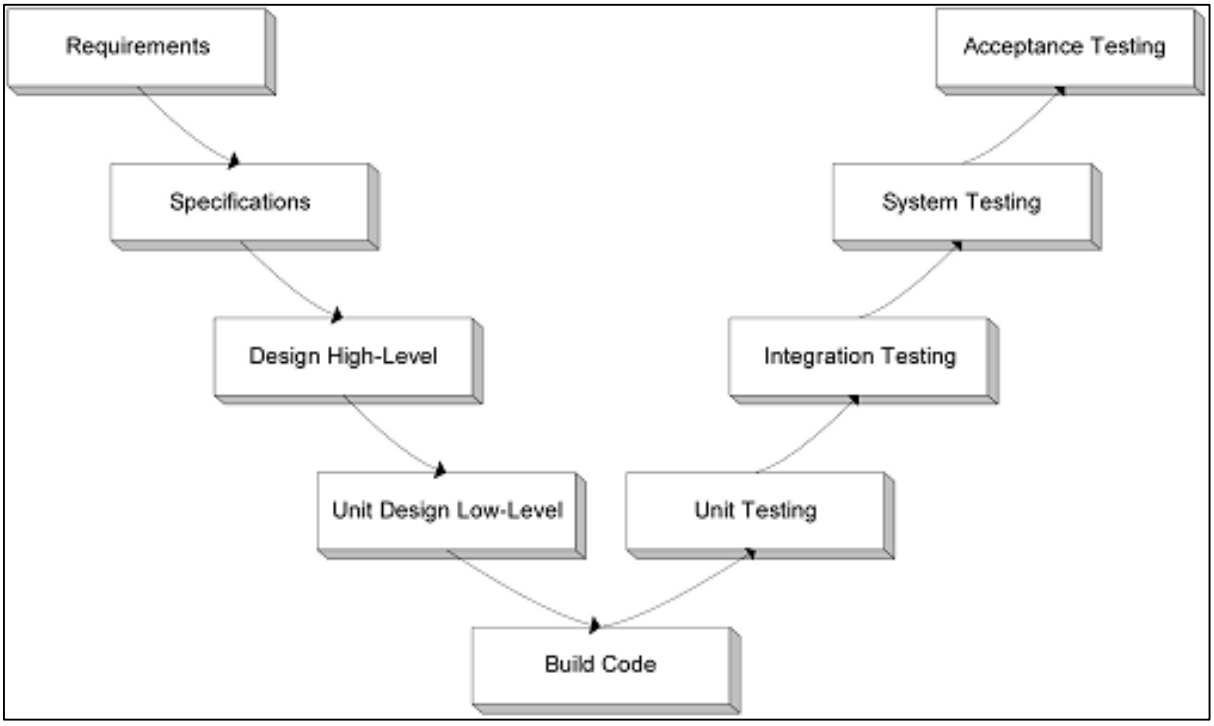


FIGURE 2 V MODEL (FEDERAL HIGHWAY ADMINISTRATION, 2004)

With the rise of object-oriented programming, it was assumed that many of the advantages that it brought (e.g. modularity) would lessen the need for software testing, but it was found that the very same principles that were applied beforehand were still required to ensure the quality of the developed software (Bertolino, 2007).

2.2.2. TYPES OF SOFTWARE TESTING

In practice, at various stages of the software development cycle, different kinds of testing activities are performed to ensure the quality of the software developed. Table 1 highlights four common activities in order of their granularity:

TABLE 1 SOFTWARE TESTING ACTIVITIES (MATHUR & MALIK, 2010)

Activity	Definition
Unit Testing	Involves testing of individual components and units of actual code to make sure these lower level items of implementation perform as intended. Within object-oriented programming, this can involve the testing of public methods of a particular class.
Integration Testing	The focus is integrating the different parts or 'units' of the system to ensure they work together as one whole. Black-box testing is done here, as the implementation details of the finer parts of the system aren't of concern, but the outcome of when they are performing together. This also involves regression testing to ensure pre-existing functionality still works as intended and defects have not been introduced.
System Testing	This involves testing of a product to ensure it performs as intended. It involves walkthroughs of different scenarios of using the software and ensuring that the functionality matches the requirements. It identifying what may have been missed during analysis, e.g. usability issues. It also involves security testing and load testing. Both manual and automated testing can form part of this phase.
Acceptance Testing	This is normally the final testing performed by the end-user or customer to ensure that the software matches customer requirements.

2.3. UNIT TESTING AND TEST DRIVEN DEVELOPMENT

Unit testing involves writing small tests to confirm that a piece of written programming language is working as intended. This involves the setting up of test data, the running of the piece of the software that manipulates the data, and the confirmation that the result of the manipulation is as intended (Wappler & Lammermann, 2005).

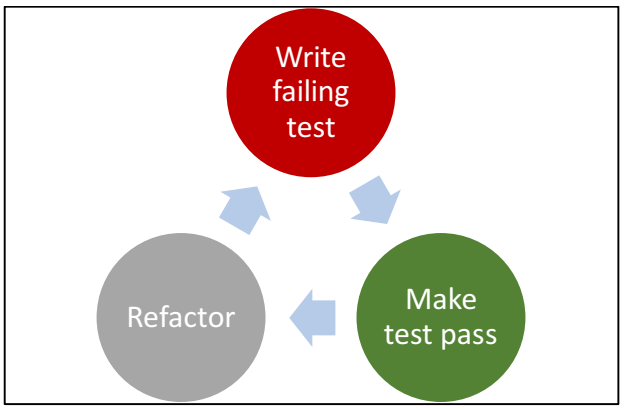


FIGURE 3 TDD CYCLE

TDD development is a practice where unit test cases are written first. Because the code implementing the functionality does not exist yet, the unit test will fail. The implementation code is then written by the developer, which causes the test to pass. After this, the developer can refactor the implemented code (i.e. make alterations to the structure of the code to increase efficiency or readability), and use the test to ensure that the production code functions as is intended. (Bhat & Nagappan, 2006; George & Williams, 2003; Williams et al., 2003).

2.3.1. BENEFITS OF TDD

There are not many generalizable studies have done to assess the benefits of TDD. Most of the studies which have been done are case studies within organizations, or experiments in controlled environments that measure the differences in various measures when TDD has been used to develop software compared to when it has not been used.

In various case studies, a common finding is that there is an increase in software quality shown by a reduction of the defects found. When compared to cases where TDD isn't used, it has been found that the number of defects found in the software was less than what was expected, or less than the defects found in a similar control project where other factors were kept similar (Bhat & Nagappan, 2006; George & Williams, 2003; Williams et al., 2003).

A second benefit that has been found in these case studies is the maintainability and changeability of the code. This is how easy it is to make changes to the code at a later stage, and the prevention of adding defects on existing functionality (regression defects) when new functionality is developed on an existing codebase. As has been mentioned above, test-driven development forms part of many methods found in agile software methodology, which seeks flexibility and the ability to change with a changing environment and changing requirements (as software is developed for organizations which may be seen as complex adaptive systems). Tests may also increase the ease of changing as existing functionality is tested, and therefore changes in the code can be made to adapt to the changing environment with less risk of introducing defects into existing functionality (Bhat & Nagappan, 2006; George & Williams, 2003).

At a study done at IBM, developers found it easier to make changes to code, and reduced the risk of regression defects when using TDD compared to when TDD was not used (Williams et al., 2003).

A last common benefit that has been found is an increase in developer efficiency with shorter pieces of focus, allowing the developer to focus on one piece of functionality at a time. When errors occur, because of the smaller scope of the work being done at that moment in time, it is easier to find and rectify the relevant fault. (George & Williams, 2003; Williams et al., 2003).

2.3.2. CHALLENGES OF TDD

Similar to the benefits of using TDD, there hasn't been much generalizable research done to understand the challenges of using TDD in practice. Again, most of the insights come from case studies and controlled experiments, but more needs to be done to understand the challenges.

The biggest challenge that is evident in the literature is the additional time taken to write the test cases. When compared to a case where no tests are written or only a few test cases were written, it has been found that there is a statistically significant increase in time writing these tests, as they are written for every piece of functionality, and for various scenarios of the feature (Bhat & Nagappan, 2006; George & Williams, 2003). Some studies have also found that writing unit tests first do not lead to an increase in the readability and maintainability of the code base (Madeyski, 2006). A cause of this could be that in an effort to write the code in such a way that tests can be written, a developer may over-complicate the code and introduce unnecessary complexity, as was mentioned in the introduction (Hansson, 2014).

2.4. GAPS IN CURRENT RESEARCH

As has been evident, there has been much research in agile software development and the various forms it has taken in different contexts. Much of the research has focused on case studies of agile methodologies as a whole (e.g. Extreme Programming (XP) and Agile Modelling), or on practices such as pair programming and its benefits (e.g. the relationship between pair programming and the number of defects found in the software) (Erickson et al., 2005).

Despite this, as has been mentioned above, there still hasn't been much research conducted providing empirical evidence of the benefits and factors affecting many of the practices of agile software development besides pair programming and some analysis methods. It has been suggested that other practices of XP besides pair programming could be a focus for studies moving forward as there isn't much research on these practices at the moment (Erickson et al., 2005).

With regard to unit testing and TDD, a common practice found in many agile software development methods, much of the research has been case studies used to discover the benefits developers found adopting the practices. There hasn't been much research conducted which have looked at the behavioural aspects that determine whether the practice is adopted by individual developers and development teams. This will be the focus of this research.

2.5. SUMMARY OF CHAPTER

This chapter reviewed some of the literature around TDD. Firstly, agile methodologies and the theoretical underpinning of complex adaptive systems were discussed. TDD became popular from agile practices, especially XP. Software testing was discussed, and TDD was placed in the context of other types of testing such as integration testing and acceptance testing. Lastly, the benefits and challenges of TDD from the literature were explored.

3. THEORY OF PLANNED BEHAVIOUR - THEORETICAL FRAMEWORK

In practice, there are many different opinions about the use of unit tests while developing software. As has been mentioned, there are benefits which can be attained from the practice of automated software testing, but there are also many criticisms of the practice, with many individuals opting not to perform the practice.

In attempting to understand behaviours of individuals in social science research, there have been those that lean towards structure as the cause of behaviour, and those leaning towards individual human agency. While each of those sides of the scale may help us understand aspects of social behaviour, by themselves it is difficult to understand social reality and the causes of human activity. The theory of structuration by sociologist Anthony Giddens has been useful in removing the duality and combining the poles of structure and agency (Jones & Karsten, 2008).

Giddens saw structure, or a social setting, as something dynamic and continuously produced by the actions of the agents within that structure. Yet the structure still has an influence the actions of the agents (Giddens & Pierson, 1998). Researchers who take a naturalistic or positivist point of view assume that there are rules and fixed norms in the universe that govern the behaviour of individuals. Therefore, to understand why individuals practice TDD, one would be required to understand the artefacts, structural properties and natural 'laws' that exist in their setting that drive individuals to behaviour. The trouble with this point of view, as Giddens pointed out, is that it assumes that human agents are inept at making choices and their wills are entirely subject to forces beyond their control (Jones & Karsten, 2008). An example of this would be that people practice TDD depending on the programming language used. The assumption in that position is that the programming language (assuming one that allows TDD to happen) is an unchanging force that has the same results for all agents who may or may not practice TDD. Yet, according to Giddens, these artefacts have no meaning apart from the meaning given to them by the agents who interact with them. When agents (members of software development teams in this case) interact with the artefacts, meaning is given to them and structure is then formed (Giddens & Pierson, 1998).

Therefore, according to the Giddens' structuration theory, social activity is based on interpretations and meanings given to artefacts and activities of others within the social system. Agency is influenced by these meanings given to artefacts and other agents, and the acting on these artefacts then reproduces structure at the moment of acting. A more specific theory that attempts to understand behaviour which takes into account the interpretive nature of how humans relate to artefacts and their environment is the Theory of Planned Behaviour. Within this dissertation, unit tests are assumed to be the artefact to which meaning is given.

The TPB is an enhanced version of the Theory of Reasoned Action (TRA) (Ajzen, 1991). This popular theory used in psychological research proposed that the intention to perform a behaviour was affected the beliefs about the outcomes of performing that behaviour. These beliefs are separated into two distinct kinds: behavioural, which form the basis of an individual's attitude towards the belief, and normative, which are the individual's beliefs about what is considered 'normal' in their environment, or their subjective norm (Madden, Ellen, & Ajzen, 1992).

In addition to the two factors proposed by the Theory of Reasoned Action, the Theory of Planned Behaviour proposes a third factor that affects an individual's intention to perform an action - perceived behavioural control. Whereas TRA assumed an agent to have 'pure volitional control' of the behaviour, TPB removes this assumption, and factors an agent's belief in their ability to perform the behaviour (Madden et al., 1992). Figure 4 below outlines the theory.

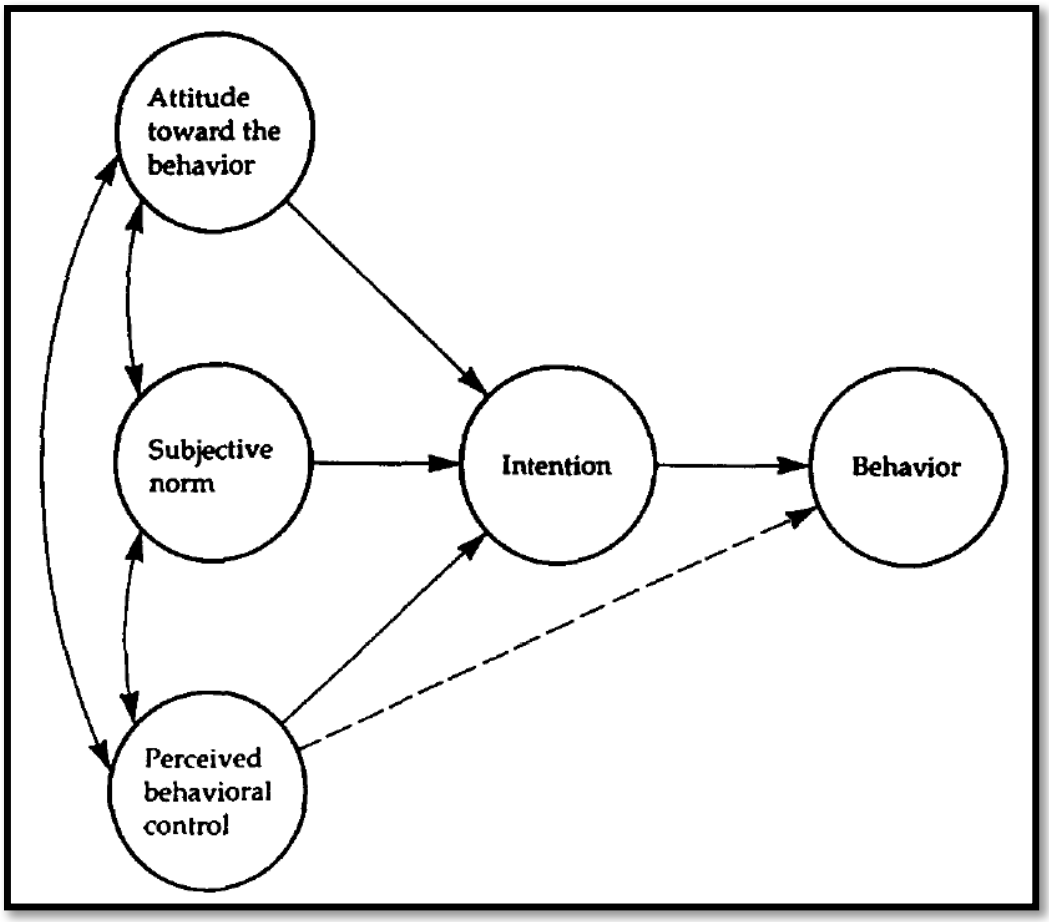


FIGURE 4: THEORY OF PLANNED BEHAVIOUR *(AJZEN, 1991)*

Behaviour relates to the actual performing of an activity. Intention on the other hand relates to the motivational factors behind performing the activity, they are "indications of how hard people are willing to try, of how much of an effort they are planning to exert, in order to perform the behaviour" (Ajzen, 1991). The greater the intention to perform the behaviour is, the more likely the agent (or software developer in this case) would perform the activity. If an agent performs the behaviour, then it can be assumed that there was intention to perform the behaviour. For this study, intention to perform the behaviour will be the main focus.

The subsequent sections will discuss each of these factors in detail.

3.1. ATTITUDE TOWARDS THE BEHAVIOUR

The attitude towards a particular behaviour affects whether an agent will intend to perform a behaviour or not. Relating to the structuration theory, this construct has to do with the meaning given to the artefact of unit tests, and the practice of TDD by the developer. According to the expectancy-value model of attitudes, these attitudes can be formed based on beliefs of the outcomes of the behaviour, as well as how much value the agent places on that particular outcome (Ajzen, 1991). A belief in a negative outcome of the action would result in a negative attitude towards the behaviour. Likewise a belief in a positive outcome would result in a positive attitude towards the behaviour. The strength of the attitude towards the behaviour is dependent on how much value the agent places on the specific outcome of the behaviour.

In the case of unit testing, an agent's attitude towards the behaviour of writing unit tests for the software they are developing is affected by the agent's belief in the outcome of that behaviour. For example, if the agent believes that writing unit tests will result in more unnecessary time spent (a negative outcome), they will have a negative attitude to the behaviour of writing unit tests. On the other hand, if the agent believes that writing unit tests improves the quality of the software they are developing (a positive outcome), it will result in a positive attitude towards the behaviour of writing unit tests. As has been mentioned, the strength of the positive or negative attitude is dependent on how much value the agent places on the positive or negative outcome of the behaviour.

Using the expectancy-value model as has been described above, the relationship between attitude, beliefs and values can be modelled with the following equation:

$$A \propto \sum_{i=1}^{n} b_i e_i$$

The equation shows that an agent's attitude (A) is directly proportional to the sum of the multiplied values of the strength of an agent's belief in certain outcome (b) and the agent's subjective evaluation of the attribute (e). Therefore, the more positive beliefs a developer has about the outcomes of the behaviour of unit testing, the greater the probability of a positive attitude towards the behaviour. Similarly, the more negative beliefs an agent has about the outcomes unit testing, the greater the probability of a negative attitude towards unit testing.

Many researchers have subdivided attitude into two components: affective and instrumental components. The affective component describes beliefs that relate to enjoyable or unenjoyable outcomes, whereas the instrumental component describes beliefs that relate to beneficial or harmful outcomes (Rhodes & Courneya, 2003).

A developer can have positive and negative attitudes to TDD based on the benefits and challenges of the practice mentioned earlier. For example, if a developer believes that TDD increases the quality of the code and reduces defects, this can result in a positive attitude towards TDD. Similarly, if a developer believes that TDD results in unnecessarily increasing the development time, this will have a negative impact on their attitude towards practicing TDD proportional to the strength of the belief.

This leads to the following hypotheses:

- H_1: There is a relationship between a software developer's attitude towards TDD and the individual's perception of the difference in the quality of the software developed.
- H_2: There is a relationship between a software developer's attitude towards TDD and the individual's perception of the difference in time taken to develop software.
- H_3: There is a relationship between a software developer's attitude towards TDD and the individual's perception of the difference in individual efficiency.
- H_4: There is a relationship between a software developer's attitude towards TDD and the individual's perception of the maintainability of the code after performing TDD.
- H_5: There is a relationship between a software developer's attitude and intention to perform TDD.

3.2. SUBJECTIVE NORMS

Subjective norms are the support or disapproval that influential individuals or groups have of the behaviour under question. The strength of the influence of subjective norms in affecting an agent's intention to perform a particular behaviour is dependent on the perceived strength of the belief held by that influential party, as well as the agent's motivation to comply with the approval or disapproval of the influential party. (Ajzen, 1991)

Subjective norm has been divided into 2 components by many researchers: an injunctive component and a descriptive component. The injunctive component is concerned with whether there is perceived pressure from an agent's social network to perform an activity or not, and the descriptive component is concerned with whether the activity is being done in the social network (Cialdini, Reno, & Kallgren, 1990; Rhodes & Courneya, 2003).

Studies and social experiments have shown that both these components affect behaviour. As an example, in a study testing the affect on norms on littering, it was found that individuals littered less in clean places, and more in dirty places (descriptive component). Similarly, when individuals saw a member of their social network littering, they littered to a greater extent (injunctive component) (Cialdini et al., 1990).

These two components can also affect the intention to perform TDD within software development teams and organizations. In the case of the injunctive component, if there is a positive attitude to unit testing within the development team that an agent is in, there will be a more positive influence on the intention to write unit tests for the software. Similarly, if there are negative beliefs held by influential members of the team, such as a senior developer, there will be a negative impact on the intention to write unit tests. In the case of the descriptive component, if the software development team or other influential parties are performing TDD, then there is a greater chance that TDD will be performed by a developer.

The following equation can be used to model the relationship between the influence of subjective norms on intention and the belief of influential parties:

$$SN \propto \sum_{i=1}^{k} n_1 m_1$$

The equation above shows that the strength of the influence of subjective norms (*SN*) on intentions is directly proportional to the sum of the products of the strength of beliefs held by influential parties (*n*) and the person's motivation to comply with the influential party (*m*).

The perceptions mentioned above, which may affect individual attitudes towards TDD, can also have a profile on influential parties within a team or organization that the agent operates in. For example, if it is believed that TDD will result in improved quality and improved developer efficiency, it will be encouraged in the context. Similarly, if it is believed that it results in more time spent, increased cost, and marginal or no increase in quality, it will not be encouraged in that context.

The influences of the subjective norm variable are determined by the kind of context the software developer operates in. For example, if a developer works within an organization or a software development team, the main subjective norms will be the team that the find themselves in. On the other hand, if a developer operates by themselves, the main subjective norms could be more external, such as blogs or books or other influential software developers.

This leads to the following hypotheses:

H_6: There is a relationship between a software developer's perception of the subjective norm of TDD and the individual's perception of the environment's attitude of the difference in time when performing TDD.

H_7: There is a relationship between a software developer's perception of the subjective norm of TDD and the individual's perception of the environment's attitude of the difference in quality when performing TDD.

H_8: There is a relationship between a software developer's perception of the subjective norm of TDD and the individual's perception of the environment's attitude of the difference in the maintainability when performing TDD.

H_9: There is a relationship between a software developer environment's subjective norm and intention to perform TDD.

3.3. PERCEIVED BEHAVIOURAL CONTROL

The perceived behavioural control refers to the agent's perceived ease of performing certain activity. If an agent believes it is relatively easy to perform a certain activity, and that they have the required resources to perform the behaviour, it will have a positive effect on the intention an individual has to perform the activity (Ajzen, 1991).

Some researchers have also divided PBC into two components, namely self-efficacy, and controllability, which has to do with how much control the agent believes he has over the activity (Rhodes & Courneya, 2003). Self-efficacy can be define as the "personal judgments of one's capabilities to organize and execute courses of action to attain designated goals", and has been shown to be a key influence in whether an individual performs an activity or not (Zimmerman, 2000).

What we now call self-efficacy may be thought to be similar to the attitude component of TPB, as they relate to beliefs about the outcome of performing a certain activity, but in a study, Albert Bandura showed them to be separate components worth noting (Bandura, 1986). While individuals may have beliefs about the positive outcomes of performing activities, they may not perform these activities because of lack of confidence, or perceived difficulty, aspects of self-efficacy (Zimmerman, 2000).

The second component of PBC, which has been described as Locus of Control in theory, has to do with the whether the benefits that can be accrued from performing an activity are in the control of the agent performing the activity or are externally controlled. It has been found that those who believe they are in control are more likely to perform behaviours than those who believes that they have less control (Spector, 1982).

In the case of unit testing, the perceived ease or difficulty, as well as the perceived availability of resources will have an impact on the developer's intention to perform unit testing. If it is seen as relatively simple endeavour (relating to self-efficacy) and all resources are seen to be available (relating to locus of control), it will have a positive impact on the intention to perform the behaviour. On the other hand, if the task of unit testing is seen as difficult, or the individual perceives they lack some resources to complete the activity, it will have a negative impact on the intention to perform the behaviour of writing unit tests.

Within a review of literature discussing the industrial adoption of TDD, it was found that many developers do not adopt the practice because of a lack of experience in the practice (Causevic,

Sundmark, & Punnekkat, 2011). From this, it can be deduced that the more experience an individual has with TDD, the more comfortable and confident they will have in the practice, and therefore perform the behaviour. Experience can therefore be seen as a factor affecting the perceived behavioural control, which in turn affects a user's intention to perform a behaviour.

This leads to the following hypotheses:

H_{10}: TDD experience and intention to perform TDD are not independent

H_{11}: TDD difficulty and intention to perform TDD are not independent

TABLE 2 PROPOSED INFLUENCES OF FACTORS AFFECTING INTENTION

Factor	Proposed Influences
Attitude	Quality Time Maintainability of code Developer efficiency
Subjective Norm	Quality Time Maintainability of Code
Perceived Behavioural Control	Experience Difficulty

3.4. SUMMARY OF HYPOTHESES

The hypotheses which have been stated are based on the TPB, and the benefits as well as the challenges of TDD discussed in Chapter 2.

Relating to the Attitude construct, the following hypotheses are tested:

H_1: There is a relationship between a software developer's attitude towards TDD and the individual's perception of the difference in the quality of the software developed.

H_2: There is a relationship between a software developer's attitude towards TDD and the individual's perception of the difference in time taken to develop software.

H_3: There is a relationship between a software developer's attitude towards TDD and the individual's perception of the difference in individual efficiency.

H_4: There is a relationship between a software developer's attitude towards TDD and the individual's perception of the maintainability of the code after performing TDD.

H_5: There is a relationship between a software developer's attitude and intention to perform TDD.

Relating to the Subjective Norm construct, the following hypotheses are tested:

H_6: There is a relationship between a software developer's perception of the subjective norm of TDD and the individual's perception of the environment's attitude of the difference in time when performing TDD.

H_7: There is a relationship between a software developer's perception of the subjective norm of TDD and the individual's perception of the environment's attitude of the difference in quality when performing TDD.

H_8: There is a relationship between a software developer's perception of the subjective norm of TDD and the individual's perception of the environment's attitude of the difference in the maintainability when performing TDD.

H_9: There is a relationship between a software developer environment's subjective norm and intention to perform TDD.

Lastly, relating to the Perceived Behavioural Control construct, the following hypotheses are tested:

H_{10}: TDD experience and intention to perform TDD are not independent

H_{11}: TDD difficulty and intention to perform TDD are not independent

3.5. SUMMARY OF CHAPTER

The theoretical framework of TPB has been discussed in this chapter. It posits that intention is influenced by attitude, subjective norm and perceived behavioural control. Attitude is made up of the beliefs about the outcome of performing an action, and the strength of those beliefs. Attitude consists of the attitude towards changes in quality, time, developer efficiency and maintainability of the code. Subjective norm is made up on the perception of what is 'normal' in the environment, as well as the inclination to comply with the environment. In this study, subjective norm is proposed to consist of perception of influential parties' belief in changes to time, quality and maintainability. Lastly, perceived behavioural control has to do with self-efficacy and perception of control. In this study, developer experience and perception of difficulty of TDD is assumed to cater for the perceived behavioural control construct.

4. METHODOLOGY

This chapter is set out to explain how the research was undertaken. First, the philosophical presuppositions behind the research are to be discussed in which the positivistic epistemological nature of understanding knowledge will be explored. In conducting social research, there are several philosophical presuppositions that have to be made about the nature of the world and the nature of reality, and how we come to know and understand this reality. Within this paper, truth is seen to be objective, and can be understood deductively. From here, the sample and the research survey that has been used will be discussed. Lastly, the statistical methods that were used in analysing this data and testing the hypotheses will be summarised.

4.1. RESEARCH PHILOSOPHICAL PRESUPPOSITIONS

The literature review chapter introduced Giddens' structuration theory. The structuration theory posits that structure is created by human interpretation of the environment they are in and the meaning given to the various artefacts within that environment. As has been discussed with the TPB and its application to the intention to perform TDD, it is posited by this paper that the main factors that influence a software developer's intention to perform TDD is their attitude towards the practice, the subjective norms in the environment they find themselves in, and the perceived behavioural control composing of self-efficacy and locus of control. If one were studying a phenomena outside the individual developer, or if these factors related to characteristics outside the individual, they would be subjective, based on perceptions of the developer in question. Because attitudes towards the various outcomes of TDD, subjective norms, self-efficacy and perceived locus of control are entirely subjective based on each individual's interpretation of their environment, one cannot assume that fixed universal laws or repetitive and consistent events lead to individual software developer beliefs.

At the same time, the TPB used as a conceptual model in this research assumes a causal nature between phenomena. It posits that attitudes, subjective norms and perceived behavioural control determine an agent's intention to perform a behaviour, which in turn influences whether or not the agent performs the behaviour. Therefore, an objective reality is assumed in which there is an objective causal nature between phenomena. This closely aligns with a positivist assumption where an objective reality exists independently from the humans in the context under study (Chen & Hirschheim, 2004), and this reality can be understood deductively. This causal relationship is assumed to be generalizable across the population, and intention to perform is assumed to be deductively understandable through the factors that influence it.

In this thesis, the 'truth' that is to be known are the factors within the TPB theoretical model as well as the intention to perform TDD. The understanding of these factors are a psychographic task, implying that it is the developer being studied who holds the knowledge about themselves. Because this knowledge is not about something outside of the developer, the knowledge can be seen to be objective, and not relative. Therefore, this research adopts a realist ontological stance and a positivist epistemological stance.

A realist ontological stance posits that an objective reality exists (Sayer, 2000). Based on the discussions above, this is aligned with the rest of the research. The positivistic epistemological stance assumes an objective reality, and posits that this objective truth can be understood deductively. The deductive discovery of the knowledge in this case is getting the perceptions of TDD, subjective norm and behavioural control from the developer themselves.

4.2. TYPE OF STUDY AND SAMPLE

Based on the goal of the research to test hypotheses and deductively explain a developer's intention to practice TDD, as well as the ontological and epistemological assumptions discussed, a quantitative methodological approach is to be used to conduct the research. These will provide standardised and generalizable responses that can be used to test the hypotheses and deduce conclusions.

This is a cross-sectional study. Therefore, the research is to analyse responses at a certain point in time, without the additional analysis in trends of unit testing and TDD behaviour over time. The population that the research is focusing in are software developers who regularly create software. The unit of analysis is an individual software developer.

4.3. RESEARCH INSTRUMENT

A survey was used as the research instrument for data collection. A survey is a common instrument for data collection where TPB is a conceptual model behind the questions, and has been used in various fields to understand psychological motivations to perform a behaviour (Francis et al., 2004).

As mentioned in a previous section, each question mapped to various measures.

Survey Question	Measure Name	Framework Construct	Data Type
GENERAL QUESTIONS			
Do you primarily work for an organization or yourself?	OrganizationVsSelf		Categorical
If you work for an organization, is test- driven development permitted in your organization?	TDDPermitted		Binary
If your work for an organization, are you an in-house software developer for a company, or do you work for a soft- ware development consultancy?	InHouseVsConsultancy		Categorical
Do you work on an individual product, or do you work on many projects for many clients?	IndividualVsManyProjects		Categorical
Is the work you do mainly on new projects creating new applications, or maintaining and making enhancements to existing applications?	CreatingVsMaintaining		Categorical
FRAMEWORK-BASED QUESTIONS			
INTENTION			
Do you practice test-driven development?	PracticeTDD		Binary
If not, do you want to practice test- driven development?	WantTDD		Binary

Survey Question	Measure Name	Framework Construct	Data Type
ATTITUDE			
Test-driven development has a positive effect on the quality of software being developed.	AttitudeQuality	Attitude towards quality	Ordinal
Test-driven development unnecessarily increases the time spent developing software.	AttitudeTime	Attitude towards time	Ordinal
Test-driven development increases the maintainability of the software being developed.	AttitudeMaintainability	Attitude towards maintainability	Ordinal
Test-driven development increases developer efficiency.	AttitudeEfficiency	Attitude towards developer efficiency	Ordinal
Test driven development makes it easier to make changes to the software.	AttitudeChangeability	Attitude towards changeability	Ordinal
I feel that test-driven development is seen as a positive thing in software development.	Attitude	Attitude	Ordinal
SUBJECTIVE NORM			
Test-driven development is encouraged in the organization I work for.	SubjectiveNorm1	Subjective norm	Ordinal
The software developer community I follow (e.g. blogs, talks, etc.) see test- driven development as a positive thing.	SubjectiveNorm2	Subjective norm	Ordinal
My organization views the practice of test-driven development as an unnecessary increase in development time.	SubjectiveNormTime	Subjective norm relating to time	Ordinal
My team / organization believes that test-driven development increases the maintainability and readability of the code.	SubjectiveNormMaintainability	Subjective norm relating to maintainability	Ordinal
My team / organization believes that test-driven development decreases the number of defects in the software created.	SubjectiveNormQuality	Subjective norm relating to quality	Ordinal
My team / organization believes that test-driven development makes it easier to make changes to existing software.	SubjectiveNormChangeability	Subjective norm relating to changeability	Ordinal

Survey Question	Measure Name	Framework Construct	Data Type
PERCEIVED BEHAVIOURAL CONTROL			
Writing unit tests before writing functionality is difficult.	TDDDifficulty	Difficulty	Ordinal
I have experience in test-driven development.	TDDExperience	Experience	Ordinal

All ordinal data refers to responses on the Likert scale. These responses were recorded on a scale from 1 to 7 where 1 represented 'Strongly Disagree' and 7 represented 'Strongly Agree'. Measures with negative responses as higher numbers (AttitudeTime, SubjectiveNormTime, UnitTestDifficulty and TDDDifficulty) were reversed to ensure that higher numbers reflect positive responses while lower numbers reflect negative responses.

Two questions were asked to determine if users had the intention to practice TDD: whether they are practicing TDD, and if not, whether they would like to practice TDD. If they answered positively to any of these questions, the respondent is assumed to have the intention to practice TDD.

4.4. DATA COLLECTION

Data collection was conducted through an online survey which was available at the URL http://research.patkayongo.co.za. The survey was distributed through the researcher's LinkedIn connections who are software developers, various LinkedIn groups and posted several times on the social media website Twitter. The survey was open from the 1st July 2015 to 10th September 2015.

After two months of collecting responses, 779 responses were collected from software developers around the world who accessed the survey through the channels above, and referrals from other software developers.

4.5. DATA ANALYSIS METHODS

As this was a quantitative study, most of the analysis performed on data was statistical in nature. For this, the R programming language and software environment was used. R is an open source language and environment, giving two advantages to this project. Firstly, because it is free, it is accessible to the researcher. Secondly, because it is open source, there are many libraries available which serve multiple purposes, which may be difficult to find in some proprietary statistical software.

Summary statistics was performed on the data to understand the general make up of the sample. This included means of the numeric data such as the years of experience, as well as modes for the ordinal data. More of this will be discussed in the next section.

For testing the hypotheses which looked at how well the model fit in with the theoretical framework, three different statistical techniques were used: logistic regression, chi-square goodness of fit tests with contingency tables, and Kruskal-Wallis tests to test the associations between variables. Because all the data for testing the hypotheses was ordinal in nature, this data which measured the factors of the theoretical model could not be assumed to be normally distributed. Therefore, non-parametric tests had to be used that do not hold to the assumption of normality.

To validate the model and ensure that the additions to the TPB actually mapped to separate factors (assumed to be attitude, subjective norm and perceived behavioural control), confirmatory factor analysis was done. Root mean square error of approximation was done to test for model fit.

The Kruskal-Wallis test is a statistical test used for determining the relationship between independent groups. The test allows for the testing of ordinal data, making it appropriate for these purposes (Chan, 1997). This test was used to test the relationship between the factors proposed in the theoretical model (e.g. between attitude towards quality, and attitude towards TDD). A second test that was used to test relationships between groups was the chi-square test with contingency tables. The 7 point Likert scale used on the ordinal data was condensed into three categories: Disagree, Neutral and Agree. The data was then summarised into contingency tables based on these condensed categories, and a chi-square test performed to test the relationships between these categories. The last statistical method that was used was ordinal logistic regression, which was used to test whether the variables that made up a factor (e.g. the factor attitude made up of attitude towards quality, time and maintainability) did in fact make up that variable, and the strength of a variable in changing a particular factor. The interpretation is similar to that of ordinary multiple regression analysis, but the logistic scale allowed for the testing of ordinal data.

Qualitative responses were also collected within the questionnaire from the last question asking for any comments. These responses were analysed by coding the data based on the constructs proposed in the theoretical model, and verifying whether they confirmed the model.

4.6. ETHICAL CONSIDERATIONS

Because individuals are being surveyed within this research, ethical considerations have to be made. Firstly, respondents were made aware of the nature and the purpose of the research before they proceeded in answering the survey questions. This was done through an opening page on the survey with the intention and the purpose of the survey (see Appendix).

Because respondents' perceptions were being recorded, these had to be kept confidential. The respondents were informed that their responses would be kept confidential, and had the option of remaining anonymous. Where an email address was provided by the respondent who wanted to know the results of the research, these were kept secure on the laptop of the researcher, and on the access-controlled site of the survey instrument provider, TypeForm. Also, in all additional comments given by the software developers, the researcher ensured that there was no way comments mentioned in the research could be linked back to an individual software developer.

The instrument was distributed online to willing parties, therefore respondents were not coerced or forced into completing the questionnaire. Because of how the questionnaire was distributed, it can be assumed that each respondent completed the survey out of their own volition.

Lastly, the design and execution was done in accordance to the University of Cape Town ethics requirements.

4.7. LIMITATIONS OF THE STUDY

While there was an intention to have a representative sample of software developers around the world, the nature of the media through which the survey was distributed and the random nature of the responses that were received make it difficult to assume that the sample represents the entire

intended population under study. Therefore, the results to be discussed in the following section cannot be generalized to the entire population.

4.8. SUMMARY OF CHAPTER

The methodology is discussed within this chapter. Firstly, a realist ontological stance and positivist epistemological stance is taken, which resonates with the researcher and the type of research to be conducted. It assumes an objective reality. This is a quantitative study, in which data was collected using online surveys. 779 responses were collected and data was analysed through statistical techniques suitable to ordinal data such as chi-square tests, Kruskal-Wallis tests and ordered logistic regression.

5. RESULTS & ANALYSIS

From the samples that were distributed through various social media channels, there were 779 responses from developers around the world. This chapter will summarise the data that was collected in preparation for the analysis of the data in Chapter 6.

5.1. SUMMARY STATISTICS

The main measure was whether individuals had the intention to practice TDD. Out of all respondents, 94.6% had the intention to perform TDD. While the number of those who have an intention to perform TDD is not representative of the entire population of software developers worldwide, the sample could give us a good understanding of the motivations and psychometric driving factors of those software developers who do practice TDD.

A new SubjectiveNorm measure was determined based on the two SubjectiveNorm measures described above. This is determined by looking at if a respondent works for an organization or works for themselves. If the respondent works for an organization, the organization subjective norm is used (SubjectiveNorm1). If the respondent works for themselves, the general software development community subjective norm is used (SubjectiveNorm2).

WORK ENVIRONMENT

The majority of the respondents (93%) work for an organization and aren't freelancing. The implication of this is they are (to varying degrees) bound to norms, rules, policies and structures of the organizations they are bound to. In addition to this 69% of respondents indicated that they are in-house software developers, implying that software isn't the primary function of the organization in which they operate in (e.g. in the case of a software development consultancy). There was an even split between those who work on multiple software development projects (51%) and those who work on an individual product (49%). Lastly, for the majority of respondents (89%), test-driven development is permitted in their organisation. For those of whom it is not permitted in their organisation, a number indicated that they still have the intention to perform TDD, and therefore these results were not omitted from the sample.

The mean years of experience for the respondents was 11.74 years. There was a standard deviation of 7.26, indicating a good spread of experience between the sample.

DETERMINANTS OF INTENTION

The various determinants of intention to perform TDD from the TPB, which will be tested in the next section, were measured from the questions in the survey. The table below highlights the central tendency of each of these determinants. Because the questions were based on ordinal data, the modes of each of the measures were used to determine the central tendency. Means were not used as a measure of central tendency to avoid assuming conformity between the distances of consecutive ordinal values on the ordinal scale.

TABLE 3 MODES OF ORDINAL MEASURES

Measure	Mode
Attitude	7
AttitudeQuality1	7
AttitudeTime	6
AttitudeMaintainability	7
AttitudeEfficiency	6
AttitudeChangeability	7
SubjectiveNorm	4
SubjectiveNormTime	4
SubjectiveNormMaintainability	5
SubjectiveNormQuality	5
SubjectiveNormChangeability	4
TeamTDD	5
UnitTestDifficulty	3
TDDDifficulty	3
TDDExperience	7

5.2. LOCATION OF RESPONDENTS

There were 779 developers who responded from countries around the world. Within the survey, there was no question of location, but Google Analytics was used to track metadata about the respondents, from which the location of the respondents could be gathered. The Google Analytics tool tracked the country from which each of the sessions that accessed the website arose. As can be seen from Figure 5, most of the respondents were from the United States of America, but a fair number were from many other countries around the world. It must be noted that a session on the site does not necessarily translate into a completed survey, but the locations of sessions can be indicative of the location of respondents.

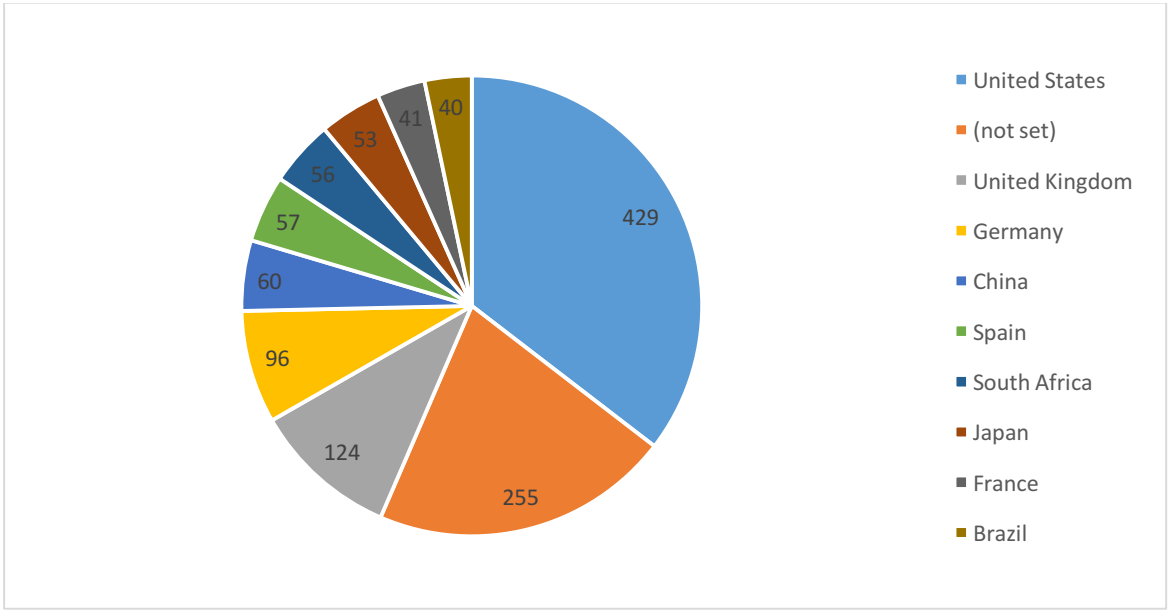


FIGURE 5 LOCATION OF RESPONDENTS

5.3. MODEL VALIDATION

To test whether the theoretical conceptual model that was constructed through the literature supported the data from the responses that were collected, structured equation modelling (SEM) was used. SEM has been growing in social science, as it provides a tool for theoretical model testing, validations and modifications using sample data.

Factor analysis forms part of the broader topic of SEM, which seeks to model constructs, and test whether constructs and the relationships between constructs support what has been proposed by a model. Within the model, there are two types of variables: observed variables (indicators) and latent variables (factors). These observed variables are from the data that has been collected, and the latent variables or factors are determined from the relationships between the indicators and the other factors (Savalei & Bentler, 2006).

Relationships between indicators and factors are determined using the covariance between the various indicators. In its simplistic form, if there is high covariance between indicators, it is then assumed that there is a relationship between these indicators, and they could represent a latent variable (or factor). The numerical value of these relationships are known as factor loadings, which can be seen as regression coefficients (Savalei & Bentler, 2006).

There are two types of factor analysis: confirmatory factor analysis and exploratory factor analysis. Confirmatory factor analysis seeks to test whether data supports a pre-specified theoretical model. This is done by comparing the theoretical covariance matrix (which represents the relationships between the constructs of the model), and the covariance matrix of the actual data received. On the other hand, exploratory factor analysis does not specify a model beforehand, but determines a model based on the data given, and the covariance between the various indicators of the model.

For the purposes of this study, confirmatory factor analysis was done. This was conducted using R for the statistical computation. Also, because the data was all based on ordinal data, ordinary factor analysis could not be done, as this assumes that the data under study is normally distributed and on a continuous scale. Performing normal factor analysis can result in biased results (Olsson, 1979; Savalei & Bentler, 2006). To mitigate this, when running the factor analysis, polychoric correlation was used.

Polychoric correlation can be used in cases where the data collected isn't on a continuous scale and is ordinal, such as the results from a Likert scale response. It is assumed that the underlying values that the responses represent are on a continuous scale, despite the responses not being on a continuous scale. The correlation between these determined by creating contingency tables of the responses, and using these to determine correlation (Olsson, 1979). Because this is all done by the software that is being used, these contingency tables didn't have to be created manually, and formed part of the automatic computation process of the factor analysis.

A confirmatory factor analysis was run on the new elements that were added to the TPB that was discussed in the literature review. When conducting the analysis in R, it was specified that there are a total of three factors, and the analysis would then determine the factor loadings. The function to conduct this analysis in R (the omegaSum function part of the psych package which uses the sem function which is part of the sem package) assumed a model where all the indicators related to all the

three factors. The resulting output from the software (Table 4) included the indicators, factors and factor loadings which are above 0.2.

TABLE 4 RESULTS OF FACTOR ANALYSIS FOR DETERMINANTS OF TPB

	g	F1*	F2*	F3*
AttitudeQuality	0.44		0.62	
AttitudeTime	0.22		0.48	
AttitudeMaintainability	0.43		0.56	
AttitudeEfficiency	0.38		0.61	
SubjectiveNormTime	0.25	0.59		
SubjectiveNormMaintainability	0.69	0.53		
SubjectiveNormQuality	0.69	0.48		
TeamTDD	0.39	0.64		
TDDDifficulty				0.62
TDDExperience	0.31			0.42

As can be seen in Table 4, the factor loadings for the three factors support the extended TPB model specified earlier. The attitude indicators all relate to one factor F2, which can be assumed as the Attitude factor. The various subjective norm indicators all represent factor F1, which can be assumed as the Subjective Norm factor. Lastly, the difficulty and experience factors all represent factor F3, which can be assumed as the Perceived Behavioural Control factor or latent variable. There are many proposed cut-off points for factor loadings using factor analysis from static values of 0.6 to values that vary based on sample size. For this paper, a cut-off value of 0.3 based on sample size will be used based on a recommendation by Hair et. al (Hair, Anderson, Tatham, & Black, 1998).

5.3.1. MODEL FIT

There are various different ways to test whether sample data fits the model that is provided. One of the most common goodness-of-fit conducted in research is the Chi-square test, which can be used to test how different the sample covariance matrix is from the theoretical covariance matrix. Despite its popularity, there are a number of limitations with its use, such that multivariate normality is assumed and its sensitivity to sample size (unnecessarily rejecting the model if there is a large sample size) (Hooper, Coughlan, & Mullen, 2008; Nevitt & Hancock, 2000). Because the current sample data doesn't have the property of multivariate normality, and the sample size is fairly large, the chi square goodness of fit test will not be used to test how well the sample data fits the conceptual model.

An alternative test for goodness of fit is the root mean square error of approximation (RMSEA). This test statistic is similar to the non-central chi-square distribution and allows some flexibility for "evaluating a model that is not exactly correct in the population" (Nevitt & Hancock, 2000).

There have been many recommendations for cut-off points for the test statistic to evaluate whether the model is a good fit or not. "Up until the early nineties, an RMSEA in the range of 0.05 to 0.10 was considered an indication of fair fit and values above 0.10 indicated poor fit. It was then thought that an RMSEA of between 0.08 to 0.10 provides a mediocre fit and below 0.08 shows a good fit".

The RMSEA index from the sample data was 0.089. According to the data above, this would indicate that this is a mediocre fit. Therefore, it cannot be concluded that the model is a good fit, and more testing of the model is required with regression testing of the hypotheses.

5.4. HYPOTHESES TESTING

In Section 5.3, it was seen that there there was a moderate fit between the data that was collected and the conceptual model that was built up through the literature. To further validate whether the model is correctly specified, the hypotheses that were stated earlier which seek to test the relationships described in the theoretical conceptual model.

Because the data that was collected to test the model is either ordinal or binary, parametric statistical methods that assume normality of the data cannot be used. To statistically test whether there is a relationship between variables, three non-parametric techniques were used: chi-squared tests with contingency tables, Kruskal-Wallis tests and ordered logistic regression was used.

The Kruskal-Wallis test is a non-parametric test that is used to test whether groups under analysis are the same or are statistically different. It is useful in this case as it allows for the testing of groups with ordinal data (Chan, 1997). The chi-squared tests with contingency tables use contingency tables were each ordinal value has been summarised into three categories: Disagree, Neutral and Agree. The Kruskal-Wallis test also includes a chi-square test, but the responses have not been condensed into three categories. Lastly, ordinal logistic regression is a regression technique that is used to predict an ordinal value based on independent values.

5.5. ATTITUDE

The conceptual model proposed included four different influences of a software developer's attitude towards practicing test-driven development: the developer's attitude towards the differences in quality of software developed, time taken, efficiency and maintainability of the underlying code.

When attempting to conduct an ordered logistic regression on the Attitude variable and all its determinants, the procedure could not be conducted in R because of the data provided, which indicates a wrong specification of the model. We can conclude that the relationships between attitude and its determinants is therefore wrongfully specified, but other tests were conducted to ensure that this conclusion is correct.

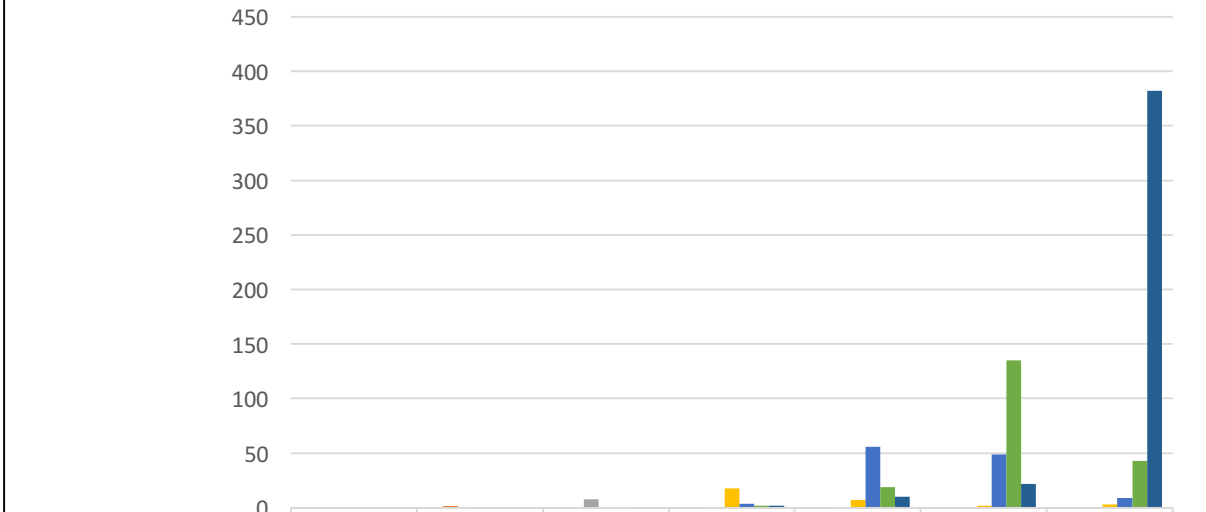
5.5.1. ATTITUDE AND PERCEPTION OF QUALITY

H_0: There is no relationship between a software developer's attitude towards TDD and the individual's perception of the difference in the quality of the software developed.

H_1: There is a relationship between a software developer's attitude towards TDD and the individual's perception of the difference in the quality of the software developed.

The relationship between attitude and perceived difference in quality can be seen in Figure 6. The rows in the table represent the responses to Attitude, and the columns in the table represent the responses towards the individual's perceived changes in quality.

As can be seen by both the values in the table and on the graph, there seems to be a positive relationship between an individual developer's attitude towards the perceived difference in quality, and a developer's attitude towards TDD.



	Highly Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Highly Agree
Highly Disagree	1	0	0	0	0	0	0
Disagree	0	2	0	0	0	0	0
Somewhat Disagree	0	1	8	1	1	0	0
Neutral	0	0	1	18	7	2	3
Somewhat Agree	0	0	1	4	56	49	9
Agree	0	0	0	2	19	135	43
Highly Agree	0	0	0	2	10	22	382

FIGURE 6 ATTITUDE TOWARDS DIFFERENCE IN QUALITY

To statistically test the relationship between the two, first a Kruskal-Wallis test was conducted. The Kruskal-Wallis test resulted in a p-value of $2.2 * 10^{-16}$ which would lead one to reject the null hypothesis of no relationship between attitude towards TDD and perceive difference in quality of software developed. The only concern is that the Kruskal-Wallis chi-squared result came to 485.29, which is fairly high, and may indicate a model which hasn't been specified correctly.

The chi-squared contingency table test from the condensed contingency table resulted in the same p-value of $2.2 * 10^{-16}$ but an even higher chi-squared value of 862.2. So again, while this leads us to

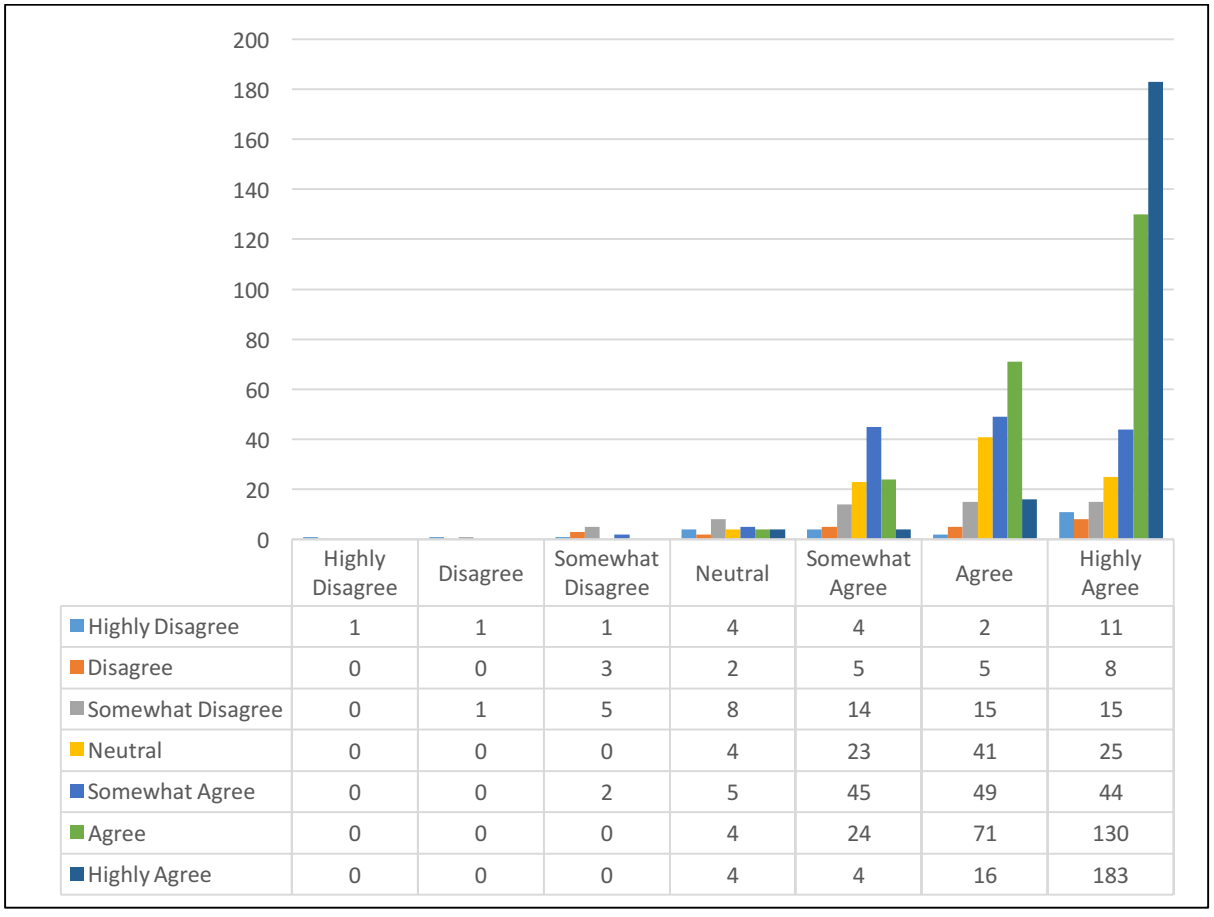
conclude that the null hypothesis should be rejected, it still represents a misspecification of the conceptual model.

5.5.2. ATTITUDE AND PERCEPTION OF DIFFERENCE IN TIME

H_0: There is no relationship between a software developer's attitude towards TDD and the individual's perception of the difference in time taken to develop software.

H_1: There is a relationship between a software developer's attitude towards TDD and the individual's perception of the difference in time taken to develop software.

The relationship between attitude and perceived difference in time taken to develop can be seen in Figure 7. The rows in the table represent the responses to Attitude, and the columns in the table represent the responses towards the individual's perceived changes in time taken to develop software.



	Highly Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Highly Agree
Highly Disagree	1	1	1	4	4	2	11
Disagree	0	0	3	2	5	5	8
Somewhat Disagree	0	1	5	8	14	15	15
Neutral	0	0	0	4	23	41	25
Somewhat Agree	0	0	2	5	45	49	44
Agree	0	0	0	4	24	71	130
Highly Agree	0	0	0	4	4	16	183

FIGURE 7 ATTITUDE TOWARDS DIFFERENCE IN TIME

In statistically testing the relationship between the two measures, the Kruskal-Wallis test resulted in a p-value of $2.2 * 10^{-16}$ leading us to reject the null hypothesis, and conclude there is a significant relationship between the two variables. Again, there was a high Kruskal-Wallis chi-squared value of 186.9 making implying a model that hasn't been specified correctly.

The chi-squared contingency table test of the condensed contingency table resulted in a p-value of $2.2 * 10^{-16}$ and a chi-squared value of 94.901. Again, this leads to the rejection of the null hypothesis,

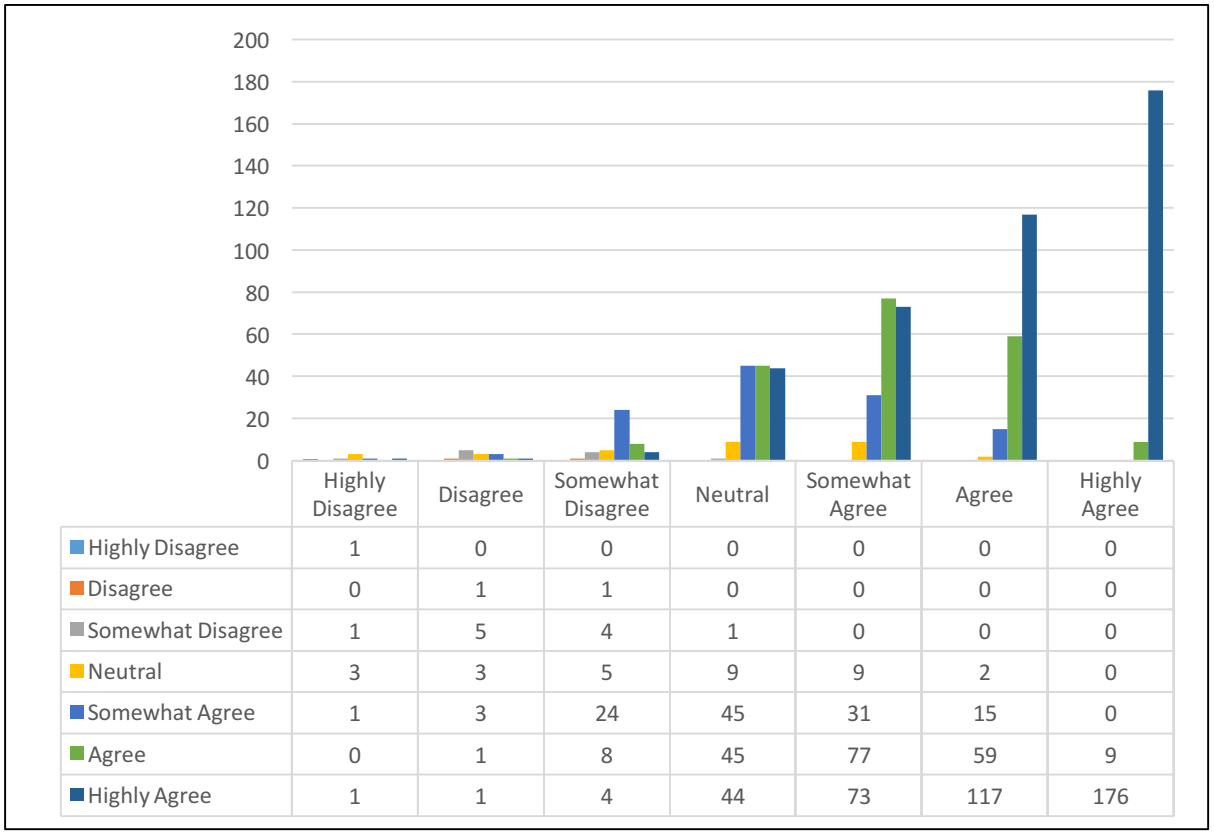
but the high chi-squared value also indicates that there may be a incorrect specification of the relationship in the conceptual model.

5.5.3. ATTITUDE AND PERCEPTION OF EFFICIENCY

H_0: There is no relationship between a software developer's attitude towards TDD and the individual's perception of the difference in individual efficiency.

H_1: There is a relationship between a software developer's attitude towards TDD and the individual's perception of the difference in individual efficiency.

The relationship between attitude and perceived difference in individual efficiency when developing can be seen in Figure 8. The rows in the table represent the responses to Attitude, and the columns in the table represent the responses towards the individual's efficiency.



	Highly Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Highly Agree
Highly Disagree	1	0	0	0	0	0	0
Disagree	0	1	1	0	0	0	0
Somewhat Disagree	1	5	4	1	0	0	0
Neutral	3	3	5	9	9	2	0
Somewhat Agree	1	3	24	45	31	15	0
Agree	0	1	8	45	77	59	9
Highly Agree	1	1	4	44	73	117	176

FIGURE 8 ATTITUDE TOWARDS EFFICIENCY

The Kruskal-Wallis test conducted when statistically testing the relationship resulted in a p-value of $2.2 * 10^{-16}$ and a Kruskal-Wallis chi-squared value of 277.88. Again, the p-value leads us to reject the null hypothesis and conclude that there is a statistically significant relationship between the variables. On the other hand, the high chi-squared value shows that the relationship may not be specified correctly within the conceptual model.

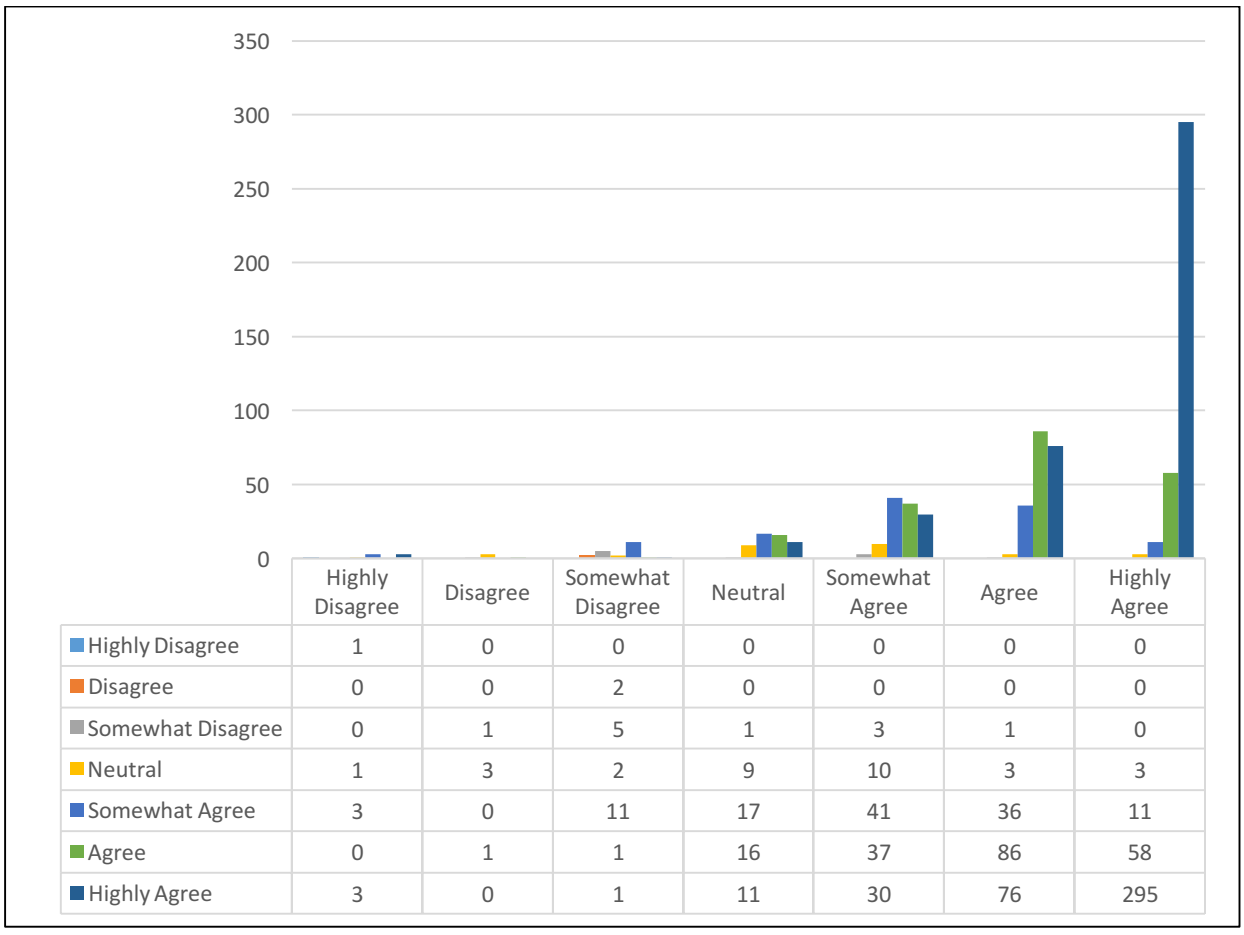
The chi-squared contingency table tests resulted in a p-value of $2.2 * 10^{-16}$ and a chi-squared value of 167.94. Again, this would lead us to reject the null hypothesis, and conclude that there is a statistically significant relationship between the variables, but the high chi-squared value may indicate something wrong with the model.

5.5.4. ATTITUDE AND PERCEPTION OF MAINTAINABILITY OF CODE

H_0: There is no relationship between a software developer's attitude towards TDD and the individual's perception of the maintainability of the code after performing TDD.

H_1: There is a relationship between a software developer's attitude towards TDD and the individual's perception of the maintainability of the code after performing TDD.

The relationship between attitude and perceived difference in maintainability of the code after performing TDD can be seen in Figure 9. The rows in the table represent the responses to Attitude, and the columns in the table represent the responses towards the individual's perception of maintainability.



	Highly Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Highly Agree
Highly Disagree	1	0	0	0	0	0	0
Disagree	0	0	2	0	0	0	0
Somewhat Disagree	0	1	5	1	3	1	0
Neutral	1	3	2	9	10	3	3
Somewhat Agree	3	0	11	17	41	36	11
Agree	0	1	1	16	37	86	58
Highly Agree	3	0	1	11	30	76	295

FIGURE 9 ATTITUDE TOWARDS MAINTAINABILITY

The Kruskal-Wallis test conducted to statistically test the relationship between the two variables had a p-value of $2.2 * 10^{-16}$ and a Kruskal-Wallis chi-squared value of 264.94. The p-value leads us to reject the null hypothesis and conclude that there is a statistically significant relationship between the variables. On the other hand, the high chi-squared value shows that the relationship may not be specified correctly within the conceptual model.

The chi-squared contingency table tests resulted in a p-value of $2.2 * 10^{-16}$ and a chi-squared value of 165.6. Again, this would lead us to reject the null hypothesis, and conclude that there is a statistically significant relationship between the variables, but the high chi-squared value indicates a wrongly specified model.

5.6. SUBJECTIVE NORM

The conceptual model describes four determinants of the subjective norm variable (the developer's perception of the beliefs of those in their environment): the perception on other's beliefs on the difference in time taken, quality of the software developed, and the maintainability of the software that has been developed while practicing TDD.

The results of the ordinal logistic regression that was run can be seen in Table 5 below. The interpretation of these is that "for a one unit increase in the predictor, the response variable level is expected to change by its respective regression coefficient in the ordered log-odds scale while the other variables in the model are held constant" (University of St Andrews, n.d.). For our data, this means that if a software developer perceives that those in his environment strongly agree that TDD has a positive effect on time (SubjectiveNorm variable with value of 7), then keeping all other things constant, the SubjectiveNorm variable will increase by 3.55 in the ordered log-odds scale.

TABLE 5 ORDINAL LOGISTIC REGRESSION RESULTS FOR SUBJECTIVE NORM

	Value	Std. Error	t value	p value
SubjectiveNormTime2	0.2953421	0.4770811	0.6190606	0.5358764334
SubjectiveNormTime3	0.8436500	0.4431042	1.9039541	0.0569161653
SubjectiveNormTime4	1.3025255	0.4230791	3.0786810	0.0020791921
SubjectiveNormTime5	1.8906020	0.4420510	4.2768871	0.0000189525
SubjectiveNormTime6	2.5549694	0.4400926	5.8055264	0.0000000064
SubjectiveNormTime7	3.5491602	0.4507501	7.8738970	0.0000000000
SubjectiveNormQuality2	1.4298126	0.9555240	1.4963649	0.1345585796
SubjectiveNormQuality3	1.6348293	0.8934731	1.8297467	0.0672878297
SubjectiveNormQuality4	1.9121901	0.8623708	2.2173642	0.0265982134
SubjectiveNormQuality5	1.9644126	0.8621364	2.2785404	0.0226944008
SubjectiveNormQuality6	2.4659754	0.8674879	2.8426627	0.0044738394
SubjectiveNormQuality7	2.7461819	0.8891056	3.0887016	0.0020103328
SubjectiveNormMaintainability2	0.2911612	0.8136267	0.3578560	0.7204510969
SubjectiveNormMaintainability3	0.6634766	0.7528159	0.8813265	0.3781411319
SubjectiveNormMaintainability4	0.9040641	0.7228585	1.2506792	0.2110515319
SubjectiveNormMaintainability5	1.4955236	0.7282232	2.0536609	0.0400085216
SubjectiveNormMaintainability6	1.8091692	0.7391089	2.4477709	0.0143743040
SubjectiveNormMaintainability7	2.9365316	0.7694910	3.8162002	0.0001355225

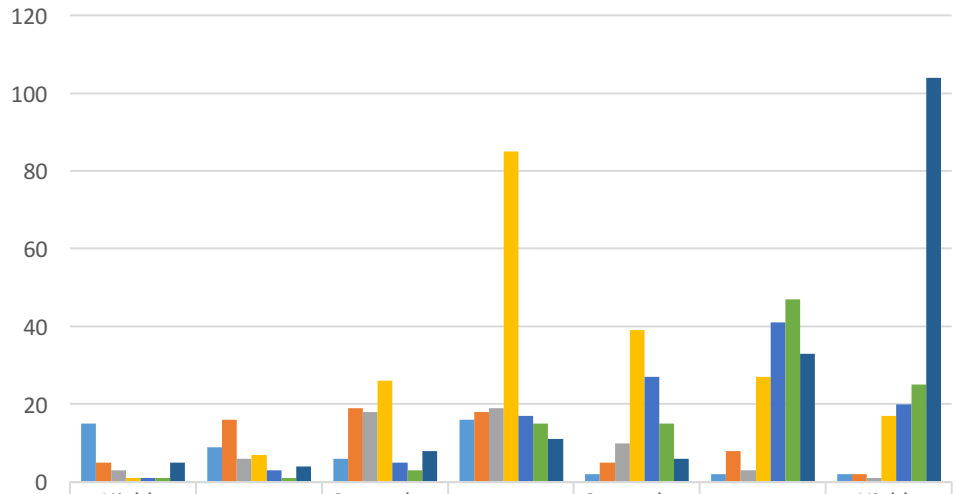
When looking at the p-values, from the baseline of Strongly Disagree, the p-values decrease when moving up the ordinal scale. Higher values have lower p-values, under the threshold of 0.05, therefore we can reject the null hypothesis (at a 95% confidence interval) that that there is no association between subjective norm, and the developer's perceived perception of the environments perception of the time difference. The same applies for SubjectiveNormQuality, and to a lesser extent SubjectiveNormMaintainability. To further test these hypotheses, Kruskal-Wallis tests and chi-squared contingency table tests will be run on the data.

5.6.1. SUBJECTIVE NORM AND TIME TAKEN

H_0: There is no relationship between a software developer's perception of the subjective norm of TDD and the individual's perception of the environment's attitude of the difference in time when performing TDD.

H_1: There is a relationship between a software developer's perception of the subjective norm of TDD and the individual's perception of the environment's attitude of the difference in time when performing TDD.

The relationship between subjective norm and the developer's perception of the environment's attitude towards the change in time when performing TDD can be seen in Figure 10. The rows in the table represent the responses to SubjectiveNorm, and the columns in the table represent the responses towards the environment's attitude to time (SubjectiveNormTime).



	Highly Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Highly Agree
Highly Disagree	15	9	6	16	2	2	2
Disagree	5	16	19	18	5	8	2
Somewhat Disagree	3	6	18	19	10	3	1
Neutral	1	7	26	85	39	27	17
Somewhat Agree	1	3	5	17	27	41	20
Agree	1	1	3	15	15	47	25
Highly Agree	5	4	8	11	6	33	104

FIGURE 10 SUBJECTIVE NORM AND TIME

When running the Kruskal-Wallis test to test whether there is a statistically significant relationship resulted in a p-value of $2.2 * 10^{-16}$ and a Kruskal-Wallis chi squared value of 284.34. The p-value leads us to reject the null hypothesis and conclude that there is a statistically significant relationship between the two variables. The high chi-squared value may indicate that the relationship may not be specified correctly in the conceptual model.

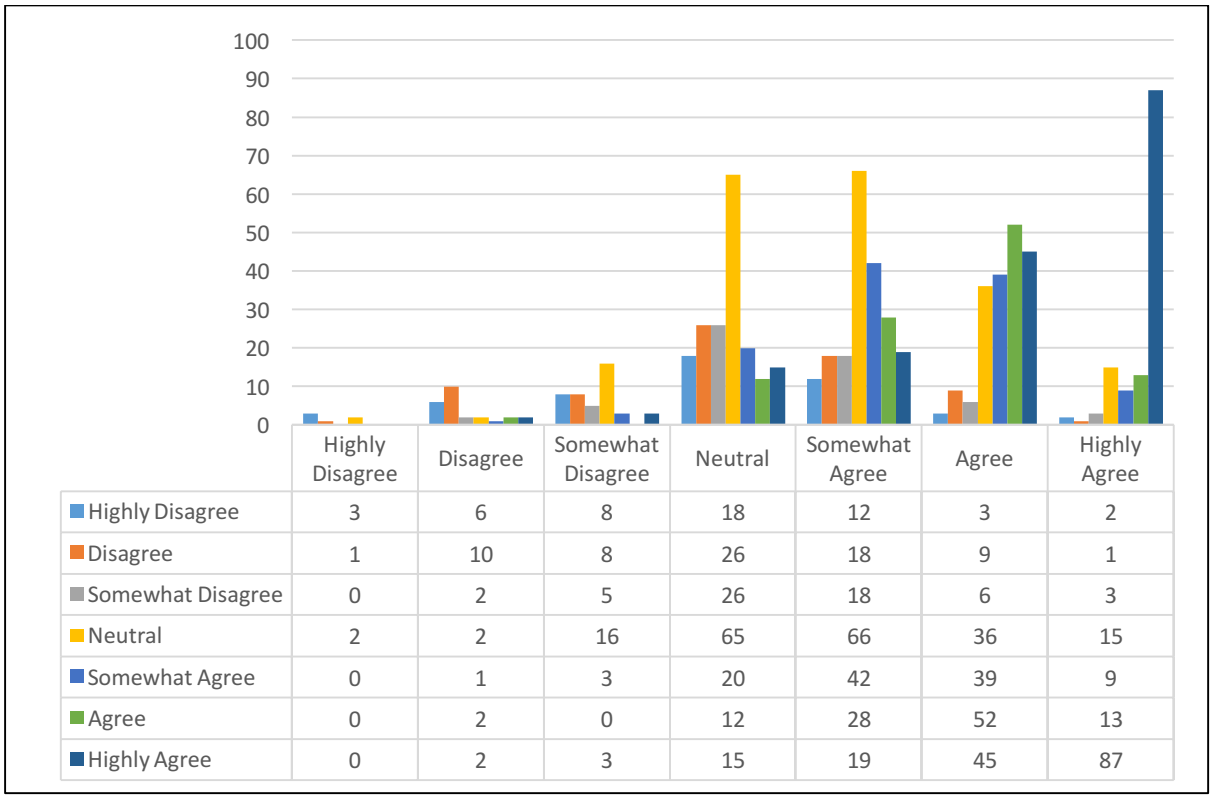
When running the chi-squared tests on the contingency tables, it results in a p-value of $2.2 * 10^{-16}$ and a chi-squared value of 278.14. Similar to the Kruskal-Wallis test, this leads us to reject the null hypothesis and conclude that there is a significant relationship between the two variables. Again, the high chi-squared value is a cause for concern.

5.6.2. SUBJECTIVE NORM AND QUALITY

H_0: There is no relationship between a software developer's perception of the subjective norm of TDD and the individual's perception of the environment's attitude of the difference in quality when performing TDD.

H_1: There is a relationship between a software developer's perception of the subjective norm of TDD and the individual's perception of the environment's attitude of the difference in quality when performing TDD.

The relationship between subjective norm and the developer's perception of the environment's attitude towards the change in quality when performing TDD can be seen in Figure 11. The rows in the table represent the responses to SubjectiveNorm, and the columns in the table represent the responses towards the environment's attitude to quality (SubjectiveNormQuality).



	Highly Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Highly Agree
Highly Disagree	3	6	8	18	12	3	2
Disagree	1	10	8	26	18	9	1
Somewhat Disagree	0	2	5	26	18	6	3
Neutral	2	2	16	65	66	36	15
Somewhat Agree	0	1	3	20	42	39	9
Agree	0	2	0	12	28	52	13
Highly Agree	0	2	3	15	19	45	87

FIGURE 11 SUBJECTIVE NORM AND QUALITY

The Kruskal-Wallis resulted in a p-value of $2.2 * 10^{-16}$ and a Kruskal-Wallis chi squared value of 232.26. The p-value leads us to reject the null hypothesis and conclude that there is a statistically significant relationship between the two variables. The high chi-squared value may indicate that the relationship may not be specified correctly in the conceptual model.

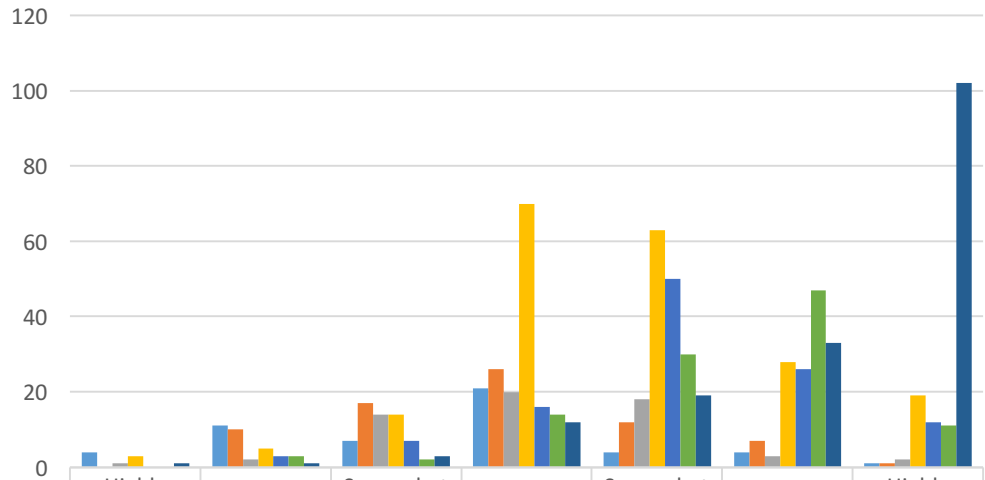
When running the chi-squared tests on the contingency tables, it results in a p-value of $2.2 * 10^{-16}$ and a chi-squared value of 143.88. Similar to the Kruskal-Wallis test, this leads us to reject the null hypothesis and conclude that there is a significant relationship between the two variables. Again, the high chi-squared value is a cause for concern.

5.6.3. SUBJECTIVE NORM AND MAINTAINABILITY

H_0: There is no relationship between a software developer's perception of the subjective norm of TDD and the individual's perception of the environment's attitude of the difference in the maintainability of the code when performing TDD.

H_1: There is a relationship between a software developer's perception of the subjective norm of TDD and the individual's perception of the environment's attitude of the difference in the maintainability when performing TDD.

The relationship between subjective norm and the developer's perception of the environment's attitude towards the maintainability of the code when performing TDD can be seen in Figure 12. The rows in the table represent the responses to SubjectiveNorm, and the columns in the table represent the responses towards the environment's attitude to maintainability (SubjectiveNormMaintainability).



	Highly Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Highly Agree
Highly Disagree	4	11	7	21	4	4	1
Disagree	0	10	17	26	12	7	1
Somewhat Disagree	1	2	14	20	18	3	2
Neutral	3	5	14	70	63	28	19
Somewhat Agree	0	3	7	16	50	26	12
Agree	0	3	2	14	30	47	11
Highly Agree	1	1	3	12	19	33	102

FIGURE 12 SUBJECTIVE NORM AND MAINTAINABILITY

The Kruskal-Wallis resulted in a p-value of $2.2 * 10^{-16}$ and a Kruskal-Wallis chi squared value of 269.21. The p-value leads us to reject the null hypothesis and conclude that there is a statistically significant relationship between the two variables. The high chi-squared value may indicate that the relationship may not be specified correctly in the conceptual model.

When running the chi-squared tests on the contingency tables, it results in a p-value of $2.2 * 10^{-16}$ and a chi-squared value of 204.07. Similar to the Kruskal-Wallis test, this leads us to reject the null hypothesis and conclude that there is a significant relationship between the two variables. Again, the high chi-squared value is a cause for concern.

5.7. INTENTION

The conceptual model describes three different influences of a developer's intention to perform TDD, namely their attitude towards TDD, the subjective norm of the environment the developer operates in, and lastly the perceived behavioural control. The following subsection will display the results of tests where these relationships are tested.

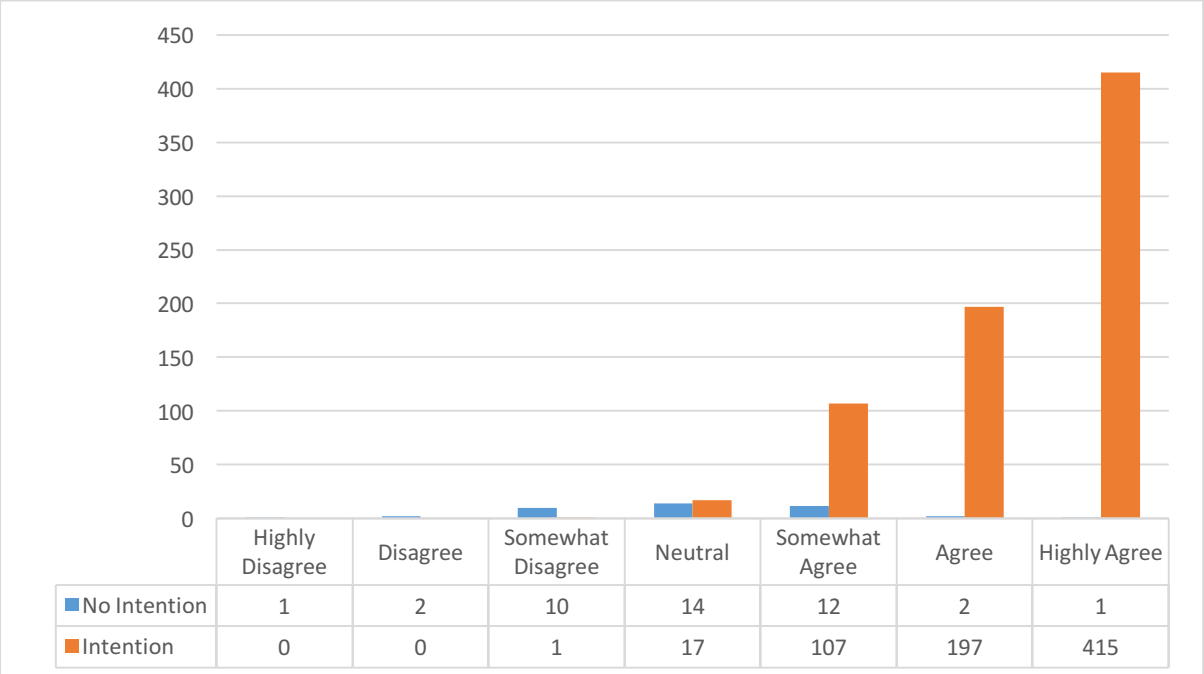
Perceived behavioural control, the developer's belief in whether they are able to perform the task at hand, is represented by two measures in this study: the developer's experience in performing TDD, and their perception of the difficulty of performing TDD. Because these are represented by ordinal data, these results of these two measures couldn't be added together, as they are not numeric and on a continuous scale. Therefore, these two variables remained as separate influences on the Intention variable when conducting statistical tests.

5.7.1. INTENTION AND ATTITUDE

H_0: There is no relationship between a software developer's attitude and intention to perform TDD.

H_1: There is a relationship between a software developer's attitude and intention to perform TDD.

The relationship between attitude and intention to perform TDD can be seen in Figure 13. The rows in the table represent the responses to intention, and the columns in the table represent the responses to attitude Attitude.



	Highly Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Highly Agree
No Intention	1	2	10	14	12	2	1
Intention	0	0	1	17	107	197	415

FIGURE 13 INTENTION AND ATTITUDE

The Kruskal-Wallis resulted in a p-value of $2.2 * 10^{-16}$ and a Kruskal-Wallis chi squared value of 340.32. The p-value leads us to reject the null hypothesis and conclude that there is a statistically significant relationship between the two variables. The high chi-squared value may indicate that the relationship may not be specified correctly in the conceptual model.

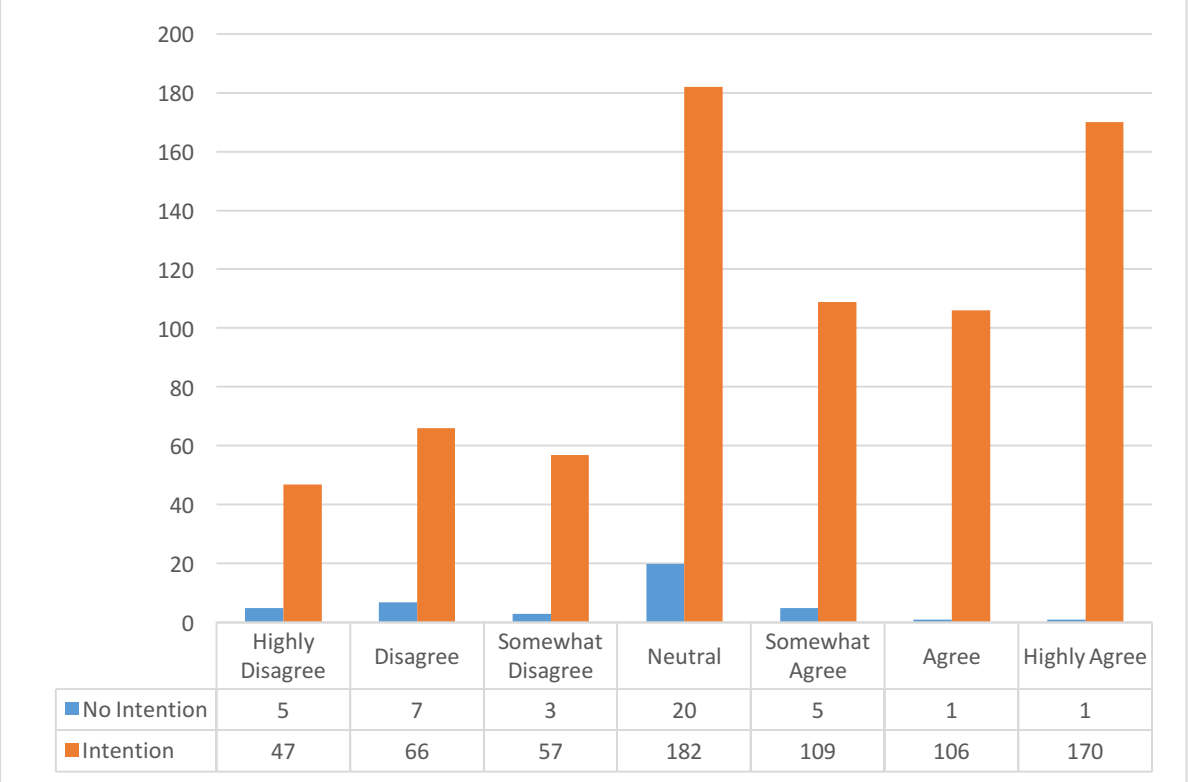
When running the chi-squared tests on the contingency tables, it results in a p-value of $2.2 * 10^{-16}$ and a chi-squared value of 322.22. Similar to the Kruskal-Wallis test, this leads us to reject the null hypothesis and conclude that there is a significant relationship between the two variables. Again, the high chi-squared value is a cause for concern.

5.7.2. INTENTION AND SUBJECTIVE NORM

H_0: There is no relationship between a software developer environment's subjective norm and intention to perform TDD.

H_1: There is a relationship between a software developer environment's subjective norm and intention to perform TDD.

The relationship between subjective norm and intention to perform TDD can be seen in Figure 14. The rows in the table represent the responses to intention, and the columns in the table represent the responses to subjective norm.



	Highly Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Highly Agree
No Intention	5	7	3	20	5	1	1
Intention	47	66	57	182	109	106	170

FIGURE 14 INTENTION AND SUBJECTIVE NORM

The Kruskal-Wallis resulted in a p-value of 0.0004192 and a Kruskal-Wallis chi squared value of 24.518. The p-value leads us to reject the null hypothesis and conclude that there is a statistically significant relationship between the two variables.

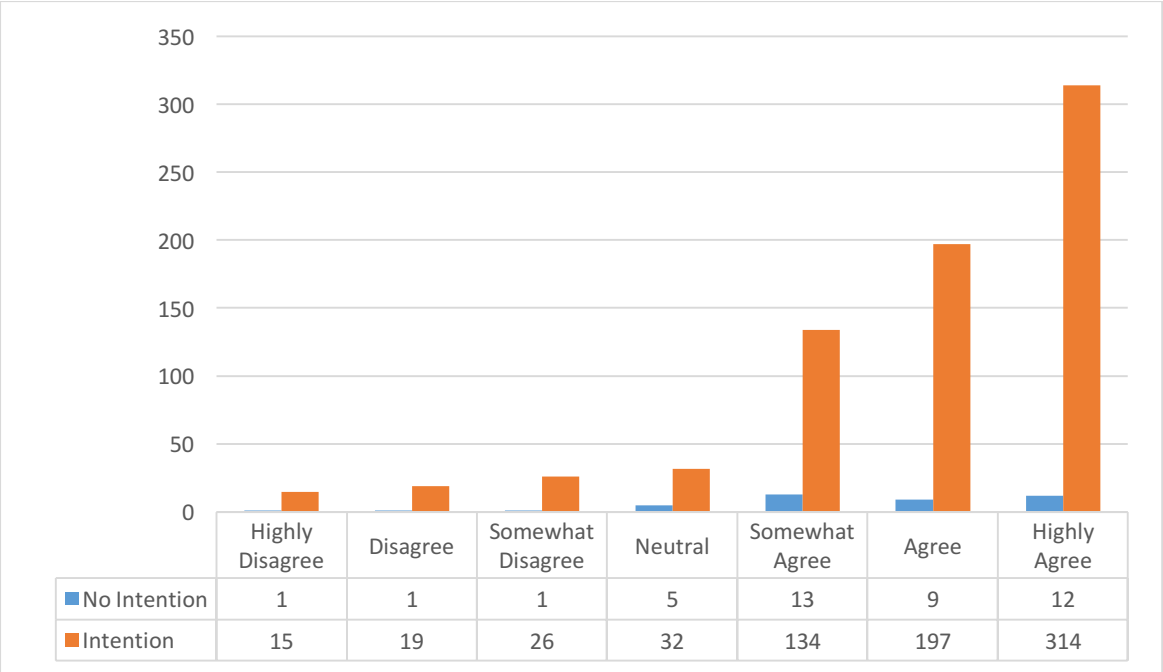
When running the chi-squared tests on the contingency tables, it results in a p-value of $3.165 * 10^{-5}$ and a chi-squared value of 20.722. Similar to the Kruskal-Wallis test, this leads us to reject the null hypothesis and conclude that there is a significant relationship between the two variables.

5.7.3. INTENTION AND TDD EXPERIENCE

H_0: TDD experience and intention to perform TDD are independent

H_1: TDD experience and intention to perform TDD are not independent

The relationship between TDD experience and intention to perform TDD can be seen in Figure 15. The rows in the table represent the responses to intention, and the columns in the table represent the responses to TDD experience.



	Highly Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Highly Agree
No Intention	1	1	1	5	13	9	12
Intention	15	19	26	32	134	197	314

FIGURE 15 INTENTION AND TDD EXPERIENCE

The Kruskal-Wallis resulted in a p-value of 0.09886 and a Kruskal-Wallis chi squared value of 10.678. The p-value leads us to not reject the null hypothesis and conclude that TDD experience and intention to perform TDD are independent.

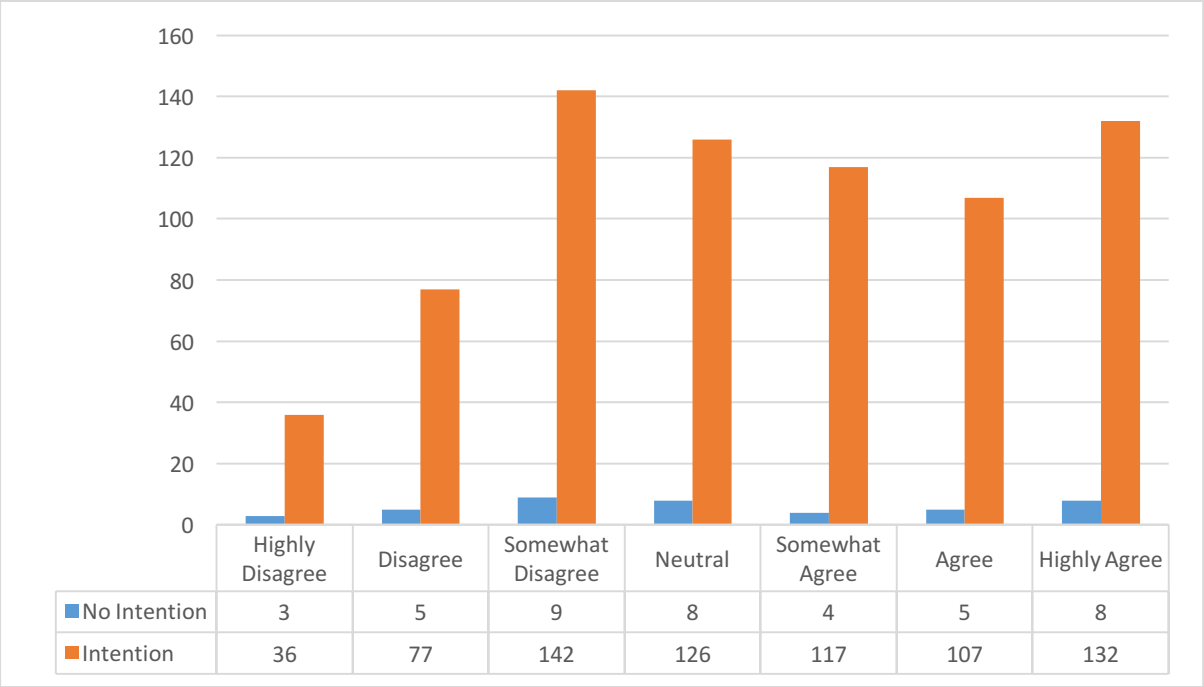
When running the chi-squared tests on the contingency tables, it results in a p-value of 0.08085 and a chi-squared value of 5.0304. Similar to the Kruskal-Wallis test, this leads us not to reject the null hypothesis and conclude that TDD experience and intention to perform TDD are independent.

5.7.4. INTENTION AND TDD DIFFICULTY

H_0: TDD difficulty and intention to perform TDD are independent

H_1: TDD difficulty and intention to perform TDD are not independent

The relationship between TDD difficulty and intention to perform TDD can be seen in Figure 16. The rows in the table represent the responses to intention, and the columns in the table represent the responses to TDD difficulty.



	Highly Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Highly Agree
No Intention	3	5	9	8	4	5	8
Intention	36	77	142	126	117	107	132

FIGURE 16 INTENTION AND TDD DIFFICULTY

The Kruskal-Wallis resulted in a p-value of 0.9273 and a Kruskal-Wallis chi squared value of 1.9155. The p-value leads us to not reject the null hypothesis and conclude that TDD experience and intention to perform TDD are independent.

When running the chi-squared tests on the contingency tables, it results in a p-value of 0.6097 and a chi-squared value of 0.98943. Similar to the Kruskal-Wallis test, this leads us not to reject the null hypothesis and conclude that TDD experience and intention to perform TDD are independent.

5.8. RESPONDENT COMMENTS

Within the survey, an optional question was asked for additional feedback from the respondents. Several respondents gave their feedback on their view of TDD.

One common view on whether to practice TDD was the type of applications that are being developed. Many respondents indicated that for new ‘greenfield’ applications, TDD was beneficial as it was easy to develop software in such a way that it is testable without having to change existing code to make it testable. It was also indicated for existing ‘brownfield’ applications, legacy code and where modifications were made to third-party applications, TDD was difficult to practice. The complexity of the application was also a determinant for practicing TDD, with some respondents indicating that simple applications without much business logic, such as data-driven applications were data is retrieved and displayed don’t require TDD, because there isn’t much to test.

Many respondents indicated their preference for practicing TDD, highlighting how it has improved their software development practice. A common theme that kept recurring was how it has improved the quality of the software developed and reduced the number of defects that have been found. Some also mentioned how it helped them in improving the design of the code because it forces one to “think about behaviour before thinking about implementation”. An example of such a comment was a respondent who said:

"Using outside-in TDD has had the biggest impact on the way I design and write code."

This relates to the attitude towards quality construct proposed in the theoretical model. Because there is a perception that practicing TDD leads to improved quality, this leads to a positive attitude towards TDD, and influences the software developer to intend to practice TDD.

Another common theme that was picked up by the comments was the increased time it takes to practice TDD. A few comments mentioned how there was increased time at the beginning when writing the initial functionality, but this resulted in decreased time later because of reduced defects introduced into the software. Because of the increased time, it was difficult to practice TDD in a few of the organizations where some respondents worked, as this resulted in an increase budget which was hard to justify. Some mentioned how some of the increased time is unnecessary as much time is spent setting up the testing tool on developer machines, and fixing tests instead of fixing functionality as the application progresses.

This is in line with the quality factors relating to both the attitude and subjective norm constructs of the theoretical model. Developers mentioned how it is hard 'to justify' the increased time taken to develop software, indicating a structural norm in the context in which they develop their software. Therefore, if the norm in the context isn't open to the perceived increased time, the developer is less likely to intend to practice TDD, as was stated in the theoretical model. Similarly, some individual developers some developers state how there is increased time at the beginning but reduced time later. This shows that there is a belief that TDD has a positive outcome on the quality of the code (attitude towards quality construct) and both long-term and short-term beliefs in the outcome on the time taken (attitude towards time construct).

Knowledge and experience in TDD came out as an important caveat in realising the benefits that can be accrued when practicing TDD. Many mentioned how TDD is difficult to practice especially at the beginning, and lack of knowledge and understanding when practicing TDD can actually result in badly written code which is hard to maintain. Training both at university level and at occupational level was recommended by respondents.

Within the theoretical model, experience and difficulty were constructs that influence the perceived behavioural control, which in turn influence the intention to perform TDD. The respondent comments are aligned with the constructs proposed in the theoretical model, as users mentioned how the difficulty is a barrier to practicing TDD. Respondents also mentioned that with more training and experience, TDD would become easier, again finding alignment with the theoretical model.

5.9. SUMMARY OF CHAPTER

Collected data was analysed to test whether it fit the proposed model, as well as to test the research questions stated in the introduction. The model was validated through confirmatory factor analysis, and a moderate model fit found through RMSEA. Through testing the hypotheses using Kruskal-Wallis test, chi-square tests and ordered logistic regression, it was found that for attitude and subjective norm, there does exist a relationship between the factors and their determinants. The factors that substituted for perceived behavioural control are found not to have a statistically significant relationship with intention. Within the comments, TDD was found to have positive effects on design, quality and maintainability of code, but some mentioned the negative effect on time taken and the need for knowledge and experience for TDD to be effective.

6. DISCUSSION AND CONCLUSION

The dynamic nature of modern software development and its challenges were discussed at the beginning of this dissertation. Agile software development methods were discussed as a way of mitigating some of the challenges, such as constantly changing requirements, resulting in constantly changing code, increasing the potential for the introduction of defects.

TDD is seen as a way to mitigate the risk of introducing regression defects into the code when changes are made, as existing functionality in the code is continually tested whenever the test suite is run, ensuring that defects are picked up soon and rectified. Because of the importance of TDD as a tool for increasing the quality of the software being developed, the research question of what factors influence developers intention to practice TDD is of importance.

The theoretical model of the TPB has been used to understand the factors affecting intention, namely attitude towards the behaviour, subjective norm and perceived behavioural control. This research questions posed that attitude was based on the beliefs about the effects of practicing TDD on the time spent, on the quality, on the developer's efficiency, and on the maintainability of the code base. The second question posed that subjective norm was based on the perceived beliefs by others on the effect of TDD on the time taken, the quality of the code, and the maintainability of the code. The third research question looked at perceived behavioural control, and posed that intention to perform is affected by the perceived difficulty of practicing TDD, and the number of years of experience.

The data showed that there is a relationship between the proposed determinants of attitude and attitude, as well as a relationship between the proposed determinants of subjective norm and the subjective norm. In the case of attitude though, the ordinal logistic regression that was used could not specify a regression function, indicating that the model was not specified correctly, and that more items may be needed. There was also a significant relationship between attitude and intention to perform TDD. In the case of subjective norm, a relationship was shown between the determinants of subjective norm and subjective norm, and the ordinal logistic regression confirmed that the proposed determinants did indeed have the predicted effects on the subjective norm. Similar to attitude, there exists a statistically significant relationship between subjective norm and intention to perform TDD. Lastly, it is interesting to note that the factors that were substituted for subjective norm, perceived difficulty of TDD and experience, did not have a statistically significant relationship with the intention to perform TDD. This shows that the TPB doesn't align with the data, and the data is more suited to the Theory of Reasoned Action, the precursor to TPB, which didn't include the Perceived Behavioural Control factor.

An interesting point that can be observed from the research questions and the data used to answer them is that the biggest determinant to performing TDD is not necessarily the level of skill of the developer, but subjective beliefs about the outcomes of performing the practice. When discussing the theoretical model earlier, Giddens' structuration theory was discussed which posited that structure is created through agents' interpretation of artefacts, and this structure in turn affects the behaviour of the agent. A similar thing has been found here. How a developer subjectively interprets the artefact of a unit test affects whether they practice TDD, as well as affecting the subjective norm of the environment they find themselves in (Jones & Karsten, 2008).

Another interesting finding is that within the comments submitted by the respondents, many mentioned that the difficulty of TDD may be a barrier for people, but as can be seen from the hypotheses, experience and perception of difficulty were actually independent of intention to perform. This shows that perceived difficulty of the task may not be the biggest barrier, to performing

TDD, but what the attitude of the developer is, and their perception of what is acceptable in the environment they work in.

Similarly, case studies have shown that TDD increases the time taken to complete software development tasks, but many of the respondents indicate that it has a positive effect on the time taken. Within the comments given by the respondents, a few mentioned that there is an increase in time, but the majority of respondents mentioned a positive effect on time. A reason for this is that the case studies which were analysed were cross-sectional, showing the effects of TDD on one project. But research and the results from this study have also shown TDD to increase the maintainability and changeability of the code, making it easier and less time consuming to make changes in the long run, something not picked up in a cross-sectional study. Yet, in the shorter term, there may be a perception that it doesn't have a negative effect on time taken, and this perception may not be empirically validated by the developer.

6.1. CONTRIBUTIONS TO PRACTICTIONERS

These insights are important for organisations who would like to introduce the practice of TDD, or would like it to get more acceptance among the developers in their organisation. What many may do is to focus on training programs and exercises to teach individuals the skill of TDD, but training programmes that just focus on the skill of TDD without convincing individuals and teams of its benefits may not be effective. On the other hand, if developers are convinced of its effectiveness in improving the quality and maintainability of software developed, developers may be more likely to engage and pursue training because the intention to perform the activity is already there.

Software teams and organisations hoping to remain competitive in the long term within this increasingly dynamic software industry need to have software quality as a focus and an objective. TDD is an approach that can contribute to this, and adopting the practice within software development teams can lead to reduced defects, and more satisfied customers.

6.2. LIMITATIONS OF THE STUDY

The various statistical methods that were used led us to reject the null hypotheses and conclude there the model was supported by the data (except experience and difficulty factors that made up the Perceived Behavioural Control construct), but anomalies were found in many of the statistical methods employed, leading to conclusion that the model is missing some constructs (especially the factors that make up the Attitude construct). These anomalies were picked up by the statistical tool R when conducting logistic regression to understand how the measures related to the factors they were measuring (attitude and subjective norm).

An additional limitation was that most of the responses were ordinal, and statistical techniques to understand and infer relationships between ordinal data are limited. The Kruskal-Wallis tests and chi-square tests based on contingency tables, as well as ordinal-logistic regression was used, but if the data was along a continuous scale and normality could be assumed, there would be a larger variety of more common statistical techniques that could have been employed to test the hypotheses.

Finally, the sample that was chosen was not representative of the software developer population. A high percentage of the sample practice TDD, something that is assumed not to be true in practice. The sample can help in understanding why those who do practice TDD actually do so, but it we cannot generalize across the entire software development community, assuming that it is not the majority who practice TDD.

6.3. FUTURE RESEARCH

Within the literature review, many of the benefits and drawbacks of TDD were used in the construction of the model came from individual and disparate case studies. A more in-depth, qualitative study of the different determinants Attitude, Subjective Norm as well as Perceived Behavioural Control would be required to better understand these and build a better and more comprehensive model that explains the determinants of intention to perform TDD. From these studies can a study such as this one be conducted and produce generalizable results for the body of knowledge.

Additionally, the benefits and negative effects of TDD need to be empirically studied in a manner that can be generalizable and start as a foundation for the increasing body of knowledge on the subject.

This research has provided a starting point for the development of a more comprehensive model to understand why software developers intend to perform TDD. It will serve as useful input for further research to better understand the determinants, from a wider and more diverse sample space which is more representative of the software developer population worldwide.

6.4. CONCLUSION

The purpose of this research was to understand the influences and factors behind the intention of software developers performing TDD. The literature highlighted several benefits and challenges of performing TDD. The intention to perform TDD was analysed through the TPB as the underlying conceptual framework.

The data that was collected did not fully support what was posited in the literature review and the conceptual model. The attitude construct and its determinants (the attitude towards the time taken, the effect on quality, and the maintainability) support the conceptual framework . Similarly the subjective norm construct and its determinants (perceived attitude of the environment (e.g. the software development team) towards quality, time taken and maintainability) support the conceptual framework. The perceived behavioural control constructs of difficulty and experience do not support the theoretical framework in a statistically significant way.

This findings of this research will be useful to practitioners who would like to adopt the practice of TDD in their teams and organizations, as well contribute to the body of knowledge for researchers who study how software teams operate.

7. APPENDICES

7.1. RESEARCH INSTRUMENT

Dear Participant,

My name is Patrick Kayongo, and I am a Masters of Commerce student at the University of Cape Town majoring in Information Systems. As part of the Masters in Information Systems at the University of Cape Town, I am conducting research to understand what the factors are that affect an individual software developer's intention to practice test-driven development. As a software developer, I am inviting you to participate in this research study by completing the survey.

The survey will take approximately 15 minutes of your time to complete. There is no compensation for completing the research. There are also no known risks. All names and identifications of individuals and organizations will be kept confidential to ensure complete anonymity. Your participation in this research is voluntary. You can choose to withdraw from the research at any time.

Thank you for assisting me in completing this research project. The research aims to gather responses from software developers to understand whether they practice test-driven development, and to understand the reasons behind their intention to engage or disengage in the practice of test-driven development.

Kind regards,
Patrick Kayongo
MComm (Information Systems) Candidate
University of Cape Town
+27 72 394 7182
pat.kayongo@gmail.com

Questionnaire.

1. Do you work for an organization?

Work for myself	Work for an organization

2. If you work for an organization, is test-driven development permitted in your organization?

Yes	No

3. Are you an in-house software developer for a company, or do you work for a software development consultancy?

In-house software developer	Software development consultancy	Other

4. Do you work on an individual product for an organization, or do you work on many projects for many clients?

Individual project	Many projects for many clients	Both

5. Is the work you do mainly on new projects creating new applications, or maintaining and making enhancements to existing applications?

New applications	Maintaining existing applications	Both

6. For how many years have you been developing software?
7. Do you practice test-driven development?

Yes	No

8. If not, do you want to practice test-driven development?

Yes	No

Rate the following statements depending on how much you agree or disagree.

9. Test driven development has positive outcomes on a software development project:

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

10. Test-driven development has a positive effect on the quality of software being developed.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

11. Test-driven development unnecessarily increases the time spent developing software.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

12. Test-driven development decreases the number of defects of software developed.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

13. Test-driven development increases the maintainability of the software being developed.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

14. Test-driven development increases developer efficiency.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

15. Test driven development makes it easier to make changes to the software.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

16. I feel that test-driven development is seen as a positive thing in software development.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

17. Test-driven development is encouraged in the organization I work for.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

18. The software developer community I follow (e.g. blogs, talks, etc.) see test-driven development as a positive thing.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

19. My organization views the practice of test-driven development as an unnecessary increase in development time.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

20. My team / organization believes that test-driven development increases the maintainability and readability of the code.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

21. My team / organization believes that test-driven development decreases the number of defects in the software created.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

22. My team / organization believes that test-driven development makes it easier to make changes to existing software.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

23. Writing unit tests before writing functionality is difficult.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

24. I have experience in test-driven development.

Strongly disagree	Disagree	Somewhat disagree	Neutral	Somewhat agree	Agree	Strongly agree

25. Do you have any comments to add?

8. REFERENCE LIST

Ajzen, I. (1991). The Theory of Planned Behavior. *Organizational Behaviour and Human Design Processes*, *50*, 179–211.

Anderson, P. (2008). Complexity Theory and Organization Science. *Organization Science*, *10*(3), 216–232.

Bandura, A. (1986). *Social foundations of thought and action: A social cognitive theory*. Prentice-Hall, Inc.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., … Thomas, D. (2001). Manifesto for Agile Software Development.

Bertolino, A. (2007). *Software Testing Research: Achievements, Challenges, Dreams*. *Future of Software Engineering*.

Bhat, T., & Nagappan, N. (2006). Evaluating the efficacy of test-driven development: industrial case studies. In *Proceedings of the 2006 ACM/IEEE international …* (pp. 1–8). http://doi.org/10.1145/1159733.1159787

Boehm, B. W. (1988). Spiral Model of Software Development and Enhancement. *Computer*. http://doi.org/10.1109/2.59

Bowes, J. (2015). Kanban vs Scrum vs XP – an Agile comparison. Retrieved from http://manifesto.co.uk/kanban-vs-scrum-vs-xp-an-agile-comparison/

Causevic, A., Sundmark, D., & Punnekkat, S. (2011). Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation* (pp. 337–346). http://doi.org/10.1109/ICST.2011.19

Chan, Y. (1997). Learning and Understanding the Variance-by-Ranks Test for Differences Among Three or More Independent Groups. *Physical Therapy*, *77*(12).

Chen, W., & Hirschheim, R. (2004). A paradigmatic and methodological examination of information systems research from 1991 to 2001. *Information Systems Journal*, *14*, 197–235.

Chiva-Gomez, R. (2004). Repercussions of complex adaptive systems on product design management. *Technovation*, *24*(9), 707–711. http://doi.org/10.1016/S0166-4972(02)00155-4

Cialdini, R. B., Reno, R. R., & Kallgren, C. A. (1990). A focus theory of normative conduct: Recycling the concept of norms to reduce littering in public places. *Journal of Personality and Social Psychology*, *58*(6), 1015–1026. http://doi.org/10.1037/0022-3514.58.6.1015

Cockburn, A., & Highsmith, J. (2001). Agile Software Development:The People Factor. *Computer*, *34*(11), 131–133. http://doi.org/10.1109/2.963450

Dybå, T., & Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information and Software Technology*, *50*(9-10), 833–859. http://doi.org/10.1016/j.infsof.2008.01.006

Erickson, J., Lyytinen, K., & Siau, K. (2005). Agile Modeling, Agile Software Development, and Extreme Programming: The State of Research. *Journal of Database Management*, *16*(4), 88.

Farcic, V. (2014). Tests as documentation.

Federal Highway Administration. (2004). *Software Reliability: A Preliminary Handbook*.

Fowler, M. (2005). TestDrivenDevelopment.

Francis, O., Francis, A. J. J., Eccles, M. P., Johnston, M., Walker, A., Grimshaw, J., … Bonetti, D. (2004). *Constructing Questionaire Based On The Theory of Planned Behaviour: A Manual for HEalth Service Researchers*. Retrieved from http://openaccess.city.ac.uk/1735/

Gartner. (2014). Gartner Says 4.9 Billion Connected "Things" Will Be in Use in 2015. Retrieved from http://www.gartner.com/newsroom/id/2905717

Gelperin, D., & Hetzel, B. (1988). The growth of software testing. *Communications of the ACM*, *31*(6), 687–695. http://doi.org/10.1145/62959.62965

George, B., & Williams, L. (2003). An initial investigation of test driven development in industry. In *Proceedings of the ACM Symposium on Applied Computing* (pp. 1135–1139). http://doi.org/10.1145/952532.952753

Giddens, A., & Pierson, C. (1998). *Conversations with Anthony Giddens: Making sense of modernity*.

Hair, J., Anderson, R. E., Tatham, R. L., & Black, W. C. (1998). *Multivariate data analysis*.

Hansson, D. H. (2014). TDD is dead. Long live testing. Retrieved from http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html

Hetzel, W. C. (1991). *The complete guide to software testing*. John Wiley & Sons, Inc.

Highsmith, J. (2002). What Is Agile Software Development? *The Journal of Defense Software Engineering*, *15*(10), 4–9. http://doi.org/10.1109/2.947100

Hooper, D., Coughlan, J., & Mullen, M. (2008). Structural Equation Modelling : Guidelines for Determining Model Fit Structural equation modelling : guidelines for determining model fit, *6*(1), 53–60.

Jacobs, J., van Moll, J., Krause, P., Kusters, R., Trienekens, J., & Brombacher, A. (2005). Exploring defect causes in products developed by virtual teams. *Information and Software Technology*, *47*(6), 399–410. http://doi.org/http://dx.doi.org/10.1016/j.infsof.2004.09.006

Javed, T., Maqsood, M. e, & Durrani, Q. S. (2004). A study to investigate the impact of requirements instability on software defects. *ACM SIGSOFT Software Engineering Notes*, *29*(3), 1. http://doi.org/10.1145/986710.986727

Jhanwar, R., & Yaryan, T. (2012). *Dynamic software update*. *Technical Report ITI-SIDI-2012/004*. Retrieved from http://www.google.com/patents?hl=en&lr=&vid=USPAT7251812&id=oMWAAAAAEBAJ&oi=fnd&dq=Dynamic+Software+Update&printsec=abstract

Jones, M. R., & Karsten, H. (2008). Giddens's structuration theory and information systems research. *Mis Quarterly*, *32*(1), 127–157. http://doi.org/Article

Larman, C., & Basili, V. R. (2003). Iterative and incremental development: A brief history. *Computer*, *36*(6), 47–56. http://doi.org/10.1109/MC.2003.1204375

Madden, T. J., Ellen, P. S., & Ajzen, I. (1992). A comparison of the theory of planned behavior and the theory of reasoned action. *Personality and Social Psychology Bulletin*, *18*(1), 3–9.

Madeyski, L. (2006). *The Impact of Pair Programming and Test-Driven Development on Package Dependencies in Object-Oriented Design–An Experiment*. *Product Focused Software Process Improvement* (Vol. 4034). Retrieved from http://www.scopus.com/scopus/inward/record.url?eid=2-s2.0-33746265818&partnerID=40\nhttp://scholar.google.ca/scholar?start=400&q=allintitle:++("test+driven"+OR+"test+first"+OR+tdd)+(development+OR+design+OR+programming+OR+approach)&hl=en&as_sdt=0,5#59

Malaiya, Y. K., & Denton, J. (1999). Requirements volatility and defect density. In *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443)* (pp. 1–13). http://doi.org/10.1109/ISSRE.1999.809334

Martin, R. C. (2014). Monogamous TDD. Retrieved from https://blog.8thlight.com/uncle-bob/2014/04/25/MonogamousTDD.html

Mathur, S., & Malik, S. (2010). Advancements in the V-Model. *International Journal of Computer Applications*, *1*(12), 29–34. http://doi.org/10.5120/266-425

Nevitt, J., & Hancock, G. R. (2000). Improving the Root Mean Square Error of Approximation for Nonnormal Conditions in Structural Equation Modeling. *The Journal of Experimental Education*, *68*(3), 251–268. http://doi.org/10.1080/00220970009600095

Nurmuliani, N., Zowghi, D., & Powell, S. (2004). Analysis of requirements volatility during software development life cycle. *Software Engineering Conference, 2004. Proceedings. 2004 Australian*. http://doi.org/10.1109/ASWEC.2004.1290455

Olsson, U. (1979). Maximum likelihood estimation of the polychoric correlation coefficient. *Psychometrika*, *44*(4), 443–460. http://doi.org/10.1007/BF02296207

Paulk, M. C. (2002). *Agile Methodologies and Process Discipline*. *Institute for Software Research*. Retrieved from http://repository.cmu.edu/cgi/viewcontent.cgi?article=1012&context=isr

Pendharkar, P. C., & Rodger, J. A. (2009). The relationship between software development team size and software development cost. *Communications of the ACM*, *52*(1), 141–144. http://doi.org/10.1145/1435417.1435449

Perpignand, E. (2010). Automated Unit Tests as Documentation.

Rhodes, R. E., & Courneya, K. S. (2003). Investigating multiple components of attitude, subjective norm, and perceived control: An examination of the theory of planned behaviour in the exercise domain. *British Journal of Social Psychology*, *42*(1), 129–146.

Savalei, V., & Bentler, P. M. (2006). Structural Equation Modeling. *Handbook of Marketing Research: Uses, Misuses, and Future Advances*, 330–364. http://doi.org/10.4135/9781412973380.n17

Sayer, A. (2000). *Realism and Social Science. Radical philosophy reader* (Vol. 1). SAGE Publications. http://doi.org/10.4135/9781446218730

Spector, P. E. (1982). Behavior in organizations as a function of employee's locus of control. *Psychological Bulletin*, *91*(3), 482–497. http://doi.org/10.1037/0033-2909.91.3.482

Stacey, R. D. (1995). The science of complexity: and alternative perspective for strategic change processes. *Long Range Planning*, *28*(6), 124. http://doi.org/10.1016/0024-6301(95)99970-B

University of St Andrews. (n.d.). Analysing Likert Scale/Type Data, Ordinal Logistic Regression Example

in R. Retrieved from https://www.st-andrews.ac.uk/media/capod/students/mathssupport/OrdinalexampleR.pdf

Wappler, S., & Lammermann, F. (2005). Using Evolutionary Algorithms for the Unit Testing of Object-Oriented Software. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation* (pp. 1053–1060). ACM. http://doi.org/10.1145/1068009.1068187

Williams, L., Maximilien, E. M., & Vouk, M. (2003). Test-driven development as a defect-reduction practice. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* (pp. 1–12). http://doi.org/10.1109/ISSRE.2003.1251029

Zimmerman, B. J. (2000). Self-Efficacy: An Essential Motive to Learn. *Contemporary Educational Psychology*, *25*(1), 82–91. http://doi.org/10.1006/ceps.1999.1016