1-1-2011

# TDD Dynamics: Understanding the Impact of Test-Driven Development on Software Quality and Productivity

Yahha Rafique
*Ryerson University*

# TDD DYNAMICS: UNDERSTANDING THE IMPACT OF TEST-DRIVEN DEVELOPMENT ON SOFTWARE QUALITY AND PRODUCTIVITY

by

Yahya Rafique

Bachelor of Science

University of Toronto, 2008

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Canada, 2011

# Author's Declaration

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Signed:_____

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signed:_____

**TDD DYNAMICS: UNDERSTANDING THE IMPACT OF TEST-DRIVEN DEVELOPMENT ON SOFTWARE QUALITY AND PRODUCTIVITY**

Yahya Rafique

M. Sc. in Computer Science, 2011

Ryerson University, Toronto, Canada

# Abstract

Test-Driven Development (TDD) is one of the cornerstone practices of the Extreme Programming agile methodology. Today, despite the large scale adoption of TDD in industry, including large software firms such as Microsoft and IBM, its usefulness with regard to the quality and productivity constructs is still under question. Empirical Research has failed to produce conclusive results; all possible results have been reported for both constructs. This research adopts non-empirical measures to gain a deeper understanding of TDD. A two-phased approach has been undertaken towards the goal. The first phase involves conducting a meta-analysis of past empirical research. The meta-analysis quantitatively combines the results of individual empirical studies and identifies moderator variables that could potentially govern the performance of TDD. The second phase of the approach involves the construction of a simulation model of a TDD-based development process. The presented model further analyzes the impact of changes in moderator variables.

# Acknowledgements

I would like to begin by expressing my appreciation for my research supervisor Dr. Vojislav Misic. Over the past two years Dr. Vojislav has played a pivotal role in my research acting as a mentor, a source of inspiration and an expert from the field of software engineering. Without him, the completion of my research would not have been possible. I would like to thank my thesis defence committee members: Dr. Eric Harley, Dr. Jelena Misic and Dr. Marcus Santos for their time, patience and expertise in assessing my research. Also, I would like to thank Dr. Tomaz Dogsa from University of Maribor, Slovenia for his feedback during my research. I am indebted to my family who have supported me in every way up to this day. Their contribution towards shaping my life for the better is truly priceless. Lastly, and most importantly, I would like to thank God Almighty, Allah, for all the blessings I have been provided.

# Dedication

To my family.

# Table of Contents

# List of Tables

# List of Figures

# List of Appendices

# Chapter 1

# Introduction

## 1.1 Motivation

Software Quality and Development Productivity are two of the most important variables in software engineering research. Today's fast and dynamically changing work climate imposes the constraint of delivering software only of the highest quality in order to facilitate businesses in carrying out their day-to-day mission critical tasks. Additionally, the development team must maximize their productivity and deliver the final product within the minimal amount of time. Attempting to harmonize these opposing objectives, researchers in the field of software engineering have been engaged in formulating better ways of developing software.

Research efforts in this area have been formalized with the development of process models. The earliest of these models, the *Waterfall* model, was formulated by Royce in 1970 [74]. Today, four decades later, originating from discussion on Royce's model an array of software process models exist including the *Spiral* model [10], the *V-shaped model* [27], the *Incremental* model [51] etc.

Arguably, in this area of research the topic that has received the most widespread attention within the last decade is that of Agile methodologies. In 2001, a group of software pioneers met in Utah, U.S.A in hopes of attaining a common ground on possible solutions to issues facing contemporary software development. What emerged from their meeting is now known as the *Manifesto for Agile Development* [7]. The Agile paradigm, unlike previous ones, does not provide a concrete model specifying the activities to be carried out and the order in which they are to

occur. Rather, it lays out in abstract terms the foundational principles underlying a successful development process.

All of the agile development methodologies are rooted in iterative and incremental development (IID) [51]. In IID the system is incrementally developed through the repeated iteration of small design, code and test phases. Today, IID itself is widely practised in industry and is regarded by many as the "modern" replacement of the Waterfall model [51].

Presently, a variety of implementations of Agile principles, known as Agile methodologies, can be found, the most of popular of which are Extreme Programming (XP) [6], Scrum [78] and Feature-Driven Development (FDD) [69].

The Extreme Programming(XP) process is characterized by 12 core practices [6]. In recent years the XP process has by far been the most investigated subject within empirical research under the agile domain [20]. Despite the fairly large number of empirical studies there is little concrete evidence on the usefulness of XP; usefulness is meant to imply the effectiveness of XP in being able to positively influence the quality and productivity outcome constructs. According to Dyba & Dingsoyr [20], this is largely due to studies suffering from differences in design, quality, consistency and directness. In light of this issue, one approach that seems promising is to individually examine each of XP's practices and ascertain whether or not it contributes towards the high level goals of improving quality and productivity.

Test-Driven Development (TDD) is one of the cornerstone practices of a XP-based development process. Today TDD is being widely adopted in industry, including large software firms such as Microsoft and IBM [66], both as part of a larger adoption of an XP-style development process and as a stand-alone practice. Despite the large scale adoption of TDD, its usefulness with regard to software quality and productivity is still under question. Similar to XP, in previous years much empirical research has been conducted in the area of investigating the efficacy of TDD. However, this research has failed to produce conclusive results, and in fact, all possible outcomes—positive, negative, and neutral—have been reported for both software quality and developer productivity

improvements in empirical studies [47]. Why is it important to analyze the effectiveness of TDD? According to [79]:

"*TDD is the heart of XP's programming practices. Without it, all of XP's other technical practices will be much harder to use.*"

In view of the important role played by the TDD practice in the success of an XP implementation and the high rate of adoption of both XP and stand-alone TDD in industry, an analysis of the impact of TDD is warranted.

This research aims to adopt non-empirical measures to gain a deeper understanding of a TDD-based development process.

In the next section we provide a more detailed description of TDD. Also, we describe the two traditonal forms of development whose performances have been used to benchmark the performance of TDD in empirical trials. Following that, we briefly discuss the approach taken in this research. Finally, we conclude this chapter stating the contributions of this thesis and outlining the organization of the remaining chapters.

## 1.2 Background on TDD

The following section assumes, in accordance with literature on requirements gathering in the agile paradigm, that requirements have been collected in the form of user stories. A User Story is a description of the functionality to be implemented from the client's perspective [17]. It can be broken into story points during estimation, and subsequent implementation, where each story point represents a fraction of the functionality comprising the user story.

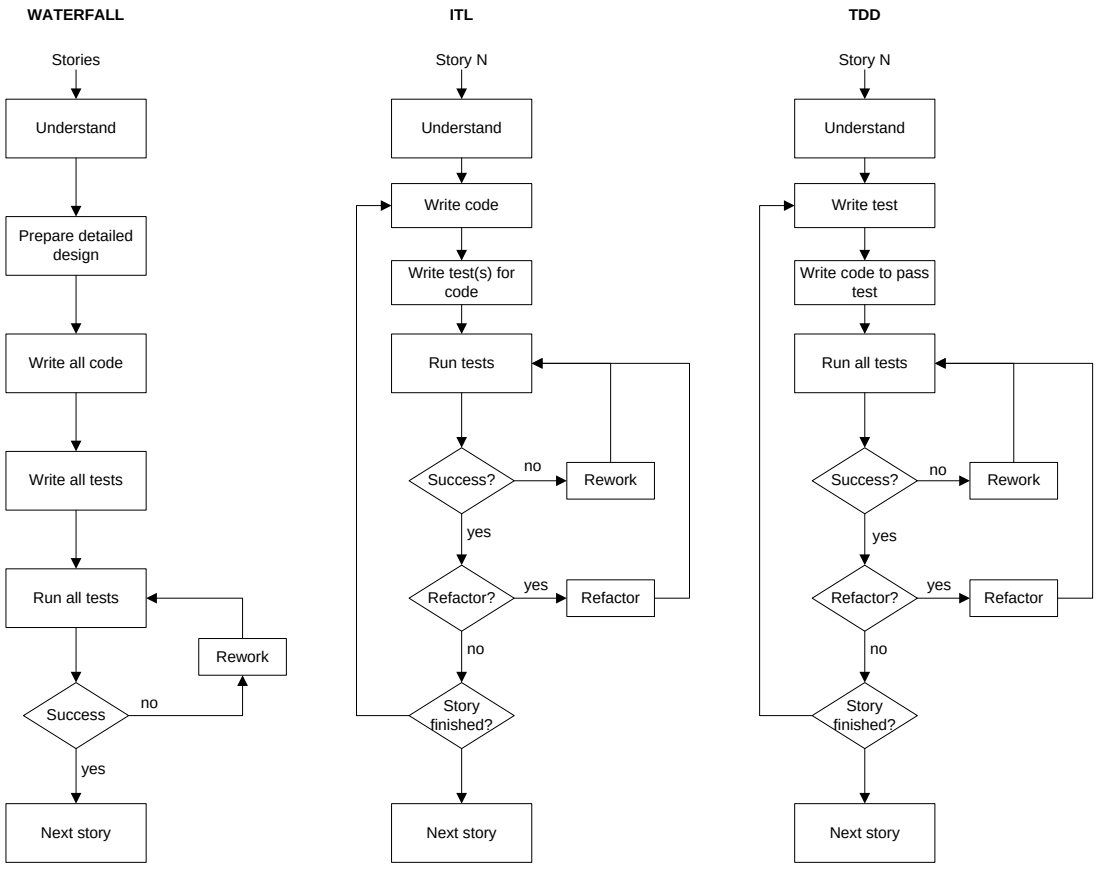



Figure 1.1: Comparison between Waterfall, ITL and TDD style development. (Adapted from Pancur & Ciglaric [70] and Janzen & Saiedian [41])

### 1.2.1 Waterfall and ITL-style development

The *Waterfall* model [74] prescribes a single iteration through the design, implementation and testing phases where the artifacts of each stage are completed and passed to the next stage of processing. A flowchart illustrating the development flow in a Waterfall-based process is shown in the leftmost diagram of Figure 1.1. Following this approach, all requirements are understood and a detailed design of the system is prepared upfront [41]. Following that, all the requirements are implemented in one go. Next, all tests related to the implementation functionality are written and run. Lastly, the code related to the failed tests is reworked until the respective tests pass. In a nutshell, the Waterfall approach prescribes that all user stories (i.e. requirements) are collectively designed, then implemented and finally tested.

Iterative Test-last (ITL) style development is a variant of the Waterfall model that is incremental and iterative. A flowchart illustrating the development flow in a ITL-based process is shown in the middle diagram of Figure 1.1. Following the ITL approach, the development of each story is spread out into multiple iterations. In each iteration a 'small' chunk of functionality expressed by the story is completed. Each iteration begins with the implementation of a chunk of functionality. Once completed, the implementation is succeeded by the preparation of tests related to that specific piece of functionality. This is followed by these tests being run and any failing code being reworked. After all tests pass, if necessary, the code is refactored before moving on to the next iteration. In this manner the entire story is completed iteratively piece-by-piece. An important difference between Waterfall-style and IT-style development is how the functionality is designed. In Waterfall-style development a detailed design is prepared upfront prior to implementation whereas when following the ITL approach only the high-level architecture is decided upfront with the intent of allowing the design to evolve as the implementation progresses, and be continually improved through regular refactorings.

### 1.2.2 TDD-style development

Test-Driven Development(TDD) is defined as the combination of Test-First development, in which unit tests are written before the implementation code needed to pass those tests [4], and refactoring, which includes restructuring a piece of code that passes the tests in order to reduce its complexity and improve its clarity, understandability, extendibility, and/or maintainability [5]. According to Beck [5], the individual who is accredited for formalizing the notion of TDD, the rhythm of TDD can be described by the following steps:

1. *Add a test*
2. *Run all tests and see the new one fail*
3. *Write just enough code to pass the test*
4. *Run all tests and see them all succeed*
5. *Refactor to remove duplication*

A flowchart illustrating the development flow in a TDD-based process is shown in rightmost diagram in Figure 1.1. TDD proposes a restructuring of the activities that make up a development process. In addition to being iterative and incremental, TDD brings two major changes. Firstly, as mentioned earlier, in each iteration development proceeds by one test with the test being written before the implementation code required to pass it. Secondly, a detailed design upfront is abandoned and replaced with the practice of refactoring working code frequently.

### 1.2.3 Comparison between traditional and TDD-style development

Stemming from the re-organization of the code-test sequence, the adoption of TDD requires a conceptual shift in developer mindset [40]. Developers are now forced to initially look at the system from the client's perspective thus contemplating the scenarios the code would be required to handle and determining potential breakpoints along execution paths prior to the actual implementation. Such a change in mindset cannot be expected to be brought about overnight and can take both Waterfall and ITL developers at least a couple of weeks to fully grasp [81].

When practicing TDD the burden of both designing and testing the system is partially placed on the developers themselves. Hence another prerequisite of conventional developers proficiently carrying out all TDD-related tasks is that they must improve their testing and design levels to a level comparable to that of dedicated professionals in these areas. In comparison with pure ITL-style development, this prerequisite disappears as developers are already performing design and testing related tasks.

Advocates of TDD claim that implementing the practice results in significant improvements in two variables over the life of the development cycle. Firstly, they claim that TDD results in large improvements in test coverage. TDD prescribes that a line of code is not written unless there is an accompanying failing test. Intuitively, in comparison with Waterfall-style development, due to this constraint a much higher level of test coverage is expected over the course of system development.

In comparison with ITL-style development, the difference in coverage is not likely to be as large as that with Waterfall-style development, however it is dependant on how rigorously the code is tested in absence of a TDD-like constraint.

Secondly, TDD supporters claim that the practice significantly improves the design quality. Both TDD and ITL take a different approach to design sometimes referred to as *Agile Design* [77]. In agile design the detailed architecture of the system is not preset, rather it is allowed to emerge incrementatlly over the course of development whilst making continual improvements via refactoring. At any point during system development the design only accounts for present needs but not speculative needs [92].

Thirdly, unlike both of the Test-Last approaches, in TDD tests are used to "drive" the design. Related to the "driven" aspect, in TDD the initial decomposition of the user story into smaller chunks is based on a focus towards independent testability, and the tests are intended to specify what the code should do which essentially is a design and analysis step [40]. Considering significant changes in the process leading to design decisions, one can hypothesize that the resulting design will exhibit different characteristics in terms of complexity, coupling and other design measures. However, empirical research investigating the design aspect of TDD has been inconclusive, and hence no definite claims can be made [47].

## 1.3 Research Approach

This research aims to analyze the efficacy of TDD in terms of two, arguably most important, outcome constructs: software quality and productivity. Typically, the former is assessed as external code quality, i.e., the number of defects per given code size unit (line of code or another suitable measure), while the latter is expressed as the number of given units produced by a developer in a given time frame (day or month) [23].

Empirical research analyzing the impact of TDD on these two constructs provides inconsistent

results. While some studies report large improvements with regard to one or both of these outcome constructs, others have the found the influence of the practice as not being significant and in some cases negative results have also been noted.

Due to the vast amount of empirical literature already available, we opted for a non-empirical approach to address this issue. The rationale behind this choice is as follows. Although empirical trials provide the most transparent view of the efficacy of a development practice/methodology, the generalizability of a trial's results is always a major cause of concern as the results are very dependant on the experimental setting in which the trial was conducted. Nevertheless, results common amongst multiple studies can, to a greater extent, be considered universally applicable. Following this intuition, this research intends to exploit existing research to obtain a better understanding of a TDD-based process.

In particular, a two-phased approach has been adopted in this research. The first phase involves conducting a meta-analysis of past empirical research. This research is not the first in its intent on combining results from previous empirical studies. However, all of the existing reviews take a qualitative approach towards combining results while this research undertakes a quantitative one. Additionally, the meta-analysis aims to identify moderator variables that could potentially govern the performance of TDD in a particular context.

The second phase of the presented approach involves the construction of a simulation model of a TDD-based development process. Software process simulation (SPS) offers an inexpensive alternative to empirical trials for analyzing the effects of policy changes such as the adoption/manipulation of development practices/methodologies. For this phase of the research the available literature on the effects of TDD is aggregrated for the construction of a System Dynamics (SD) simulation model of a TDD-based process. The intent of the modeling process is to contribute towards the larger scale effort of a SD model of a XP-based development process. As an application of the model, it is simulated to obtain a deeper insight into the impact of changes in the previously identified moderator variables on software quality.

## 1.4 Contributions of this thesis

This thesis has two main contributions:

- A meta-analysis of empirical studies on the impact of TDD on external quality and productivity. The analysis commences with an examination of the differences in rigor level amongst the selected studies. Following this, meta-analytical techniques [52] are applied to derive effect sizes for trials in the selected studies and subsequently a summary effect size. Next, the studies are grouped based on their experimental context, and then differences between the results of subgroups are analyzed. Lastly, the impacts of potential moderator variables are investigated.

- To the best of my knowledge, this research is the first to present a System Dynamics simulation model of a TDD-based process. The development process is modeled at the iteration level where the steps in each iteration are described in Figure 1.1. The scope of this model is limited to examining TDD as a stand-alone practice. As an application of the presented model, the impacts of changes in the moderator variables, identified earlier, are simulated and subsequently analyzed.

## 1.5 Organization of this thesis

This thesis is organized as follows:

Chapter 2 reviews the related work and is split into two sections the first of which provides details of previous reviews on TDD, corresponding to the first contribution of this thesis, and the latter providing a brief modeling description and results of previous studies that employ SD simulation modeling to to analyze practices/methodologies within the agile paradigm.

Chapter 3 provides details of the meta-analysis on TDD outlining the methodology of study selection and effect size calculation, the results of the analysis and evidences on the identified

potential moderator variables.

Chapter 4 presents the TDD SD model commencing with an overview of SD modeling, following that with a description of the TDD model and concluding with an analysis of the impact of the moderater variables.

Finally, Chapter 5 provides a summary of the present work and highlights promising directions for future research.

Extra details regarding the research highlighted in Chapter 3 and Chapter 4 are given in Appendix A and Appendix B respectively.

# Chapter 2

# Related Work

## 2.1 Previous Reviews on TDD

This section presents a summary of the methodology and results of three previous reviews of empirical studies on TDD. The presented reuslts are limited to those on the external quality and productivity outcome constructs.

Siniaalto [82] examined the findings of 13 studies. Studies were classified into three categories depending on whether they were conducted in an industrial, semi-industrial, or academic context. With regard to quality, strong improvements were found amongst the studies conducted in the industrial context; however, in the other two groups improvements varied between significant and small to non-existent. With regard to productivity, studies within the industrial and semi-industrial groups reported all possible outcomes, while the results from the academic group were limited to non-negative impact.

Kollanus [47] conducted a review of 40 studies. Studies were classified based to their experimental design into three categories: controlled experiments, case studies, and other studies. Weak evidence was found for an overall improvement in quality whereas moderate evidence was found for an overall drop in productivity. Several studies were found to report that the improvement in quality coincided with a drop in productivity. Using intuition given in one of the analyzed studies [37], it was hypothesized that an improvement in quality, and subsequently a drop in productivity, could be the result of increased test effort rather than an intrinsic benefit of TDD.

Turhan *et al.* [88] conducted a review of 22 studies; this review was further summarized and

contrasted with expert opinion in [80]. Similar to Kollanus [47], studies were divided into controlled experiments, pilot studies, and industrial-use studies; additionally, studies were classified as being high-rigor or low-rigor. The rigor rating of a study was determined by three variables: developer experience, process definition and scale of study (determined by the number of subjects and task size). Rather than formulating an overall conclusion with respect to each of the outcome constructs, the authors commented on the performance of TDD in each of the three groups. The majority of the industrial-use and pilot studies reported a positive effect on quality but results on controlled experiments were inconclusive. With regard to the productivity construct, results were inconsistent with controlled experiments, pilot studies and industrial studies suggesting an improvement in productivity, mixed effects and worse productivity respectively. After filtering low-rigor studies, results related to quality in each category were found to be more spread out with the results from the controlled experiments and the industrial studies leaning towards no difference in quality and the results from the pilot studies still leaning towards an improvement. Even when low-rigor studies were not being considered, no definite conclusions could be derived regarding the productivity impact of TDD in each of the three categories.

All three reviews described here have similar shortcomings that are addressed in the forthcoming meta-analysis. In particular, my research differs from them in a number of important aspects:

1. Focus is placed on externally observable variables, namely defect density and developer productivity, both of which are directly measurable. Internal code quality is not considered as there is no widely accepted measure for it. Also, test coverage is not considered as studies differ in the type of coverage (line, branch, method etc.) they report.

2. A quantitative approach has been adopted to measure the magnitude of the improvement/drop in each investigated empirical study. This permits a comparison of TDD's performance amongst the studies and allows the calculation of a summary value which indicates the overall magnitude of change TDD has shown in the studies thus making conclusions from the

review more meaningful.

3. Studies are divdided in two categories only, based on the experimental context (i.e., academic/industrial), thus avoiding the risk of having too few studies in a group which might lead to results that can't easily be generalized.

4. The meta-analysis is limited to studies that conform to the textbook definition of TDD with the exception of small differences, thus minimizing the risk that an observed effect is due to some other factor in addition to, or even instead of, TDD itself. Moreover, research works that report results already mentioned in another research work, that itself have been included/excluded, are excluded from the analysis.

5. An integral part of the meta-analysis is the investigation into moderator variables that govern TDD's performance. Although this aspect has been briefly discussed by Shull *et al.* [80], this research takes a deeper look into the analyzed emprical studies, identifies potential moderator variables and supports them with empirical evidence.

## 2.2 System Dynamics Modeling in the Agile Domain

This section presents a summary of studies that have modeled some aspect of agile software development using System Dynamics modeling. Research works in this area can be divided into two categories. The first category of works analyze the development process at a relatively low level and typically involve an investigation of the effects of individual agile practices such as Test-Driven Development, Pair Programming etc. In the second category of works, the development process is viewed at a relatively higher level and these works usually involve the modeling and analysis, in whole or in part, of agile methodologies such as XP, FDD, Scrum etc. which are comprisd of a set of agile practices.

Misic *et al.* [63] analyse two practices from the XP methodology namely Pair Programming

(PP) and pair switching, and frequent task switching. The authors aim to investigate the impact of variables that govern the performance of these two practices. With regard to the earlier practice, two variables are identified as having a moderating effect: psychological compatibility and pair adaption speed. With regard to the latter, the only moderator variable identified is task learning speed. Based on their simulations, the authors make the observations that for a successful XP implementation, that includes the two investigated practices, managers must pay more attention towards psychological compatibility of pairs and should avoid frequent pair and task switching unless the pair adaption and task learning speeds, respectively, are higher than the industry threshold.

Wernick & Hall [90] use simulation modeling to analyse the impact of PP on trends in software evolution. They develop a high level SD model of a 'base' software development process and then simulate it under different parameter settings. Each parameter setting corresponds to an advantage brought about by applying PP for example, in one setting the input effort is reduced citing that this benefit was reported in a previous empirical trial. The simulated advantages include reduction in development time, input effort, fault generation and an improvement in system maintainability/code design. Results from their simulation runs indicated that PP promotes system longevity and leads to requirements being delivered faster.

Cao *et al.* [14], rather than focusing on a specific agile methodology, present an SD model that is a integration of smaller submodels or subsystems(co-flows) which collectively model a set of agile practices. The modeled practices include agile planning, short iterations, customer involvement, refactoring, unit testing and pair programming. The submodels simulate the following sectors of agile software development: customer involvement, change management, agile planning and control, and refactoring and quality of design. The presented model was used for analyzing the impact of the refactoring practice on project performance and the economics of PP. The refactoring process was found to impact the cost-of-change which during the simulation process varied cyclically and increased over time. Moreover, it was observed that projects could be delayed due to major refactoring cycles. With regard to the results on the latter practice, PP was found to reduce

the cost of each task delivered, the cost of rework and the total cost of refactoring. Results pertaining to the refactoring practice in this study were used as the foundation for the design co-flow in my research.

Kuppuswami *et al.* [49] modeled a development process that employed twelve XP practices. The aim of the modeling process was to investigate the effect of individual XP practices on the cost of change curve. Based on the authors' measurement framework, they noted that each of the practices lowered the cost-of-change depending on the extent of their usage. The simulation results further depicted that the largest impact on the cost-of-change was made by the onsite-customer practice and the smallest impact was made by the refactoring practice.

Yong & Zhou [96] also modeled an XP-based development process albeit at a slight higher level than Kuppuswami *et al.* [49]. The authors have a modeling goal similar to that of [49]; to analyze the effect of individual XP practices on development time. As an application of the presented model, tested the impact of inclusion/exclusion of the TDD and PP practices on development time. Results from the simulation process depicted that both practices lower development time; a 20 % drop in finish time was noted when both practices were included.

Wu & Yan [93] modeled an XP-based development process at a level of detail similar to Yong & Zhou [96]. However, in [93] the impacts of all practices other than XP were collected modeled using a single model element. This research aimed to use the SD model to assess the extent of the impact of PP on development time. After simulating the inclusion/exclusion of the PP practice under different conditions of task size and workforce size, the author concluded that the adoption of PP does reduce development time however the magnitude of the improvement is small and the order of 3%.

The above studies indicate that research on SD modeling within the agile domain still remains largely unexplored. The majority of the available studies focus on the XP methodology at large or analyzing the Pair Programming practice. The impact of TDD has only been modeled at a very high level typically as a simple model element. To the best of our knowledge, this research is the

first to employ software process simulation modeling to perform an in-depth analysis of the impact of TDD on software development.

# Chapter 3

# Meta-Analysis on TDD

This section presents a meta-analysis of empirical research on TDD that has been published up to December 2010. The analysis has been conducted whilst adhering to the general guidelines for reviews in software engineering [46]. While every effort was made to include all studies, the review is limited to studies that compare the performance of TDD with a more conventional development process in an empirical setting as will be described below. Additionally, research reported in this paper is limited to aggregating empirical findings on two, arguably most important, outcome constructs: *software quality* and *productivity*.

## 3.1 Methodology

The Meta-Analysis approach has gained considerable attention in the last twenty-plus years as one of the effective ways to quantitatively summarize and, if possible, interpret the results of a collection of single studies on a given topic [38]. The analysis proceeds through a number of distinct steps, as follows.

### 3.1.1 Study Identification and Selection

The identification and selection process was divided into three stages.

First, we identified candidate studies by querying electronic databases for articles either published in peer-reviewed journals or included in proceedings of conferences. The databases that were searched include ACM Digital Library, IEEEXplore, SpringerLink, ISI Web of Science, and

Scopus. Each database was queried using the strings 'Test Driven Development,' 'Test First Development,' and 'TDD' with the search parameters set to look in the Article Title, Abstract and Keyword fields. The generated matches were filtered so that only those studies included in peer-reviewed journals or proceedings from conferences would be displayed, and subsequently selected for the next stage of processing.

The matches from the first stage were pre-screened for relevance, which was primarily done by reading through the titles and abstracts but in some cases also involved going through the introduction. All research studies that were found to be relevant and those whose relevance was still unclear were selected for a more detailed analysis in the next stage.

In the final stage, all of the selected studies were read and filtered based on the inclusion and exclusion criteria mentioned in the next section. Accordingly, a final list of studies was derived that would form the subject of the meta-analysis.

### 3.1.2 Inclusion and exclusion criteria

Studies were included in this meta-analysis if they reported results from one or more trials in which the effectiveness of TDD was compared with that of a more traditional (i.e., Test-Last) approach. Such trials took the design of subjects being divided into two or more groups, each of which developed the same or similar products with at least one group following either development approach. Studies were only included if they reported quantitative data on at least one of the investigated outcome constructs. The use of other agile practices along with TDD was not considered as a limiting factor although it is recognized as a threat to the validity of this analysis; this and other threats are discussed in more detail in Section 3.2.5. Also, the strict operationalization of the outcome constructs was not a mandatory requirement for inclusion, as it was desirable to incorporate as many relevant studies as possible.

During the selection process, a number of studies were deemed unsuitable for inclusion in our

study, due to the following.

- Some studies lack a quantitative component, relying instead on qualitative assessment or practitioner perceptions [62; 48; 73]; as a result, their findings are impossible to include in the meta-analysis.

- Some studies provide data obtained only with TDD, possibly over multiple releases of the same product, but without the control group [2; 76]. Therefore, discussing improvements obtained through TDD is not possible.

- Some studies focus on other outcome constructs, mostly design quality but also test coverage [42; 41; 55; 57; 81; 83; 84]. However, this approach suffers from two drawbacks. First, both measures are, at best, only indirect measures of quality. Second, there is no unified metric for either of them; in particular, design quality is notoriously difficult to assess. Therefore, we have decided against using these variables as outcome constructs in this analysis.

- In some cases, the development process did not follow TDD with sufficient rigor: e.g., in studies reported in [21; 22] subjects received feedback and were allowed to verify the completeness of their tests multiple times prior to submission; in [95], tests in the treatment group were authored before or in conjunction with code; finally, in [58], both TDD and traditional processes are applied to the same code interchangeably, which makes their impact difficult to ascertain.

- Finally, papers that simply repeated the results of another paper were not considered, as is the case with [13], [60], and [28], which contain similar or less extensive data as [12], [91], and [29], respectively.

It is important to add that the exclusion does not imply that these studies are without merit; it simply means that the approach taken in these studies did not align well with the goals of this analysis.

### 3.1.3 Data Extraction

The data extracted from the studies was classified into three categories: Context, Rigor, and Outputs. The earlier two categories were intended to provide a description of each study whereas the latter presents the different formats in which data was reported in the studies. The attributes in each category are described next.

Attributes in the Context category provide high-level information of the study and include:

- *trial*, name of the trial (set to the authors' name of the respective study. If the study reports results on multiple trials that are treated individually then authors' names are followed by T$ where $ indicates the trial number);
- *subjects*, number of subjects in the study;

Attributes in the Rigor category aimed to assist in identifying the extent of the universal applicability of a study's results, according to the criteria for study rigor described in [80]. These attributes include the following:

- *CI*, specifies whether the development process in the control group was iterative and incremental;
- *OA*, indicating whether other agile practices such as pair programming were included in either of the development processes;
- *experience*, development or programming experience of the subjects;
- *size*, task size of the final application;
- *duration*, duration of the project;
- *conformance*, information about process conformance (i.e., adherence to the widely accepted principles of TDD development);

- *training*, details of training the subjects may have received prior to the trial.

The last category, Outputs, as mentioned earlier, contained attributes for the common formats in which results were reported amongst the studies. Some studies reported their results in multiple formats and those included in this category are not the only ones in which results were reported. This category is comprised of two subcategories that were formulated while reading the papers during the selection process. In particular, studies were identified as being in two distinct categories with regard to how they reported their results; each subcategory of Output attributes corresponds to a category of studies and includes the output formats that were selected for subsequent effect size calculation, as described in the next subsection, in this category of studies.

Studies in one category reported their results using a notation that allowed tests of statistical significance to be run, i.e., they reported the mean and standard deviation of the results of each of the subject groups, and perhaps also the $p$-value. Evidently, these results facilitate further manipulation needed for the meta-analysis. Studies in the other category reported only either the percentage improvement, and/or a single value for an outcome construct for each group, which was then transformed, prior to conducting the meta-analysis, into a percentage improvement value. As such, the attributes in the Output category include:

- $\overline{\mu_i}(x)$, mean value of variable $x$, where $x$ denotes quality or productivity in study $i$, for each of the treatment (TDD) or control (traditional) groups;
- $\sigma_i(x)$, standard deviation of variable $x$ in study $i$ for each of the groups above;
- $v_i$, within-study variance in study $i$;
- $p - value_i$, denotes the probability that in study $i$ the difference in the results of the two groups is due to chance (i.e., assuming that the null hypothesis is true);
- $\%imprv_i$, percent improvement obtained by the treatment group over the control group in study $i$.

Data pertaining to the Context and Rigor categories is given in subsection 3.2.1 of the Results section in Tables 3.1 and 3.2. Data pertaining to the Output category is not mentioned in this handout to avoid redundancy as the calculated effect sizes for the individual studies are given in the Results section.

### 3.1.4 Statistical analysis

The effect size is defined as the magnitude of the difference in final result upon employing either development approach (i.e. TDD and traditional). The originally intended procedure was to compute a standardized effect size [30] for each of the analyzed trials and then to synthesize individual effect sizes into a summary effect size using one of the common meta-analytical statistical models. However, the larger share of studies, in particular among the industrial ones, had not provided enough data for standardized effect size calculation. Consequently, the analysis was split into two sub-analyses, dubbed Standardized and Unstandardized. For the former, standardized effect sizes were calculated wherever possible, and subsequently combined using the meta-analytical statistical models. For the latter, the unstandardized effect size was calculated for each analyzed trial, and the summary effect size was then calculated as the simple mean of the individual effect sizes.

#### 3.1.4.1 Standardized analysis

*Calculating the Effect Size.* All standardized effect sizes in this paper were computed using the Comprehensive Meta Analysis V2 tool by BioStat Inc [39]. The Hedges' $g$ statistic was chosen as the standardized effect size measure for the analysis, as it exhibits better characteristics for smaller samples when adjusted for small sample bias compared to other parametric measures such as Cohen's $d$ and Glass' $\Delta$ [32]. The Hedges' $g$ statistic is calculated as

$$g = \frac{m_t - m_c}{s_{pooled}} \tag{3.1}$$

where $m_t$ and $m_c$ refer to the mean values reported for the treatment and control groups, respectively, and $s_{pooled}$ refers to the pooled standard deviation:

$$s_{pooled} = \sqrt{\frac{(n_t - 1)s_t^2 + (n_c - 1)s_c^2}{(n_t - 1) + (n_c - 1)}} \tag{3.2}$$

Variables $n_t$ and $n_c$ denote the number of subjects in the treatment and control groups, respectively, and $n_{total}$ denotes their sum.

Small sample bias is accounted for by multiplication by a correction factor [32]:

$$cf = 1 - \frac{3}{4(n_{total} - 2) - 1} \tag{3.3}$$

According to Kampenes *et al.* [44], effect sizes with a magnitude in the range 0.0-0.37, 0.38-1.0, and 1.0 and above, can be considered as small, medium and large sized effects, respectively. Positive effect sizes represent an improvement as a result of applying the treatment whereas negative values imply a detrimental impact.

*Research Synthesis.* Individual study effect sizes were combined to obtain a summary effect size using two popular statistical models, namely the fixed-effects model and the random-effects model [11]. The principal difference between these two models is the underlying assumption. The fixed-effects model assumes that only one true effect size underlies all studies; hence differences in individual effect sizes are solely due to sampling error, i.e., as the sample size increases the effect size is likely to converge to the one true value. On the other hand, the random-effects model is premised on the assumption that differences amongst individual effect sizes are not just due to sampling error but might also be the result of other variables and factors that have not yet been taken into account. As a result, the effect size may vary from study to study, and is expected to be distributed about some mean effect size. The two models, thus, attempt to answer slightly different questions: while the fixed-effects model aims to derive the one true effect size, the random-effects

models aim to find the mean of the curve along which all possible effect sizes are distributed.

Both models follow a similar procedure to derive the summary effect size. First, each study $i$ of $k$ studies is assigned a weight $w_i$. Then the summary effect size $T_s$ is computed as the weighted average of the individual effect sizes $T_i$:

$$T_s = \frac{\sum_{i=1}^{k} w_i T_i}{\sum_{i=1}^{k} w_i} \tag{3.4}$$

A study's weight $w_i$ is calculated as the inverse of its error variance. Due to differing modeling assumptions, a study's error variance is calculated in different ways in the fixed- and random-effects models [11]:

$$\begin{aligned} w_{if} &= \frac{1}{v_i} \\ w_{ir} &= \frac{1}{v_i + \tau^2} \end{aligned} \tag{3.5}$$

In the fixed-effects model, the only source of variance is the sampling process, hence the weights $w_{if}$ are computed as the inverse of just the within-study variance $v_i$. In the random-effects model, other factors could also add on to the overall variance. Consequently, the weights $w_{ir}$ are computed as the inverse of the sum of the within-study variance $v_i$ and a constant $\tau^2$ representing the between-study variance, which is calculated using the DerSimonian and Laird formula [32]:

$$\tau^2 = \begin{cases} \frac{Q - df}{C}, & \text{if } Q > df \\ 0, & \text{if } Q \leq df \end{cases} \tag{3.6}$$

The degrees of freedom $df$ is equal to the number of studies being analyzed, minus one. The $Q$ statistic is the weighted sum of the squares of the deviations of each study's effect size from the summary effect size, with both the weights and the summary effect being computed following the

the fixed-effects model beforehand as shown below:

$$\begin{aligned} df &= k-1 \\ Q &= \sum_{i=1}^{k} w_{if}(T_i - T_{sf})^2 \end{aligned} \tag{3.7}$$

Finally, $C$ is a scaling factor used to ensure that $\tau^2$ is reported in the same metric as the within-study variance $v_i$ i.e,:

$$C = \sum_{i=1}^{k} w_{if} - \frac{\sum_{i=1}^{k} w_{if}^2}{\sum_{i=1}^{k} w_{if}} \tag{3.8}$$

The choice of which model to use is dependant on factors such as knowledge of the environment, previous empirical observations and findings, and the like; however, the modeling process itself also provides assistance. In particular, the $Q$ statistic from (3.7) is consistent with the level of heterogeneity, i.e., a significant value of $Q$ indicates a statistically significant level of heterogeneity [32]. Hence a significant value of $Q$ should lead to the rejection of the homogeneity hypothesis and the adoption of the random-effects model. In addition to $Q$, the $I^2$ statistic has also been computed as:

$$I^2 = \frac{Q - df}{Q} \times 100\% \tag{3.9}$$

This statistic is used to quantify the degree of true heterogeneity, i.e., the extent to which the total variance is the result of between-study variance [38]. $I^2$ values of 33.3 and 66.6% delimit low, moderate, and high levels of heterogeneity.

#### 3.1.4.2 Unstandardized analysis

*Calculating the Effect Size.* Percentage improvement was chosen as the unstandardized effect size measure. The formula used to calculate the effect size was

$$\Delta x = \left| \frac{x_t - x_c}{x_c} \right| * 100 \tag{3.10}$$

All of the analyzed studies had adopted one of two designs for their trials and the exact values for $x_t$ and $x_c$ were dependant on the design chosen. In the first type of design, each group of subjects jointly developed a system. When using this design, for each investigated outcome construct, the study authors had reported a single value for each subject group and/or went a step further and computed the percentage improvement. If only the percentage improvement was reported then it was accepted without change, but if the authors had reported actual group values, the percentage improvement was recalculated using (3.10) with the variables $x_t$ and $x_c$ set to the values reported for the treatment and control groups, respectively.

In the second type of design, all subjects in a group individually developed the same system. In this case, for each investigated outcome construct, the study authors had either reported individual values for each subject in the group and/or they had reported the mean and the standard deviation of the individual values for each group. In this design the means for the subject groups were used to set the variables $x_t$ and $x_c$.

*Research Synthesis.* As mentioned above, in the Unstandardized analysis the summary effect size was calculated as the average of the individual effect sizes.

## 3.2 Results

A total of 20 studies were selected for this review; Tables 3.1 and 3.2 lists data collected for the context and rigor related attributes for academic and industrial studies, respectively. A full citation

Table 3.1: Details of academic studies.

| trial | subjects | *CI* | *OA* | experience | size (LOC) | duration | conformance | training | other details |
|---|---|---|---|---|---|---|---|---|---|
| Desai *et al.* | 166 [a] | N | N | junior UG | | 10 hrs [b] | | Lab materials provided instructions on TDD, prepared by a person with limited TDD experience and no curriculum development experience. | Seven assignments dealing with writing shape classes like Triangle, Rectangle etc. Introductory Java course. |
| Erdogmus *et al.* | 24 | Y | N | intermediate UG | 272 [c] | 32 hrs [d] | Post-survey and tests used to judge conformance for each subject. | Students trained in Test-First and Test-Last techniques. Task: bowling score keeper. | Introductory Java course. |
| Flohr and Schneider | 18 | Y | Y | G | | 23 hrs [e] | Process conformance was poor. | A 2 hour lecture on TDD and testing. More training was needed. | Task: library for a system aimed to provide graphical description of communications flows in software processes. |
| Gupta and Jalote T1 | 22 | N | N | G, senior UG | 1600 | 31.5 hrs | In the post-survey, 70% of TDD group and 88% of Traditional group said they adhered to their approach. | Subjects given necessary training to use TDD for developing Java programs; however, in the post-survey 47% of TDD group stated they need more training. | Student course registration system. Advanced OO analysis and modeling course. |
| Gupta and Jalote T2 | 22 | N | N | G, senior UG | 1600 | 34.5 hrs | see above | see above | Simple ATM system, same course as above. |
| Huang and Holcombe | 39 | Y | Y | intermediate G | 1175 | 110 hrs | Project Manager was appointed to ensure conformance. | 10 hours of advanced training of TDD and other XP practices. | Different projects resembling realistic tasks. Software engineering course. |
| Janzen and Saiedian | 10 | N | N | intermediate, senior UG | 650 [f] | 1.5 hrs | | Students were instructed on how to write automated tests and how to develop in Test-First and Test-Last manner. | HTML pretty print system. Software engineering course. |
| Madeyski | 56 | Y | N | intermediate, senior UG | | 12 hrs | - | The course introduced Java using TDD and pair programming as the key XP practices. | Finance accounting system. Introductory Java course. |
| Muller and Hagner | 19 | N | N | G | | | | The course in which experiment was conducted covered pair programming, TDD, refactoring and planning techniques. | Main class of a graph library. Second course on XP. |
| Pancur *et al.* | 38 | Y | Y | senior UG | | | | | 1 long and 3 short assignments, 1 half-day assignment taken over a period of roughly three weeks. |
| Vu *et al.* | 14 | N | N | G, senior UG | 3070 [f] | 147.7 hrs [f] | | Test-First and Test-Last methodologies introduced through lectures and student presentations. | Software engineering course. |
| Xu and Li | 8 | N | N | intermediate, senior UG | 344 | 4 hrs | One of the authors acted as a mentor who monitored the programming process for all the individuals. | Short training sessions in which students were given reading materials and asked to implement a simple program to get used to of procedure and tools. | Task: bowling score keeper. Software engineering course. |
| Yenduri and Perkins | 18 | N | N | senior UG | | | | Students were given required user manuals and a short description of forming test cases. They were also trained in applying both TDD and traditional approaches. | Software engineering course. |
| Zhang *et al.* | 8 | N | N | G, UG [g] | | June to Aug. 2005 | | | Working attendance management system. |

[a] assuming the 2007 class and 2008 class had the same number of subjects

[b] average duration per project determined by combining hours spent by both UT and GT groups

[c] estimated using average of George & Williams study and Xu & Li's study since all three worked on implementing same problem

[d] 4 hrs per week in a 8 week course

[e] average based on individual subject attempts

[f] average of TF and TL group

[g] subjects refered to as students without highlighting seniority

Table 3.2: Details of industrial studies.

| trial | subjects | *CI* | *OA* | experience (years) | size (LOC) | duration | training | other details |
|---|---|---|---|---|---|---|---|---|
| Bhat and Nagappan T1 | 8 | N | N | 6 to 10 [a] | 6000 | 3840 hrs [b] | | Networking common library. |
| Bhat and Nagappan T2 | 20 | N | N | 6 to 10 [a] | 26000 | 7360 hrs [b] | | Web Service application. |
| Canfora *et al.* | 28 | N | N | 5 | | 10 hrs | 3 hr training session included a seminar on TDD and lab exercises to increase familiarity with the practice. | TextAnalyzer system. |
| George and Williams | 24 | N | Y | | 200 | 5 hrs [c] | | task: bowling score keeper. Groups in three companies. |
| Lui and Chan | | N | N | - [d] | | | | Custom software for manufacturing plants in less developed areas in China. |
| Nagappan *et al.* [e] | 12 | N | N | 10+ [a] | 155200 | 3200 hrs [b] | | Part of the development of an IDE tool. |
| Slyngstad *et al.* | 100 | N | N | | 14671 [f] | | | Five releases of an internally reusable framework. Developers distributed worldwide, mostly in Norway and Sweden. |
| Williams *et al.* | 14 | N | N | | 64000 | | Dedicated TDD coach was assigned as technical leader. | Subjects developed a release of a device driver. |

[a] majority of the developers had this experience level
[b] assuming they worked 160 hrs per month
[c] approximated from box plots and average of TDD and control groups
[d] study states that developers were inexperienced
[e] Only of the four trials mentioned in the paper is included
[f] average size of a release

of these studies is given in Appendix A.

Labels UG and G in Table 3.1 denote undergraduate and graduate students, respectively. Four of the studies [18; 31; 8; 71] reported results on multiple trials. In two of these studies [31; 8] each trial was eventually treated as a seperate study when computing the summary effect size since adequate details were provided regarding each trial and the development task per trial was comparable in size to that of other studies that consisted solely of one trial. For the remaining two studies, the study effect size was the calculated as the average of the effect sizes recorded for each of the trials.

### 3.2.1 Assessment of Study Rigor

In this section the major differences in rigor level amongst the studies are highlighted.

Firstly, studies differed in their definitions of both the treatment and control groups' processes. With regard to the control group's development process, the adopted development style varied be-

tween a true Waterfall and an ITL-based process. However, due to sometimes missing or different details, not all of the traditional processes explained in the studies can be classified to fall into one of these two categories. With regard to the treatment group's process, studies differed in what they assumed to be practicing TDD. Firstly, the granularity level of the development cycles was found to differ amongst the studies. Also, the frequency with which tests were run differed with some studies having reported that all tests were regularly run whereas others reported that only the newly-added test(s) were run after a change. Although the larger share of the studies examined the behaviour of TDD in isolation, as mentioned earlier, some of the studies assessed the performance of TDD in combination with other agile practices.

Secondly, there was considerable variation in the experience level of the subjects amongst the studies. In studies conducted in an academic context, subjects ranged from junior undegraduate students to graduate students; in industrial studies, experience level ranged from 1 or 2 years, to over 10 years of industrial programming experience. Intuitively, as mentioned earlier, in order to be able to accurately compare TDD with a more conventional process subjects should possess the technical skill set required to proficiently employ either process. Unfortunately, most of the analyzed studies only discussed subjects' programming experience and very few referred to their testing, designing or any other software engineering-related experience. Hence the degree to which the subjects met the skill requirements, especially in the case of the TDD, is uncertain.

Thirdly, studies differed in the size of the development task and/or the duration of the trial. Data on task size was available for thirteen trials, with the value having ranged from 272 LOC to 155,200 LOC. Data on duration was available for fifteen trials and varied within the range from 1.5 hours to 46 months. The task size was found to be larger than 2000 LOC in only six of the trials. Similarly, the duration was found to be longer than 100 hours in only five of the trials. Both of these findings indicate that a system comparable to one developed as part of a commercial project was only developed in a quarter of the trials. Trials where the development task is relatively small are less favourable when assessing the usefulness of TDD since benefits might not be visible [34].

Fourthly, studies differed in the extent of the TDD training the treatment group's subjects were given prior to a trial. In most of the studies subjects had little or no prior exposure to the TDD practice; hence their ability to apply TDD would have been highly dependent on the training they had received. The extent of the training reported in the studies ranged from the distribution of short supplementary handouts to dedicated lecture sessions on TDD. In a few studies the initial training was further complemented with a dedicated coach who mentored and/or overlooked the entire development process. As mentioned earlier, the level of training is a determinant of the difficulty developers face when applying TDD. Multiple studies that coupled trials with a practitioners' post-perception survey reported that a significant share of the subjects had faced difficulty in applying the technique. There are at least two plausible explanations for this. First, as mentioned earlier, the shift in mindset could have been difficult to grasp and, in view of the relatively short durations of some trials, it could have been that the TDD group's subjects were not able to adapt to the new mindset for a large share of the trial's duration or possibly the entire duration. Second, it is plausible that domain and tool-specific issues made it more difficult to practice TDD [16]. In particular, it could have been that tool support was limited or a significant share of the development task included user interface/database development. All these scenarios pose challenges that make it more difficult to implement TDD. As a result of insufficient training, it is possible that subjects in these trials could not correctly implement TDD and consequently the experienced benefits were lower than would otherwise have been the case.

Fifthly, studies were found to differ in the indicated amount of test effort or time spent testing by subjects in the TDD group. Whilst following TDD a larger test effort, in comparison to traditional approaches, is expected as the writing of tests is viewed as part of the requirements analysis and as such developers are intended to spend more time pondering over the requirements when writing tests. More time spent thinking about the requirements during test development is expected to materialize in developers having an improved conceptual understanding of the system hence reducing their likelihood of introducing defects. Although very few of the analyzed stud-

Table 3.3: Quality: results of standardized analysis.

| type | # | model | effect size | | | heterogeneity | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | $g$ | 95% confidence interval | $p$ | $Q$ | $I^2$ | df | $p$ |
| academic | 8 | fixed | 0.296 | 0.084 . . 0.508 | 0.006 | 47.712 | 85.329 | 7 | 0.000 |
| | | random | 0.081 | -0.538 . . 0.700 | 0.798 | | | | |
| industrial | 1 | fixed | 0.311 | 0.033 . . 0.588 | 0.028 | 0.000 | 0.000 | 0 | 1.000 |
| | | random | 0.311 | 0.033 . . 0.588 | 0.028 | | | | |
| OVERALL | 9 | fixed | 0.301 | 0.133 . . 0.470 | 0.000 | 47.718 | 83.235 | 8 | 0.000 |
| | | random | 0.114 | -0.357 . . 0.585 | 0.635 | | | | |

ies provide quantitative data on the time spent testing, as opposed to the number of tests written, based on description given for the TDD process it is hypothesized that the time spent testing did differ across trials and hence could have provided differing levels of improvements in quality and productivity.

Lastly, although subjects in a group were instructed to utilize a particular development process, few studies provided explicit details on actual conformance levels or on any steps that were taken to ensure a higher degree of conformance. High conformance levels are important since if developers even temporarily fall back to traditional approach (because of to time constraints, ease of technique, or other factors), the validity of the trial is reduced.

### 3.2.2 Effect of TDD on Quality

#### 3.2.2.1 Standardized analysis

*Overall Result.* Standardized effect sizes were calculated for nine trials utilizing a total of 434 subjects. A summary of the meta-analysis is shown in Table 3.3. The summary effect size was 0.30 and 0.11 under the fixed-effects and random-effects models respectively. Heterogeneity, as shown by $Q$ and $I^2$, was significant at a high level. Using both models a small overall improvement in quality was found. Assuming a .05 significance level, results were statistically significant under the fixed-effects model but not under the random-effects model.

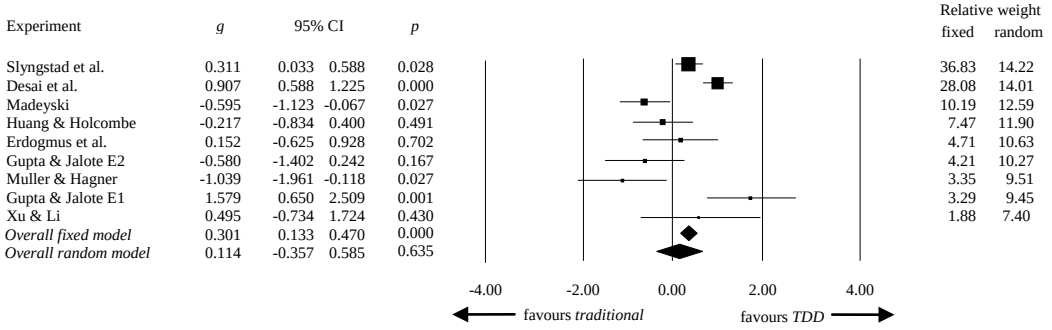



Figure 3.1: Forest plot of trials according to results on quality.

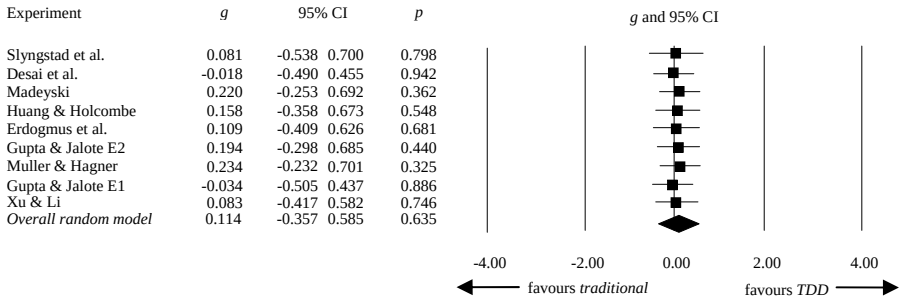



Figure 3.2: One-study removed analysis of trials according to results on quality.

*Forest Plot & One-Study Removed Plot.* The forest plot and the one-study removed plot of the analysis are shown in Figs. 3.1 and 3.2, respectively.

Forest plots provide a display of trials' effect sizes in comparison with each other and the overall result using each meta analytical model. Additionally, they provide an indication of the relative weight of each study in the analysis. In this plot a square is used to depict the effect size of a study and the lines stemming from it specify the 95% confidence interval. The square size is determined by the relative weight of the study. Relative weights are determined by normalizing the weights used in calculating the summary effect sizes under each model; these are are displayed on the right of the plot. Studies are ordered based on their weight in the random-efffects model. The overall result using a meta analytical model is shown by a diamond whose centre indicates the

estimate and whose horizontal edges denote the 95% confidence interval.

From the forest plots, it can be seen that in five of the nine trials a positive effect size value was recorded. Large improvements in quality were reported in the 1st trial from [31] and the trial in [18]. The only report of a large drop in quality is [65]. Under the assumption of a fixed-effects model, the three trials [85; 18; 54] accounted for more than half of the weight of the analysis. Study weights become more uniform in the random-effects model when the between-study variance is added.

A One-study removed plot shows the sensitivity of the overall result to each of the included trials. Plots of this type shown in this thesis are all based on the random-effects model; plots employing the fixed-effects model have been ignored since the heterogeneity level was found to be high. In these plots the *g* value next to a trial shows the overall effect size if the respective trial was excluded.

Under the assumption of a random effects model, the one-study removed plot reveals that the result of the standardized analysis on quality is most sensitive to the inclusion of two of the trials that carry the most relative weight namely [85; 18].

#### 3.2.2.2 Unstandardized analysis

For the unstandardized analysis effect sizes were computed for 18 trials which employed a total of 571 subjects. Fig. 3.3 shows a histogram that illustrates the distribution of the percentage improvement. The mean value for the percentage improvement in quality was 28%. In comparison with the standardized analysis, the effect size illustrates a more prominent impact; however, due to the effect size measure being unstandardized there is a possibility that the effect is exaggerated.

The individual effect sizes for the trials are shown in Table 3.4. 12 out of 18 trials reported a quality improvement higher than 10%, with [66] reporting the highest result of 90%; 5 trials reported an impact within the range -10% to 10%, and only [54]—an academic study in which TDD was applied in an introductory Java course—reported a 27% drop in quality.

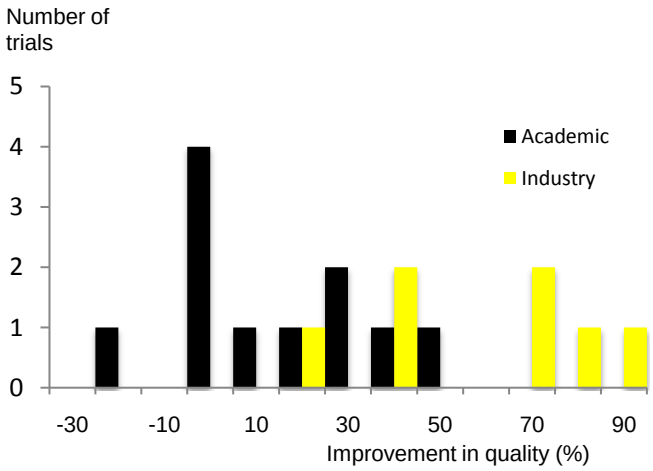



Figure 3.3: Distribution of improvements in quality.

Table 3.4: Quality: results of unstandardized analysis.

| trial | improvement (%) |
|---|---|
| Madeyski | -27 |
| Gupta and Jalote T2 | -7 |
| Huang and Holcombe | -3 |
| Pancur *et al.* | -3 |
| Erdogmus *et al.* | -2 |
| Desai *et al.* | 3 |
| Gupta and Jalote T1 | 16 |
| George and Williams | 18 |
| Vu *et al.* | 28 |
| Zhang *et al.* | 28 |
| Slyngstad *et al.* | 33 |
| Yenduri and Perkins | 35 |
| Williams *et al.* | 39 |
| Xu and Li | 49 |
| Bhat and Nagappan T1 | 62 |
| Lui and Chan | 67 |
| Bhat and Nagappan T2 | 76 |
| Nagappan *et al.* | 90 |
| OVERALL | 28 |

#### 3.2.2.3 Subgroup Analyses

Only one of the studies conducted in industry reported data adequate for standardized effect size calculation. Accordingly, the standardized analysis can only be trusted with respect to academic studies. In this subgroup the application of TDD resulted in a small overall positive impact on quality similar to the overall result.

In comparison with the standardized analysis, the unstandardized analysis which combined

Table 3.5: Productivity: results of standardized analysis.

| type | # | model | effect size | | | heterogeneity | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | $g$ | 95% confidence interval | $p$ | $Q$ | $I^2$ | df | $p$ |
| academic | 7 | fixed | 0.316 | -0.015 . . 0.647 | 0.061 | 29.955 | 79.970 | 6 | 0.000 |
| | | random | 0.253 | -0.515 . . 1.021 | 0.518 | | | | |
| industrial | 1 | fixed | -1.111 | -1.887 . . -0.335 | 0.005 | 0.000 | 0.000 | 0 | 1.000 |
| | | random | -1.111 | -1.887 . . -0.335 | 0.005 | | | | |
| OVERALL | 8 | fixed | 0.096 | -0.208 . . 0.401 | 0.536 | 40.949 | 82.906 | 7 | 0.000 |
| | | random | 0.070 | -0.688 . . 0.829 | 0.856 | | | | |

results from 11 academic and 7 industrial studies, provided a more elaborate view of TDD's impact on quality in industry. The mean percentage improvement dropped to 11% for the academic subgroup but rose to 55% for the industrial subgroup. Although results from trials conducted in academia were mixed, all trials conducted in industry reported a positive impact on quality, with [29] reporting the smallest improvement of 18%.

### 3.2.3 Effect on Productivity

#### 3.2.3.1 Standardized analysis

*Overall Result.* For the standardized analysis on productivity, effect sizes were computed for eight trials which employed a total of 153 subjects. The summary of the meta-analysis is given in Table 3.5. Summary effect sizes of 0.10 and 0.07 were determined under the fixed-effects and random-effects models, respectively, both of which indicate a very small to negligible impact on productivity. Under the fixed-effects model, heterogeneity was significant at the high level. $p$-values indicate that both results are statistically insignificant at the .05 level.

*Forest Plot & One-Study Removed Plot.* The forest plot and the one-study removed plot for the analysis are shown in Figs. 3.4 and 3.5, respectively. Individual effect sizes were almost equally divided with four effect size values pointing in either direction. The range of effect sizes was larger and approximately double the range of corresponding effect sizes computed for the quality

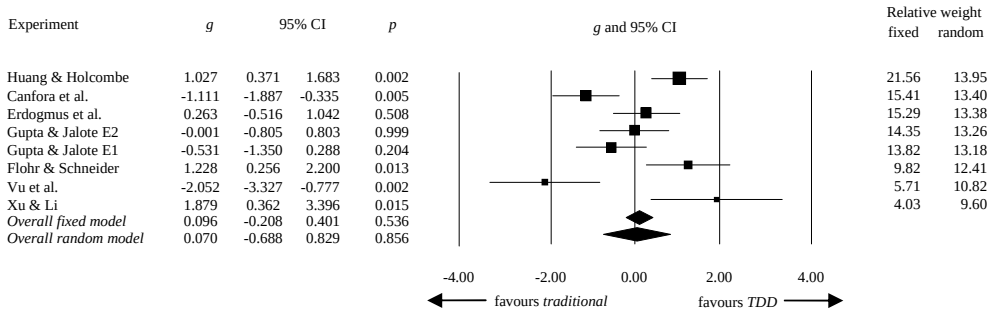



Figure 3.4: Forest plot of trials according to results on productivity.

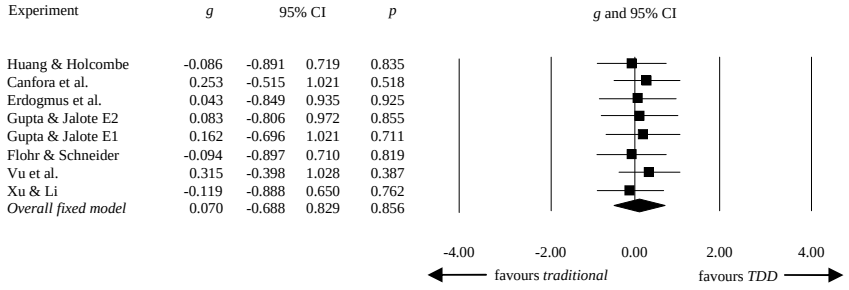



Figure 3.5: One-study removed analysis of trials according to results on productivity.

construct. Interestingly enough, the effect sizes for five of the eight studies were in sharp contrast with the summary effect size. In particular, three trials [37; 24; 94] reported large improvements in productivity, whereas two trials [12; 89] reported large drops in productivity. Unlike the analysis on quality, relative weights computed for the trials were more evenly distributed. Moreover, the one-study removed plot illustrates that differences in sensitivity level amongst the studies is also less.

#### 3.2.3.2 Unstandardized analysis

The unstandardized analysis on productivity combined data from sixteen trials which utilized a total of 267 subjects. The distribution of the percentage improvement is shown in the histogram in Fig. 3.6. The mean value for the percentage improvement was -3%.

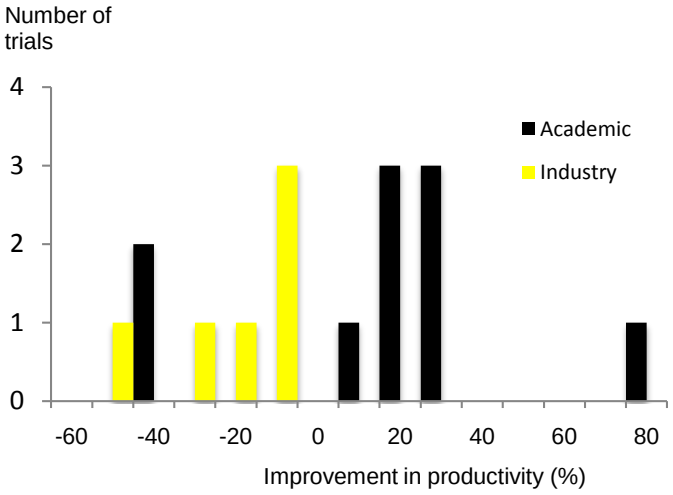



Figure 3.6: Distribution of improvements in productivity.

The unstandardized effect size for each individual trial is given in Table 3.6. Similar to the standardized analysis, the trial effect size values strongly diverge from the overall result. Suprisingly, no trial reported a percentage improvement in the range -10% to 10%. Rather, approximately half of the trials reported an improvement greater than 10% with the other half reporting a drop greater than 10%. The largest improvement in productivity, 72%, was reported by [37]; the largest drop of 57% was reported by [12].

Table 3.6: Productivity: results of unstandardized analysis.

| trial | improvement (%) |
|---|---|
| Canfora *et al.* | -57 |
| Xu and Li | -49 |
| Janzen and Saiedian | -46 |
| Bhat and Nagappan T1 | -30 |
| Nagappan *et al.* | -23 |
| Williams et al. | -18 |
| George and Williams | -16 |
| Bhat and Nagappan T2 | -15 |
| Zhang *et al.* | 10 |
| Gupta and Jalote T2 | 14 |
| Vu *et al.* | 19 |
| Gupta and Jalote T1 | 20 |
| Flohr and Schneider | 21 |
| Yenduri and Perkins | 25 |
| Erdogmus *et al.* | 28 |
| Huang and Holcombe | 72 |
| OVERALL | -3 |

#### 3.2.3.3 Subgroup Analyses

Analogous to the standardized analysis on quality, the standardized analysis on productivity only utilized data from one industrial trial. Hence, the results can only be trusted to illustrate the productivity-related performance of TDD in the academic sector. Interestingly, upon division the summary effect size rose to 0.32 for the academic subgroup.

The unstandardized analysis on productivity aggregated data from 10 academic and 5 industrial trials. Upon the division into subgroups a result opposite to that of the unstandardized analysis on the quality construct was observed: the mean percentage improvement in productivity was 11% for trials conducted in academia and -27% for those conducted in industry. The majority of the trials in the academic subgroup reported a positive impact on productivity. In contrast, all of the trials conducted in industry reported a detrimental impact on productivity, with the lowest drop being 15% which was reported in the 2nd trial of [8].

### 3.2.4 Moderator Variables

The histograms of the previous sections depict a large variation in the effect size of both of the investigated outcome constructs. Although this variation can partially be attributed to the experience level of the subjects, as shown by the subgroup analyses, even when comparing effect sizes within each subgroup there still exists a visible degree of variation in the effect size values. In order to reduce uncertainty and gain a deeper understanding of the impact of TDD on the development process, it is necessary to attempt to rationalize this variation or fluctuation in results. One approach towards this objective is to identify moderator variables that would influence the extent of the improvement experienced in a particular construct. In this research, based on the discussion of rigor in Section 3.2.1, some variables are identified as having moderating effects, and for the remainder of this section available evidence related to these variables is analyzed. In particular, empirical literature including studies part of the meta-analysis as well as other research up to De-

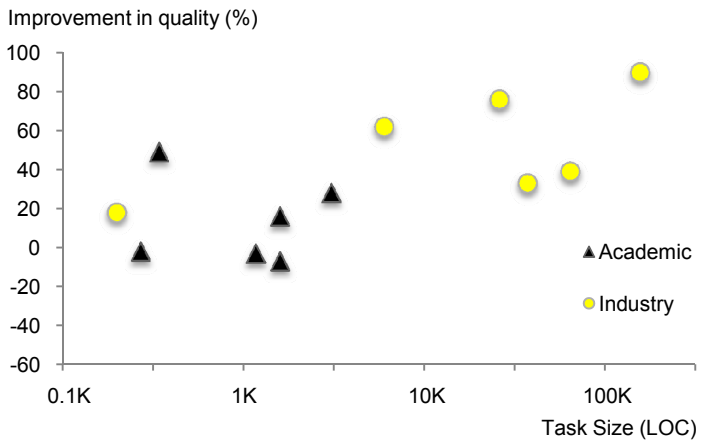
cember 2010 is examined for details of the effects of each moderator variable on the performance of TDD.

#### 3.2.4.1 Task size

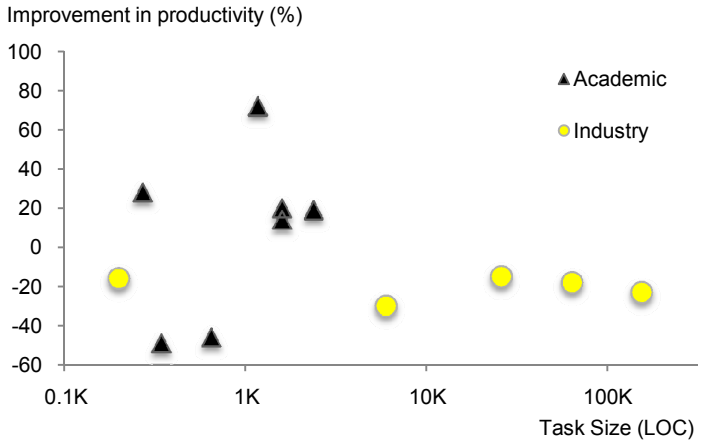
It is important to consider the task size when anticipating the extent of the improvement TDD will provide. During our search for empirical studies on TDD, no study was found to explicitly analyze the impact of task size, however two of them hinted on the possible influence of this variable. When recalling experiences on the adoption of agile practices in industry, Hodgetts [34] claimed that the implementation of TDD took several two-week XP cycles to show significant results. Also, in a study conducted in an academic environment, Erdogmus *et al.* [23] explained a lack of improvement in quality by reasoning that when the programming task is small, adhoc testing strategies, visually inspecting code, and other convential practices could perhaps substitute the benefits brought about by TDD, thus making the advantages of TDD more or less transparent.

In the conducted meta-analysis, task size was reported for twelve trials in the analysis of quality, and for eleven trials in the analysis on productivity. Fig. 3.7 shows scatterplots of task size versus the percentage improvement measures for both outcome constructs; for clarity, the $x$-axis uses a logarithmic scale.

For the quality construct, the plot indicates a visible trend, but this trend is less prominent for the productivity construct. To investigate whether or not task size is correlated with either of the outcome constructs, Spearman correlation coefficients were computed for each construct using the percentage improvement and the log transform of task size; the values thus obtained and their significance levels are given in Table 3.7. As can be seen, task size was found to be correlated with quality but not productivity.




(a) Improvement in quality.




(b) Improvement in productivity.

Figure 3.7: Scatterplot showing the relationship of task size with improvements in quality and productivity.

Table 3.7: Correlation of task size with quality and productivity.

| | correlation coefficient | *N* | *p*-value |
|---|---|---|---|
| quality | 0.63 | 12 | 0.029 |
| productivity | -0.05 | 11 | 0.873 |

#### 3.2.4.2 Developer experience

Although the success of TDD is dependent on a number of skills, including those in programming, testing, design, refactoring and thinking in a TDD style [4; 5], programming experience, extent

Table 3.8: Empirical studies investigating the influence of developer experience

| study | subjects/pairs type | # | context | experience (years or otherwise) general | Java | testing | TDD | code size (LOC) | duration (hours) |
|---|---|---|---|---|---|---|---|---|---|
| Müller and Höffer | Novice | 11 | XP course 2004 | 6 | 2.4 | 4 used JUnit | 1 tried TDD | 450 | |
| (individual) | Expert | 7 | Multiple Sources | 7.0 | 6.2 | 4.3 (JUnit) | 3.4 | | 4-5 hrs |
| Höfer and Philipp | Novice | 12 | XP course 2006 | 4.8 | 2 | 1 used JUnit | - | | |
| (pairs) | Novice | 12 | XP course 2008 | 6.4 | 4 | 3 had used JUnit | 3 tried TDD | 450 | 4-5 hrs |
| | Expert | 12 | company (see text) | 7.8 | 7.2 | 5.5 (JUnit) | 3.0 | | |
| Madeyski | 2nd year | 39 | | | | | | 27 | |
| (pairs) | 3rd year | 27 | Programming in Java Course | | C and C++ | - | - | user stories | 12 hrs |
| | 4th year | 4 | | | | | | | |
| | MSc | 70 | | | | | | | |

of exposure to TDD and possibly testing experience are the only types that have been empirically investigated. In our search three studies were found explicitly examining the impact of experience level namely [64; 35; 56]. Details of studies are given in Table 3.8. This section presents a brief review of the findings of these studies.

Muller & Hoffer [64] compared the performance of final-year undergraduate students from an XP course with that of professionals, selected from different sources, with over five years of industrial programming experience; the latter were also more experienced with regard to use of an automated testing tool and exposure to TDD. All subjects individually developed a Java-based elevator control system following the TDD approach until they felt they were done; their programs were then evaluated using a number of prepared acceptance tests. With regard to the productivity construct, subjects in the professional group generally had a lower implementation time. This result was statistically significant and was attributed, by the authors, to a faster coding speed and higher programming experience level. In contrast, with regard to the quality construct, a larger proportion of the programs prepared by the students passed the acceptance tests; this result was also statistically significant. The authors attempting to rationalize this finding hypothesized that professionals could have viewed the acceptance testing process as a regular adjunct to testing, and hence taken it more lightly in comparison to students who viewed the acceptance tests as a more formal assessment criteria, and thus ensured, to a greater degree, that the implemented functionality was in working order prior to submission.

Hofer & Philipp [35] repeated the trial of Muller & Hoffer [64] using the same experimental task but this time having two distinct student/novice groups. Additionally, subjects in this trial worked in pairs. Subjects in the novice groups were students that took the same XP course as Muller & Hoffer [64] in later years, with one group being selected from the 2006 class and the other from the 2008 class. Students in the novice groups slightly differed in programming experience, but they had similar expertise in TDD and the JUnit tool. Most of the subjects in the expert group were employees of a company specializing in agile software development and consulting. Interestingly, results from [35]'s trial significantly differed from that of [64]. With regard to the productivity construct, subjects in the expert group generally took longer to complete the program than both of the two novice groups. The difference between the professional and the novice 2008 group was statistically significant but those between the professional and the novice 2006 group and between the two novice groups were not. The extra time utilized by the professional group was reasoned to have been spent refactoring code, a finding that was also statistically significant. Results for quality were not reported.

Madeyski [56] analyzed the impact of experience on quality using data from a previously conducted trial explained which is explained in [54]. A much larger trial in terms of the number of subjects, this trial differed from the previous two trials in that it was geared towards comparing the performance of pairs applying the classic testing approach with those applying the TDD approach. As such, not as much attention was given towards reporting the different types of skills subjects possessed in comparison with the above two studies. In this study subjects' experience level was determined by the academic year in which they were enrolled. The experimental task was a finance-accounting system which subjects developed over eight 90-minute laboratory sessions. Focusing on the performance of the TDD group, results indicated a small correlation between programmer experience and quality; however this observation was not statistically significant.

With regard to the studies included in the meta-analysis, although they reported data on subjects' experience level, it was found to be difficult to compare subjects across studies. Conse-

quently, a future work aims at developing a more general framework for recording the experience level and using it to analyze data related to experience level within the included studies.

#### 3.2.4.3 Test Granularity

Test Granularity, as it pertains to this research, is defined as the number of story points covered by each test eg. with a granularity of 1, a single test approximately covers each story point whereas with a granularity of 0.5, two tests would be required to cover the same sized story point. The notion of granularity has been formulated to investigate the impact of the number of tests written. In other words, it is assumed that a difference exists between the number of tests written by subjects in the control group and those written by subjects in the treatment group because subjects in each group write tests at differing levels of granularity.

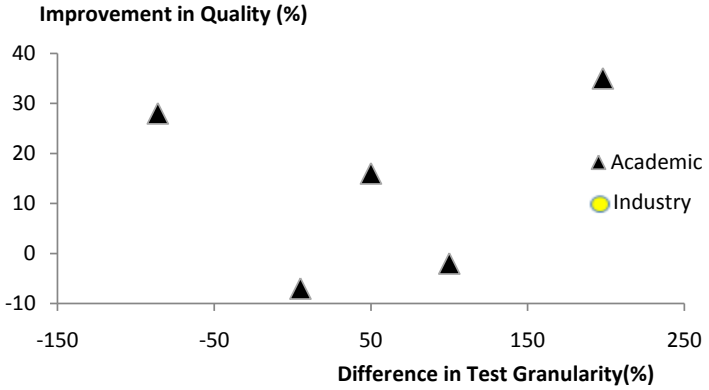
During the search for empirical studies on TDD, no study was found to explicitly examine the effect of test granularity. However, this variable can be hypothesized to have an impact on TDD's performance based on claims made in previous research. In particular, Erdogmus *et al.* [23], when attempting to rationalize the improvement in productivity experienced upon the application of TDD, state that by proceeding a single test at a time in a TDD-manner the development task is divided into smaller and more manageable pieces, and as such at any point during development the scope is narrower in comparison with a traditional development style. This leads to a lower cognitive load being placed on the developer. On the basis of this claim, it can be hypothesized that even when applying TDD a higher granularity level is preferable as it results in a narrower development scope and thus imposes a lower cognitive load.

Some of the empirical studies did record the number of tests that were written when either approach was applied. In this section data on the tests written will be extracted from the studies included in the meta-analysis and correlated with the improvements in quality and productivity.

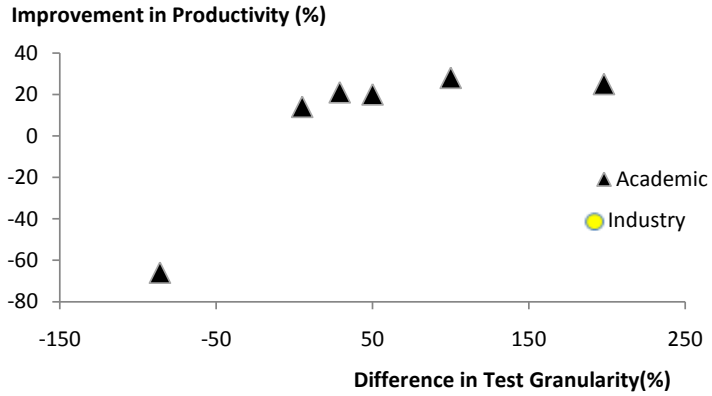
Data on tests written was recorded in six of the trials included in the meta-analysis, all of which were conducted in an academic environment. Differences in the number of tests written

Table 3.9: Percentage difference in number of tests written

| author | Difference in tests written (%) | Improvement in Quality (%) | Improvement in Productivity (%) |
|---|---|---|---|
| Vu *et al.* | -86 | 28 | -66 |
| Gupta and Jalote T2 | 5 | -7 | 14 |
| Flohr and Schneider | 29 | n/a | 21 |
| Gupta and Jalote T1 | 50 | 16 | 20 |
| Erdogmus *et al.* | 100 | -2 | 28 |
| Yenduri and Perkins | 198 | 35 | 25 |




(a) Improvement in quality.




(b) Improvement in productivity.

Figure 3.8: Scatterplot showing the relationship of the difference in test granularity with improvements in quality and productivity.

(or test granularity) for these trials are expressed as a percentage in Table 3.9. Scatterplots of the difference in test granularity versus each of the outcome constructs is given in Fig. 3.8. For both constructs there appears to be a linearly positive relationship with test granularity i.e. as the difference between the number of tests written increases, the improvements in quality and productivity also increase.

To determine the strength of the correlations of the constructs with this potential moderator variable, a Spearman correlation test was conducted using the data in Table 3.9. The resulting Spearman correlation coefficients and their significance levels are given in Table 3.10. The coefficients reveal that there is a weak correlation between differences in test granularity and quality however the moderator variable and productivity are strongly correlated.

Although none of the industrial trials included in this analysis provide a comparison between the number of tests written, the influence of test granularity can still be evaluated to some extent. In particular, Nagappan *et al.* [66] reported the results of four trials in which the ratio $TestKLOC/SourceKLOC$, operationalized here as the test granularity, differed from 0.39 to 0.89. Interestingly, the this ratio and the increase in development time, as a result of adopting TDD, was found to be somewhat correlated with the highest value for the ratio resulting in the lowest increase in development time. Similar to the trials conducted in academia, no visible correlation was found between the test granularity and the improvement in quality.

Table 3.10: Correlation of difference in test granularity with quality and productivity.

| | correlation coefficient | *N* | *p*-value |
|---|---|---|---|
| quality | 0.14 | 5 | 0.731 |
| productivity | 0.76 | 6 | 0.021 |

#### 3.2.4.4 Other Variables

There are two remaining moderator variables that deserve to be discussed, however, very little quantitative evidence of their impact on TDD's performance is available.

The time spent testing or test effort, as mentioned on the section of rigor, is an important concern when applying TDD since developers are supposed to use the tests to drive the analysis of the requirements. Ideally, the writing of tests, prior to the writing of code, is intended to trigger a prolonged thought process where developers spend more time thinking about the different scenarios the functionality-to-be-implemented will be required to undergo. This, in effect, is expected to result in developers having an improved understanding of the system.

During the search no empirical study was found explicitly examining the effect of differing levels of test effort. In the trials included in this meta-analysis the notion of test effort has been used interchangeably with the number of tests written and the time spent testing. In this research these concepts are treated seperately using two variables with the variable test granularity refering to the number of number of tests written and the variable test effort referring to the average time spent thinking about each story point when writing its associated test(s).

From the studies included in the meta-analysis only Huang & Holcombe [37] provided on the time spent testing; they found that the TDD group spent approximately 85% more time testing. However, even in this study it is unclear what portion of the extra time was spent contemplating analysis issues.

The last moderator variable that was analyzed as part of this research is domain and tool specific issues (DTSI). Despite widespread industrial initiatives towards TDD within the past few years, domain and tool-specific issues are still hindering TDD's adoption [16]. In particular, there are two key issues facing the adoptiong of TDD. Firstly, tool support for TDD-style development is only sufficient when development is being done using one of the popular languages; in others the capabilities of the development tools is still fairly limited. Lack of adequate tool support is expected to make it more difficult to apply TDD, thus resulting in a productivity hit, and also holds the possibility of lower conformance levels, which in turn can have a detrimental impact on both quality and productivity. Secondly, TDD-style development of GUIs and databases is more difficult. With regard to GUI-development, practitioners argue that following the TDD approach

is impractical. They claim that automated GUI tests are inflexible and error-prone to change, and considering the high volatilty of GUIs during development, the TDD approach is not suited for GUI development [5; 33]. Although approaches to develop GUIs in a TDD manner have been suggested [75; 33], the development of GUIs is still viewed as significantly more challenging in comparison with solely application-level code writing.

Similar to user interface development, database development faces its own challenges, but these challenges are not as hard to overcome. While techniques to develop database business logic and schemas in a TDD fashion have been formulated [3; 68], they do not address three key issues. Firstly, staging the database to an exact state for each test can be time-consuming and error-prone. Secondly database-related tests are more resource-intensive to run so they can't be expected to be executed at the same frequency as the code-related tests. Thirdly, creating a mockup of the database for running tests might result in simplifying some of its intricacies that are required for providing an accurate reflection of actual practice. In view of the issues related to GUI/DB tests it is highly possible that, given the time constraints, it is not feasible to thoroughly test every small component of functionality in these tiers when the target system is GUI/DB intensive. A less rigorous testing process can in turn be expected to lower the performance of TDD in terms of quality and perhaps productivity.

Finding empirical data on the impact of domain and tool specific issues is difficult since it is hard to quantitatively formulate the precise GUI/DB intensity of a target system. No empirical study was found to quantitatively examine the impact of such issues on TDD's performance. In the trials included in the above meta-analysis, most of them involved development using one of the popular object-oriented languages and where the experimental task was not very GUI/DB-intensive. Arguably, these conditions were favorable for the application of TDD. As a result, the nature of the impact of these issues on TDD is still relatively unknown. Nonetheless, one study [89] reported test coverage results with/without the inclusion of GUIs; a drop of 20% was discovered upon the inclusion of GUIs into the coverage analysis.

### 3.2.5 Threats to Validity

The major obstacle in conducting this meta-analysis was the lack of data available for computing the standardized effect size in each trial. Although this obstacle was partially overcome by using an unstandardized effect size measure, all unstandardized measures within the context of this research suffer from two principal disadvantages. Firstly, since indicator variables used to operationalize the outcome constructs may differ from study to study, differences in scale may affect the accuracy of comparison between the results of two studies. Secondly, ignoring the standard deviation within the results of subjects in a group might result in an incorrect assessment of the actual effect size when comparing the results of two groups a study, thus leading to conclusions that are misleading or exaggerated.

Underlying the selection of a parametric effect size measure such as the Hedges' $g$ is the assumption that the outcome construct being investigated is normally distributed; if the distribution of one or both of the outcome constructs is largely skewed, this assumption could be violated. For the purpose of this analysis, it cannot be confirmed whether or not the outcome constructs is normally distributed. However, choosing a non-parametric measure is not a feasible option as such measures can only be used for characterizing data but not for relating it back to a population [32]. Due to the lack of availability of raw data that could point to the right distribution to use, and considering that means are meaningful indicators for both of the outcome constructs, a parametric measure was adopted as the best option.

As observed by Shull *et al.* [80], many among the studies on TDD consider only the implementation time when calculating productivity thus ignoring any impact on the duration of other phases of development. Consequently, it is difficult to achieve a comprehensive understanding of the productivity-related impact of TDD, and effect sizes reported in such studies must be taken with caution as they might not reflect the true extent of the impact.

Publication bias [32] is another factor that should be considered when a meta-analysis is under-

taken but the data in the studies included here—esp. those in the industrial group—is insufficient to make a meaningful analysis of this kind.

## 3.3 Discussion

The summary effect sizes in our meta-analysis suggest that TDD has a small to medium sized overall positive effect on quality and a very small to negligible negative effect on productivity. The trials' effect sizes were more assertive of the summary effect size on quality as opposed to the one on productivity. The correctness of the summary effect size on productivity can be questioned due to the sometimes large and contradictory differences between the summary value and those of individual trials. In order to gain a better understanding of the TDD-based process, it was deemed necessary to rationalize the variation in effect size across the trials. Five moderator variables were identified as influencing the outcome when applying TDD: task size, developer experience, test granularity, test effort and domain & tool specific issues. In this section these variables will be used to reason the variation in effect size as illustrated by the histograms. Using the same grouping strategy as the previous sections, the variation to be addressed can been classified into two types: inter-subgroup and intra-subgroup.

Inter-subgroup variation, or differences in the results of the two subgroups, can mostly be attributed to the impact of the task size and developer experience. The above analysis on moderator variables found a strong correlation between the task size and the magnitude of the improvement in quality. Hence it is possible that since programs implemented in the academic studies were generally much smaller in size in comparison with programs implemented in the industrial studies, the resulting improvements in quality also tended to be relatively smaller. Another possible reason for inter-subgroup variation is the difference in experience level. Although the impact of experience level is not immediately clear, in the study by Hofer & Philipp [35], where professionals applying TDD took longer to complete the task than novices, the added time spent was

attributed to developers having spent more time refactoring code. Also, professionals in this trial were found to maintain higher test coverage levels. Hence, using reasoning similar to that given by Flohr & Schneider [24], it could be that higher coverage levels in the case of professionals gave them increased confidence in refactoring code without the fear of breaking any functionality, thus leading to them having spent more time refactoring code. The extra time spent in refactoring code could perhaps be used to justify the drop in overall productivity noticeable in the industrial studies. Moreover, according to Shore & Warden [79], a simpler design, due to a higher refactoring frequency, would also be expected to result in a reduction in the number of bug breeding grounds. Consequently, larger improvements in quality would be expected in industry which can in fact be affirmed looking at the individual trial effect sizes. However, the impact of TDD on design is still a subject of debate in empirical research and should be further investigated. One more reason for the variation in effect sizes amongst the two subgroups could be the amount of design work done upfront prior to the start of the implementation. As highlighted in the section on rigor, there is much confusion surrounding the exact definition of a TDD-based process. One area that is relatively unclear is the amount of design work that should be done upfront. Most of the studies conducted in academia failed to include any details on any design work done in advance. In contrast, in four of the trials conducted in industry the authors admitted to constructing a reasonably detailed upfront design prior to the iterative TDD cycle [66]. Perhaps, more emphasis on design work upfront, as seen in industry, makes the resulting design more manageable hence facilitating higher improvements in quality.

Intra-subgroup variation, or differences in the effect size within each subgroup, are plausibly the result of differences in test granularity, test effort and domain/tool support. Firstly, larger relative differences in the number of tests written amongst the two development approaches in some trials could have resulted in the higher productivity improvements. Although test granularity was not found to be linearly correlated with an improvement in quality, a finer level of test granularity in some studies could have raised the minimum quality achievable [23; 37]. Secondly, differences

in test effort amongst the trials in a subgroup could have resulted in differing levels of improvement in developer understanding of the system, as a result of applying TDD, and consequently the differing levels of improvements in quality and even perhaps productivity. Lastly, although in the larger share of the trials the experimental task was "favourable" to the application of TDD, domain and tool support issues are expected to have made it more difficult to apply TDD in some trials, especially in the industrial ones, and in these cases both improvements in quality and productivity are expected to be lower than would have otherwise been the case.

# Chapter 4

# A System Dynamics Model of TDD

This section presents a System Dynamic simulation model of a TDD-based development process. The goal underlying the development of this model is to contribute towards the broader effort of producing an elaborate model of the XP-based development process. As an application of this model, the impact of moderating variables identified in the meta-analysis will be further examined . The presented model has only been validated with tests that relate to the quality construct. Consequently, this model should only be used to analyze the impact of changes on quality. Although productivity was kept in mind when making decisions related to the design of this model, these decisions were made following intuition derived from empirical literature on TDD. Since the empirical literature provides no definite conclusions on TDD's impact on productivity, as illustrated by the meta-analysis, all productivity-related decisions made while designing this model might not be entirely correct, and hence any productivity-related results have not been included in this chapter.

## 4.1 Overview of SD modeling

### 4.1.1 Introduction to Software Process Simulation (SPS)

To introduce the field of SPS some key concepts must first be understood. According to Kellner *et al.* [45]:

1. A *model* is an abstraction (i.e., a simplified representation) of a real or conceptual complex system. A model is designed to display significant features and char-

acteristics of the system, which one wishes to study, predict, modify, or control. Thus a model includes some, but not all, aspects of the system being modeled.

2. A *simulation model* is a computerized model that possesses the characteristics described above and that represents some dynamic system or phenomenon.

3. A *software process simulation* model focuses on some particular software development /maintenance / evolution process. Since all models are abstractions, a model represents only some of the many aspects of a software process that potentially could be modeled namely the ones believed by the model developer to be especially relevant to the issues and questions the model is used to address.

The application of simulation modeling techniques to the discipline of software development is an area that has been gaining momentum over the past two decades. One of the main categories of applications for which SPS has been found to be useful is process improvement and technology adoption [45]. Within this category, SPS provides an inexpensive alternative for policy makers to evaluate the consequences of their process improvement decisions before putting them into practice.

There are two main approaches that have been used for SPS: *Discrete-Event* and *Continuous-Time*. Discrete-event approaches involve passing a system through a sequence of events at discrete points in time. Systems are modeled as a set of entities that are interconnected using logical relationships [61]. Moreover, entities are described using the their associated attributes that can be uniquely set. Continuous-time approaches [53] provide a "higher-level" of abstraction of the system or process being modeled focusing on patterns and trends in behaviour as opposed to system responses to individual events. Systems consist of a collection of continually changing elements which are represented using a set of differential equations that are integrated over time.

From a modeler's perspective, each of these approaches have their advantages as shown in Table 4.1. The main advantage of the discrete-event modeling approach is that it allows entities to be

Table 4.1: Advantages/Disadvantages of Discrete-Event & Continuous-Time Simulation Approaches (Adapted from Kellner *et al.* [45])

| | Discrete-Event | Continuous-Time |
|---|---|---|
| Advantages | attributes allow entities to vary<br><br>CPU efficient since time advances in events<br>queues and interdependence capture resource constraints | accurately captures the effects of feedback<br>clear representation of the relationships between dynamic variables |
| Disadvantages | continuously changing variables not modeled accurately<br>no mechanism for states | sequential activities are more difficult to represent<br>no ability to represent entities or attributes |

uniquely identifiable, using values set for their attributes, whereas the continuous-time approach provides more assistance in modeling complex feedback loops between elements [50]. Discrete-Event approaches are typically favoured for modeling sequential processes as the modeling mechanism naturally lends itself for modeling this type of process. However, where the advantages of continuous time approaches are required, either both a discrete-event and a continuous-time approach can be combined to form a Hybrid approach [50] or a continuous time approach can be manually extended for sequential capability [59].

### 4.1.2 A Brief Introduction to System Dynamics

System Dynamics (SD) modeling is a popular implementation of a continuous time approach. It was introduced by Jay Forrester to apply engineering principles to social systems [25], and was first used in software process simulation modeling by Abdel-Hamid and Stuart Madnick [1].

The different types of elements in SD models will now be discussed based on definitions from [53]. A graphical representation of these elements is given in Fig. 4.1.

A *Level*, also called a Stock, represents an element that accumulates or shrinks over time, eg. defect level, personnel level etc.

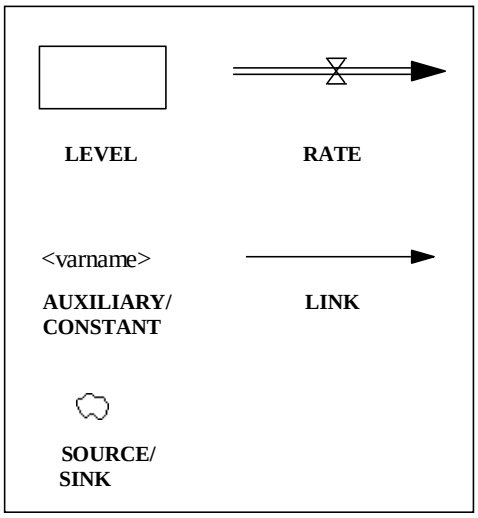



Figure 4.1: System Dynamics Modeling Elements

A *Rate*, also called a Flow, represents an action and results in material flow to/from attached levels over time depending on its value, eg. rework rate, implementation rate etc.

An *Auxiliary* variable assists in adding details to level and rate elements. They may be "score-keeper"' variables, eg. percent of job completed, or simply constants, eg. average delay time.

A *Source/Sink* is a point of reference for communication with systems or processes external to the current system that are not being modeled. They act as infinite suppliers/repositories to the attached rates.

A *link* is used to represent a dependency between two elements. Links are the essential tool used to implement feedback and control.

SD models view the target system or process as a collection of *flows* that accumulate in different *levels* over time [61]. The two important aspects of an SD model are its structure and behaviour [49]. The structure of an SD model refers to the set of model elements and how they are interconnected. The behaviour of an SD model refers to the way that model elements vary over time.

In an SD model all rates are active at all times. To implement a sequential process, as mentioned in the previous section, constraints must be manually inserted to prevent all rates from running at once.

## 4.2 The TDD SD model

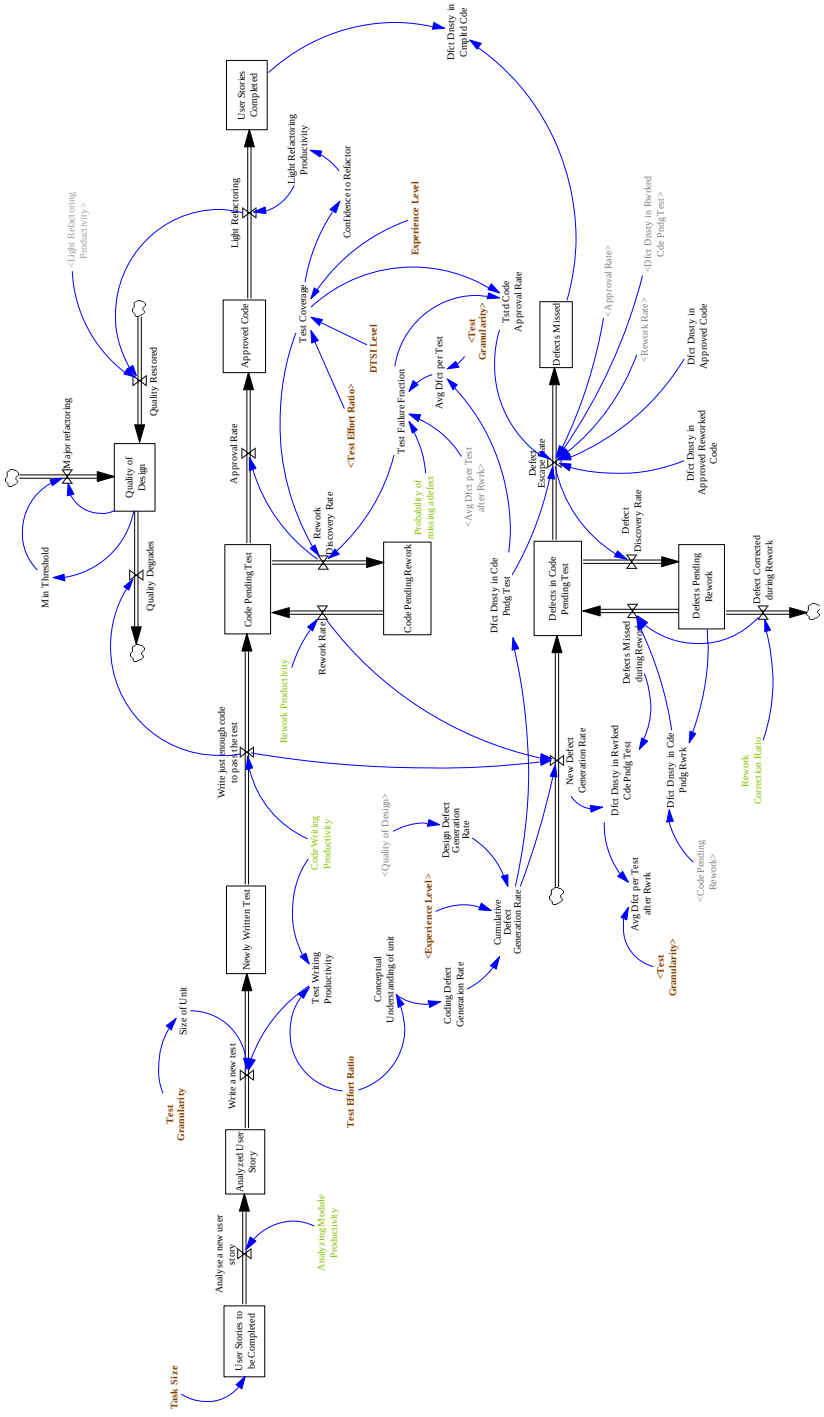



Figure 4.2: The TDD System Dynamics simulation model (Adapted from Rahmandad & Hu [72] and Cao *et al.* [14])

The System Dynamics model of the TDD-based development process is given in Figure 4.2. A simplified view of the model is presented with some elements being hidden for ease of understanding. The complete model is given in Appendix B.

In essence, the model displays the interaction between three subsystems using a parallel co-flows structure. In this section each of the co-flows, and their respective subsystems, will be described in detail, namely the *Development*, *Defect* and *Design* co-flows.

### 4.2.1 The Development Co-flow

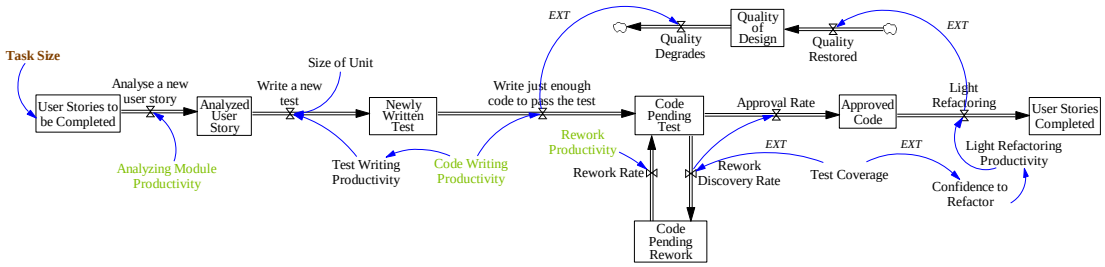



Figure 4.3: The Development Co-Flow (Connections to the other co-flows are marked *EXT*)

The *Development* co-flow, shown in Figure 4.3, is the main co-flow in the parallel structure and models the development of a predefined number of user stories following the TDD approach. Details of the development flow of a user story are shown in Figure 4.4 and are described next.

The development of each story begins with a short analysis/design phase where the high-level architecture of the story-to-be-implemented is approximated. This activity is followed by story implementation, one test at a time iteratively in the TDD manner. Each TDD iteration commences with the developer writing a new test. This is followed by the writing of just enough code to pass the test. Once the developer feels that the newly added code is in working order, he/she runs the test suite which includes the newly added test. Instead of modeling the process such that the newly added test sometimes fails and at other times passes, the process has been modeled such that a certain portion of the newly added code is always approved whereas the remaining code

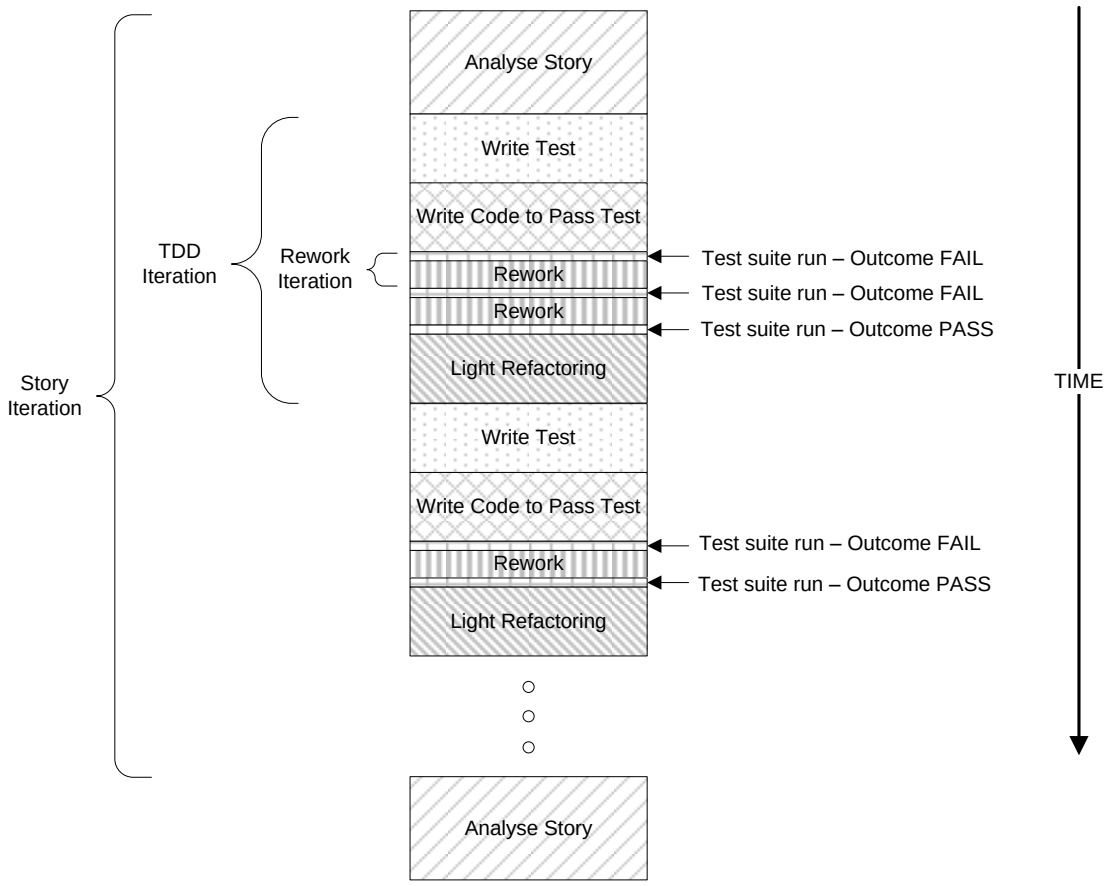



Figure 4.4: Development Flow of a User Story

is identified as being defective and left to be reworked. This modeling change was made as it facilitated easier modeling of the *Defect* co-flow and because it does not change the final result. The next activity after running the test suite is reworking the faulty code. This is followed by re-running the test suite. The reworked code itself gets divided into code that is approved and code that is identified as being defective. New rework iterations are triggered as long as the rework generated in the previous iteration is above a certain threshold. Once the rework phase is over, code is refactored marking the end of a TDD iteration.

The scope of this model has been limited to modeling the design and implementation phases of the SDLC as TDD predominantly impacts these phases. As such it assumes that requirements have been collected beforehand. On the other end, this model only portrays a development process

up to the end of unit testing in the traditional sense.

The three variables in this co-flow are *Task Size*, *Test Granularity* and *Test Effort Ratio*. Task size is defined as the number of user stories (derived from the number of use cases) that are to be completed at the start of the development process. Test granularity defines the number of story points that are covered by a test in a TDD iteration i.e. a higher value of test granularity would result in a larger number of tests per story iteration. Test effort ratio is used to indicate the amount of time spent by the developer thinking about each story point prior to writing the test(s). It is modeled as a ratio so that the time spent writing test(s) for a story point can be stated in relation to the time spent writing code that implements that story point.

With regard to the development co-flow this model makes a number of assumptions:

- Since productivity, defined as the amount of material that can be processed in a time step, is not to be investigated using this model, for simplicity purposes the individual productivities for all activities modeled in the co-flow remain constant for the duration of the simulation.

- All user stories are assumed to have the same size and complexity.

- All stories are divided into the same number of tests which is determined by the test granularity.

- The same constant amount of test effort, determined by the test effort ratio and the code writing productivity, is spent on each story point.

### 4.2.2 The Defects Co-flow

The *Defects* co-flow, shown in Figure 4.5, models the generation and detection/removal of defects over the duration of the development process. The quality of the system being developed is measured by the current defect level of the system; the quality of the end product is evaluated as the

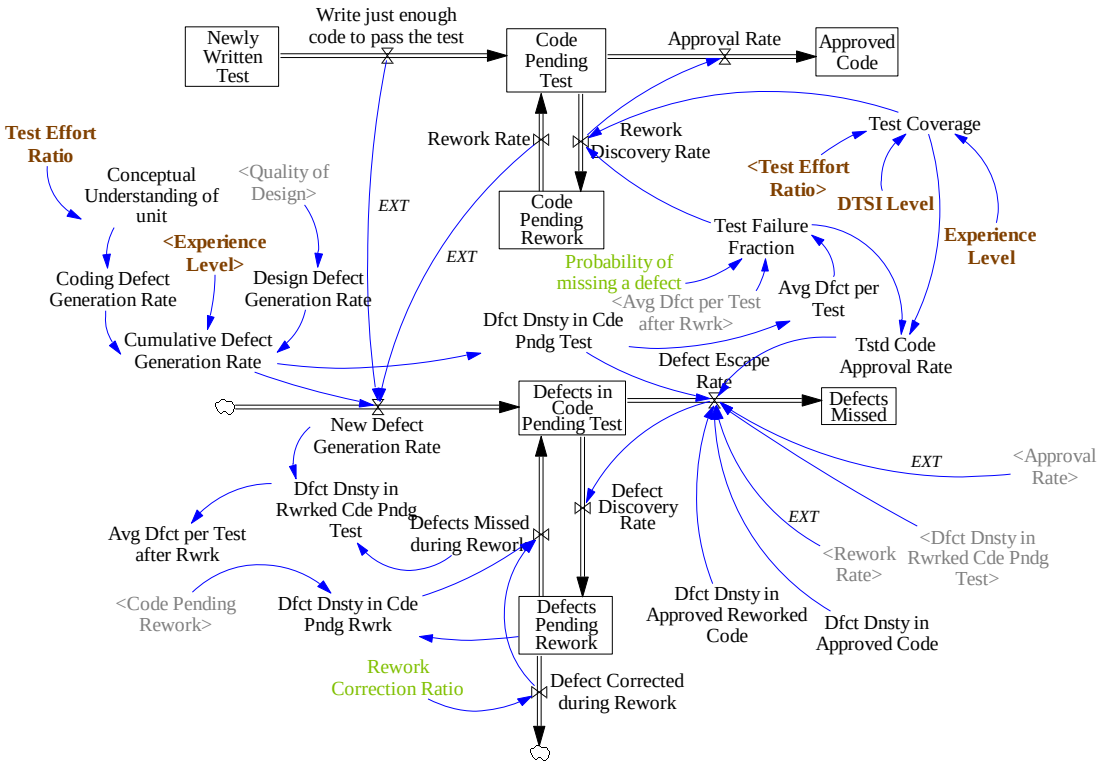



Figure 4.5: The Defects Co-Flow (Connections to the other co-flows are are marked *EXT*. Adapted from Rahmandad & Hu [72].)

number of defects in the system once all user stories have been completed with all identified defects having been fixed. This defect level is indicated by the model element *Defects Missed*. In this section the defect generation and the defect detection process is explained.

#### 4.2.2.1 Defect Generation in the Model

The defect generation process has been modeled such that the number of defects generated in a TDD iteration is proportional to the amount of code being written. Within the confines of the modeling scope, based on the defect classification scheme of the COQUALMO model [36] the two major sources of defects in the modeled development process are the design and coding activities. According to [9; 43; 87], the nominal defect generation rate in the design and coding activities is typically arround 25 defects/KLOC and 15 defects/KLOC respectively. Since these activities are somewhat merged when following TDD, in this model defects are typically injected at a combined

rate of 40 defects/KLOC when the code for a test is being written.

However, the defect generation rates for these activities are not fixed at the typical values mentioned above; rather these typical values are used as middle values on a scale with the actual values of the rates being determined by other elements in the model. In this model the range of the design and coding defect generation rates is chosen to be 12.5-37.5 defects/KLOC and 7.5-22.5 defects/KLOC respectively. With regard to the design defect generation rate, it is modeled to be dependant on the current quality of design. The intuition behind this dependancy is that the process of updating/enhancing the design is more error-prone when the existing design is in a worse condition. With regard to the coding defect generation rate, it is modeled to be dependant on the developer's conceptual understanding of the story point(s) being implemented, which in turn is dependant on the variable *Test Effort Ratio*. This is rationalized using the intuition that if developers spend more time pondering over each story point, they will have a better understanding of the system, and consequently will be less likely to introduce defects during implementation. Finally, since the number of defects generated in both the design and coding activities is dependant on developers' experience, the sum of the activity defect generation rates is adjusted, depending on the variable *Experience Level*, to give the actual rate at which defects are introduced into the system.

In addition to the initial implementation, the rework activity has also been modeled to give rise to defects in the presented model. In particular, this model considers each iteration of rework giving rise to defects of two types. The first type of defects are those defects that previously did not exist in the code and have been newly introduced by the rework activity itself. The second type of defects are those defects that were present prior to the rework activity and still remain in the code after it has been reworked.

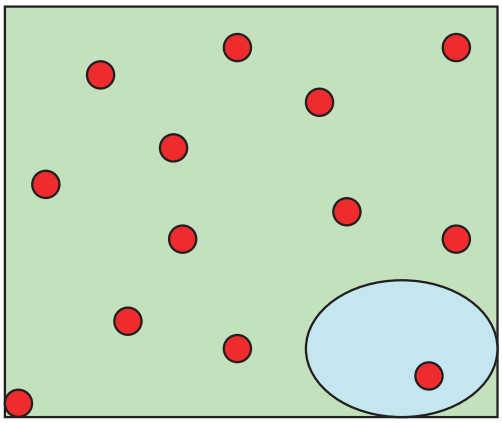


Figure 4.6: The framework for modeling of defects per story point. The above rectangle represents the space of story points; the dots represent the defects; the oval represents a test that covers some story points and any defects in them. (Adapted from Rahmandad & Hu [72])

#### 4.2.2.2 Defect Detection & Removal in the Model

The number of defects that is detected, and subsequently removed, in a TDD iteration is modeled to be dependant on the portion of the newly added code that is identified as being defective and is subsequently reworked. An important aspect of the presented model is how the code pending test is divided into code that is approved and code that is identified as being defective. The main idea underlying the scheme on how code is divided, and consequently how the defects generated in an iteration are divided into those that are reworked and those that escape into the final product, has been taken from Rahmandad & Hu [72] and is explained in this section.

The conceptual framework for modeling defects is shown in Figure 4.6. Since the generated defects can exist anywhere in the space of implemented code with equal probability, the defects per area(unit: story point) meets the criteria of a Poisson distribution. Adopting this distribution, the probability of $k$ defects being covered by a test of area $a$ assuming a defect density $d$(unit: defects/story point) is given in Equation (4.1), eg. assuming a defect density of 1.5, the probability

of one defect being covered in a test having a granularity level of 1 (i.e. $a = 1$) is $1.5e^{-1.5}$.

$$P(Defects = k) = \frac{e^{-da}(da)^k}{k!} \tag{4.1}$$

In a practical enviroment, it is common for the testing process to be flawed. There are two causes of imperfect testing that are considered within the presented model. Firstly, the written test might not be rigorous enough to exercise all of the branches through code; thus some of the defects in code will be missed during testing. Assuming that the probability of missing a defect during the testing process is $\lambda$, the probability of a test, that covers $k$ defects, passing is $\lambda^k$. Consequently, the probability of a test passing is given in Equation (4.2). Also, the expected number of defects in code that is covered by a test which passes, i.e., the defect density in approved code related to a test, is given in Equation (4.3).

$$P(Test\ Passing) = \sum_{i=0}^{k} P(defects = k)\lambda^k = e^{-da(1-\lambda)} \tag{4.2}$$

$$Defect\ Density = \sum_{i=0}^{k} kP(defects = k)\lambda^k = da\lambda e^{-da(1-\lambda)} \tag{4.3}$$

The second cause of imperfect testing is lack of adequate test coverage. A portion of the code might fall outside the coverage of its associated test. Consequently, all defects in this portion of the code escape into the final product undetected. The defect density of code that is approved without testing, due to coverage issues, is assumed to be the same as that of the code prior to the testing process.

Based on the discussion on imperfect testing, the code pending test is classified into three categories: code that is tested and rejected, code that is tested and approved and code that is approved without testing. The first of these categories is code that is identified as being defective, and is subsequently reworked, whereas the latter two categories form the set of approved code. The

portion of the code pending test that falls into the first category is formulated as the product of *Test Coverage*, which denotes the coverage level, and *Test Failure Fraction*, which denotes the fraction of the code under coverage that fails or is rejected during testing. The *Test Failure Fraction* is computed as 1-$P(testpassing)$.

With regard to the division of the generated defects, the number of defects that escape into the final product is the sum of the number of defects from each of the latter two code categories mentioned above. The number of defects from the first of these two categories, code that is tested and approved, is formulated as the product of the portion of code in this category and the defect density in tested and approved code given in (4.3). The number of defects from the other category, code that is approved without testing, is formulated as the product of the portion of code in this category and the defect density of the code prior to the testing process.

There are four variables related to the defect detection & aspect of the defect co-flow namely *Experience level*, *DTSI level*, *Test Effort Ratio* and *Test Granularity*. The DTSI level denotes the difficulty faced in applying TDD due to domain and technology specific issues. The first three of these variables collectively determine the level of test coverage maintained for the duration of the simulation. *Experience level* and *Test Effort Ratio* are modeled to be positivitely correlated with test coverage whereas *DTSI level* is modeled to be negatively correlated. *Test Granularity* is one of the elements used to determine what fraction of the portion of the newly added code that lies under coverage is identified as being defective i.e. the *Test Failure Fraction*.

### 4.2.3 The Design Co-flow

Although investigating the impact of TDD on design quality is not one of the objectives of this research thesis, considering the emphasis on agile design and refactoring, any analysis on the TDD practice should, at least to some extent, consider the impact of changes to design on other aspects of the development process. The design co-flow shown in Figure 4.7 presents a simplified view

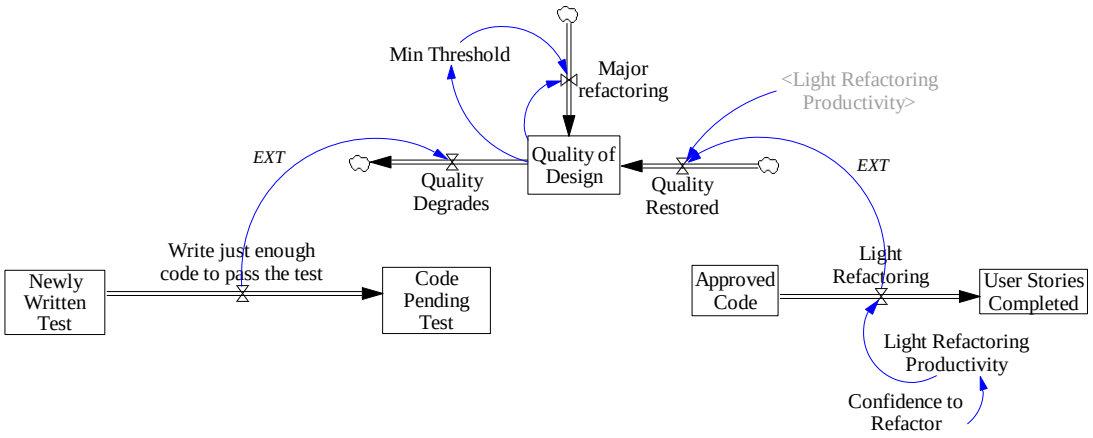



Figure 4.7: The Design Co-Flow (Connections to the other co-flows are are marked *EXT*. Adapted from Cao *et al.* [14].)

of the evolution of design over the development cycle that has been adapted from Cao *et al.* [14]. This section elaborates on the details of this viewpoint and the interactions of the Design co-flow with the Development and Defect co-flows.

In TDD theology refactoring is tightly interleaved with coding and testing. In practice, however, the frequency with which refactoring takes place and the exact point(s) during development at which it is undertaken both vary from company to company. Consequently, for simplicity purposes, this model assumes that light refactoring takes place at the end of every TDD iteration.

In literature on software design, there is no single metric that is used as a universal indicator of the quality of design [14]. The chosen metric varies from context from context depending on the purpose of the product and its future vision. Consequently, the perception of the quality of design has been represented as a model element that ranges between 0 and 1. The value of the *Quality of Design* varies over the duration of the development process. The view of design evolution assumed in this model is explained next.

Prior to the start of the development process, the *Quality of Design* level assumes a value of 1 denoting a perfect design since no activities that could lower design quality have taken place. Once development has started, in each TDD iteration the quality of the design degrades as functionality is added/updated. Following the implementation of functionality, in each TDD iteration the de-

sign quality is then restored by some light refactoring. In practice, the amount of refactoring that takes place alongside coding is usually insufficient [67] due to factors such as schedule pressure. Consequently, at the end of each iteration the quality of design is lower than its level at the start of the iteration. As a result of falling a small amount of in each iteration, the quality of design eventually drops to an unacceptable level where the process of making changes is too expensive and error-prone. At this point, new development is stopped and a major refactoring phase is triggered where all efforts are geared towards refactoring code and improving its quality. Once the quality of design is back to an acceptable level, new development is resumed.

The Design co-flow interacts with both the defect generation and the detection/removal process of this model. With regard to defect generation, the current value of the stock *Quality of Design* determines the precise value of the *Design Defect Generation Rate* within the range specified earlier. As mentioned earlier, this modeling assumption can be rationalized on the reasoning that a better design reduces "bug breeding grounds" and hence reduces the number of design-related defects. With regard to defect detection/removal, a higher value of test coverage would result in developers spending more time refactoring code in each iteration thus reducing the overall drop in design quality at the end of every TDD iteration. The intuition behind this modeling assumption is explained as the following. In TDD the automated test suite acts as a tool for regression testing to ensure that the process of refactoring has not broken any functionality. A higher value of test coverage strengthens the safety guarantee offered by the test suite thus increasing developers' confidence in refactoring code and in turn resulting in them spending more time performing it [4; 14].

There are three variables impacting the design co-flow namely *Experience level*, *DTSI level*, *Test Effort Ratio*. The impact of all these variables is indirect and through the *Test Coverage* element of the defect co-flow.

## 4.3 Model Calibration and Validation

### 4.3.1 Model Calibration

Model calibration is one of the major obstacles in the field of software process simulation. This is partially due to the nature of the field where it is intrinsicly difficult to assign numerical values to all of the model elements. Moreover, effective calibration prerequisities the availability of data-sets from real projects which are not readily available. Traditionally, the earlier issue has been resolved using judemental estimation whereas for the latter issue modelers have resorted to data-sets that have been previously made available on the internet, research papers such as in [15; 61].

The high-level data used to calibrate this model was taken from the empirical research work by Dogsa & Batic [19] which provides sufficient details on a trial analyzing the use of TDD in industry. Some of the model elements were calibrated based on empirical data reported in other research on System Dynamics modeling of software processes. The remaining elements were calibrated using judgemental estimation. After the initial calibration the model was validated using the tests in the next section. Following that, the model was simulated and data on the evolution of the modeled TDD process was analyzed. The model was iteratively tweaked and re-calibrated after comparing simulation results with real-data in [19] and based on conversations with one of the authors of the study. A complete listing of model elements and their values is given in Appendix B.

### 4.3.2 Model Validation

According to Forrester & Senge [26], there is no single test that can be used to "validate" a System Dynamics model. Instead, multiple tests can be used to build confidence in a model's results. Based on the two important aspects of an SD model, structural and behavioral, a variety of tests related to each aspect can be conducted to strengthen model validity. This section details the measures

that were taken to ensure that the model achieved a sufficient level of validity with regard to each of these two aspects. The validation process consisted of two steps. The first step involved a structural assessment of adjacent elements, model subsystems, and overall model structure [14]. Once sufficient confidence in the structural validity of the model had been established, in the second step the behavioral validity of the system was assessed by analyzing the extent to which the behavior generated by the model structure mimicked empirical practice [26].

#### 4.3.2.1 Structural Validation

Sterman [86] specifies 12 tests for assessing the model structure; 6 of these tests are related to structural validation and were undertaken for this part of the research. The tests that were conducted include Boundary Adequacy, Structure Verification, Dimensional Consistency, Parameter Verification, Extreme Conditions and Integration Error. Table 4.2 shows a summary of the analysis on structural validation providing a description of each test's intended purpose, the steps taken in this research and test results.

#### 4.3.2.2 Behavioral Validation

The behavioral validation of a model aims to assess the extent to which the model reproduces the dynamic behavior found in practice (instead of comparing point estimates) [14]. Sterman [86] mentions six tests related to behavioral validation; however, only four were found to be applicable in this research. A summary of these tests is given in Table 4.3.

Table 4.2: Summary of Structural Validation Tests (Selected from tests by Sterman [86])

| Test | Purpose of Test | Procedures Conducted in this Research | Test Results |
|---|---|---|---|
| Boundary Adequacy | Determines whether the important concepts are endogenous to the model | Model Subsystems were exhaustively compared with available literature to ensure that any potentially important feedback loops were not omitted | Model improved to include all of the identified endogenous concepts within model scope |
| Structure Verification | Determines whether the model structure is consistent with available descriptive knowledge of system being modeled | 1) Relationships between model elements, inputs and outputs were compared with previous SD models in the field of software development 2) Defect and design sub-sytems have already been validated in previous research Rahmandad and Hu [2010]; Cao et al. [2010] | Passed |
| Dimensional Consistency | Determines whether the equations used in the model are dimensionally consistent | Verified dimensional consistency using Vensim's built-in tool | Passed |
| Parameter Verification | Determines whether parameter values are consistent with available descriptive and numerical knowledge of system being modeled | Every model element has been set based on a value extracted from one of data from real project on TDD, other empirical findings in literature and judmental estimation | Passed |
| Extreme Conditions | Determines whether system illustrates logical behavior when selected parameters are assigned extreme values | Analyzed simulation results to ensure correctness under extreme values of input for each variable and improved model whereever required | Passed |
| Integration Error | Determines whether simulation results are sensitive to the time step | 1) Manually inspected the differential equations to ensure independency of time step. 2) Compared results under three different time steps. | Passed |

Table 4.3: Summary of Behavioral Validation Tests (Selected from tests by Sterman [86])

| Test | Purpose of Test | Procedures Conducted in this Research | Test Results |
|---|---|---|---|
| Behavior Reproduction | Determines whether model reproduces behavior of interest in the system | 1) Model behaviors compared with those from real project 2) Model behaviors compared with behaviors of similar SD models | Model improved to mimic behavioral dynamics found in real project and other research literature |
| Behavior Anomaly | Determines whether the change/deletion of assumptions results in anomalous behaviors | If assumptions are significantly changed or the refactoring effort is removed, the model displays anomalous behaviour | Performs Well |
| Family Member | Determines whether model can generate behaviors of other instances in same system | Model represent behaviors generated in only one class of development i.e. Test-Driven Development | N/A |
| Surprise Behavior | Determines whether model generates previously unobserved behavior | None of the model elements were found to exhibit behavior counter to intuition and literature | Perfoms Well |
| Sensitivity Analysis | Determines the degree to which model's behaviors or results change when the underlying assumptions are varied | univariate and multivariate analyses at differing values for the identified moderator variables | Performs Well |
| System Improvement | Determines whether modeling the process assisted in improving the system | Goal was solely to model the existing system | N/A |

## 4.4 Results

### 4.4.1 Research Questions

The goal underlying the development of the presented TDD SD model is to contribute towards the larger scale effort of developing an SD model of an Extreme Programming(XP)-based development process. As an application of this model, it is simulated to analyze the effects of changes in the moderator variables identified in Chapter 3. Related to this application, the simulation process is

expected to answer the questions below. Both of these questions are answered using results from multiple multivariate and univariate sensitivity analyses that were conducted using the moderator variables as the sensitivity parameters.

- *RQ:1* What is the impact of varying the moderater variables on the defect level of the final product?
- *RQ:2* How do changes in the moderator variables influence the defect level of the final product? Can we confirm the results from the meta-analysis using the simulation data?

### 4.4.2 Analysis of Simulation Results

#### 4.4.2.1 Impact on Defect Level

For answering *RQ:1*, a multivariate sensitivity analysis was conducted using all of the moderator variables as sensitivity parameters except the task size. In particular, 1000 simulation runs were conducted for the model with the values assumed by *Experience Level*, *Test Effort Ratio*, *Test Granularity* and *DTSI Level* in each simulation run being selected from the distributions given in Table B.1 in Appendix B. A task of roughly the same size as the one chosen in Dogsa & Batic [19] was preset for all the simulation runs.

A histogram of the number of defects, recorded at the end of each of each simulation run, is shown in Fig. 4.8. From the diagram the defect distribution appears to be Beta in shape. In order to assess the goodness-of-fit of the model's defect distribution to a Beta distribution, the Kolmogorov-Smirnov (K-S) test was performed. For the K-S test the null hypothesis is that the distribution can be approximated by Beta distribution. Results from the test at the 5% significance level are shown in Table 4.4.

Since the Kolmogorov-Smirnov statistic *ksstat* is less than the critical value *cv* the null hypothesis is accepted and we can assert that within the modeling scope the number of defects generated can be approximated using a Beta distribution. However, outside the modeling scope it is plausible

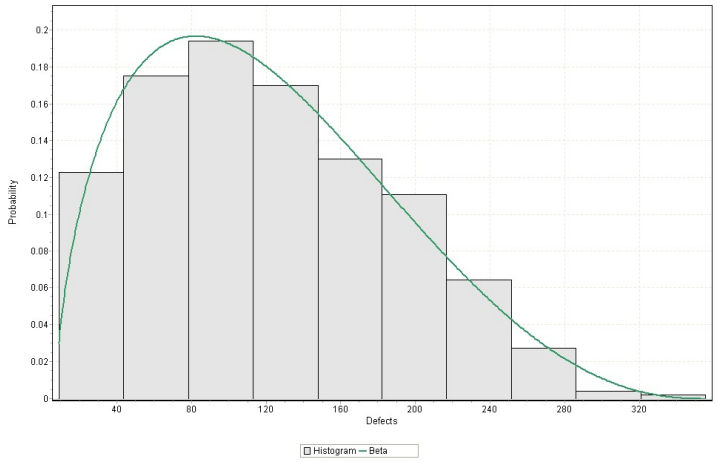



Figure 4.8: Distribution of defects in final product (i.e. model element *Defects Missed*) over 1000 simulation runs

Table 4.4: Kolmogorov-Smirnov test on the defect distribution of the TDD model

| Attribute | Value | Comments |
|---|---|---|
| $\alpha$ | 122.83 | Sample Mean |
| $\sigma$ | 66.27 | Sample Standard Deviation |
| $\sigma^2$ | 4391.71 | Variance |
| *h* | 0 | Null Hypothesis (0: accepted 1: rejected) |
| *p* | 0.86 | P value (if $p < 0.05$, can reject null hypothesis at the 95% confidence level) |
| *ksstat* | 0.019 | Kolmogorov- Smirnov test statistic |
| *cv* | 0.043 | Critical Value |

that the defect level can be approximated by a Gaussian distribution. In the conducted simulation runs, the shape of the defect distribution is most likely the result of the relationships of both the *Experience Level* and *Test Effort Ratio* with *Test Coverage* which are modeled to be approximately logarithmic in shape.

#### 4.4.2.2 Nature of the Impact on Defect Level

To answer *RQ:2* the impact of changes in each variable will now be discussed.

*Analyzing the effect of changes in Task Size*

The meta-analysis in Chapter 3 found a correlation between the task size and the magnitude of the improvement in quality. The intuition given behind this correlation was that benefits of TDD might not be immediately apparent and it may take some time before they become visible. To

Table 4.5: Simulation Results for different Task Sizes

| Task Size | # of Use Cases | Mean(Defects) | Variance(Defects) |
|---|---|---|---|
| T1 | 20 | 36.17 | 19.35 |
| T2 | 40 | 74.02 | 39.36 |
| T3 | 63 (from research by Dogsa & Batic [19]) | 122.83 | 66.27 |

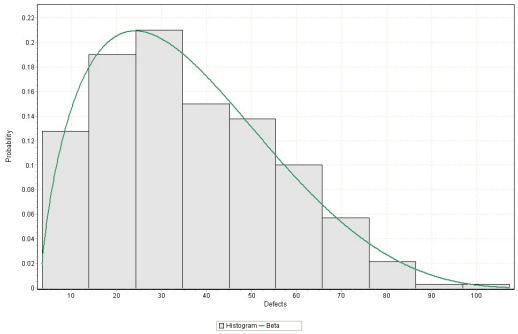
affirm this claim the simulation process, whose details are specified in the analysis for *RQ:1*, was repeated using two different task size values. In other words, the model was simulated 1000 times for each of the two new task size values.

It is important to highlight that the model uses the unit 'user story' for the requirements whereas Dogsa & Batic [19] use the unit 'use case'. The conversion from user stories to use cases has been done based on data given in their research work.

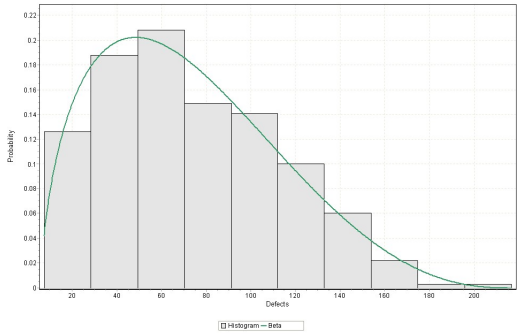
The mean and standard deviation of the defect distribution at the three values of the task size is given in Table 4.5. Moreover, the histograms of the three defect distributions are shown in Figure 4.9.

If the claim of benefits taking longer to materialize is true and a correlation between the task size and the magnitude of the improvement does exist, what must also be true is that the range of the magnitude of the improvement, spanned as a result of the performance of TDD varying from context to context (due to differing values of moderator variables other than task size), is correlated with the task size. As such, the range of the magnitude of the improvement should be larger for larger task sizes. Differences in the ranges of the magnitude of the improvement, under the three task size settings, can be estimated by comparing the differences in ranges of the defect-level distribution of the end product. From Figure 4.9 it is visible that the claim of the task size being correlated with the magnitude of the improvement in quality is indeed true since the defect distribution curve for task size T3 has a much larger horizontal spread in comparison to the curve for task T2, which in turn has a significantly larger horizontal spread than the curve for task size T1.

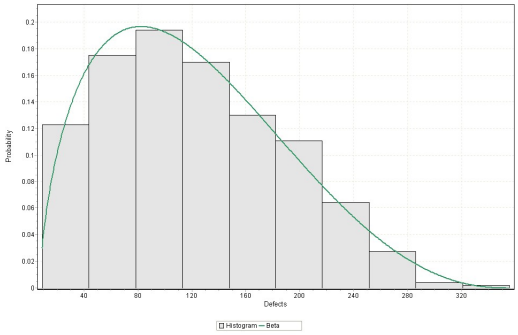
*Analyzing the effect of changes in Experience Level*




(a) Defect Distribution for T1




(b) Defect Distribution for T2




(c) Defect Distribution for T3

Figure 4.9: Defect Distribution for differing Task Size

Perhaps one of the most important variables with regard to its influence on the success of a TDD implementation is developers' *Experience Level*. As mentioned in the previous chapter, the *Experience Level* is an indicator of the proficiency with which developers can carry out development-related TDD tasks. From the defect co-flow it is visible that this variable influences the *Test Coverage* and *Cumulative Defect Generation Rate* elements. Consequently, this section will analyze the effect of varying these two concepts on the simulated development process.

For this part of the analysis the model was simulated three times with experience level being the only investigated variable. The remaining variables were held constant and assumed the same value across all three simulation runs. The three values chosen for *Experience Level* correspond to low, medium and high levels of experience. Details of the simulation runs are given in Table 4.6.

Looking first at the impact of experience on the defect detection/removal aspect, Figure 4.10 illustrates the changes in the evolution of the design quality as a result of the indirect impact of

Table 4.6: Simulation Details for Analyzing Effect of Experience Level

| Simulation | Experience Level | Task Size | Test Granularity | Test Effort Ratio | DTSI Level |
|---|---|---|---|---|---|
| EXPR 1(Low) | 0.4 | 63 use cases | 1 test/story point | 0.6 | 0.2 |
| EXPR 2(Medium) | 0.7 | 63 use cases | 1 test/story point | 0.6 | 0.2 |
| EXPR 3(High) | 0.9 | 63 use cases | 1 test/story point | 0.6 | 0.2 |

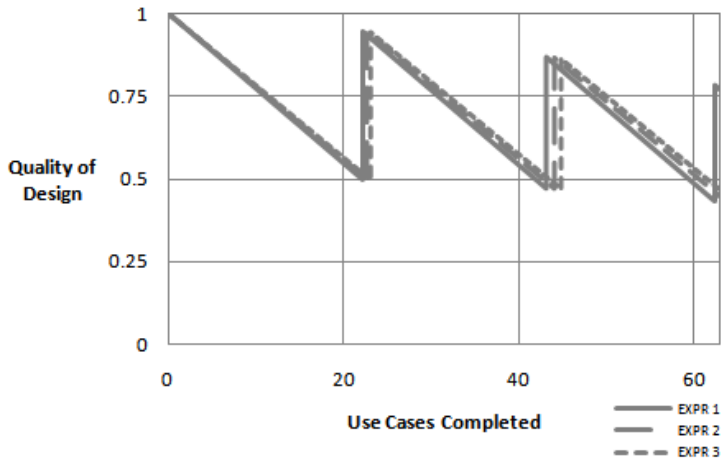



Figure 4.10: *Quality of Design* vs. *Use Cases Completed* for differing Developer Experience Level

changes in *Experience Level* through the *Test Coverage* element. As mentioned earlier, *Experience Level* has a positive impact on *Test Coverage*. It is visible from the graph that at a lower experience level the design quality deteriorates at a faster rate. Differences in design quality become more prominent as the simulations progress. Moreover, in simulation EXPR 1 major refactoring, indicated by the spikes in the graph, was triggered thrice over the duration of the development process whereas in EXPR 2 and EXPR 3 major refactoring only needed to be triggered twice for project completion.

With regard to the defect generation process, from the model it is visible that the *Experience Level* directly impacts the *Cumulative Dfct Generation Rate* and also has an indirect impact through the *Quality of Design*. Modeling the *Experience Level* to have two distinct impacts on the *Cumulative Dfct Generation Rate* might seem incorrect however this modeling decision can be rationalized

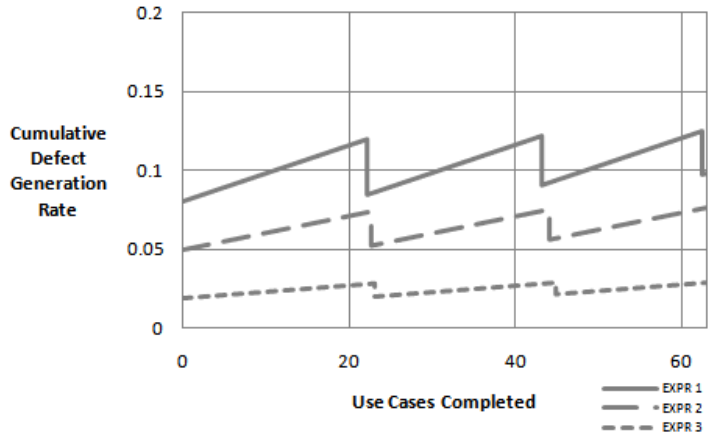



Figure 4.11: *Cumulative Defect Generation Rate* vs. *Use Cases Completed* for differing Developer Experience Level

in view of the assumption that the *Experience Level* model element provides a wholistic view of the different types of developer experience. As such, the direct impact on *Cumulative Dfct Generation Rate* represents the influence of design and coding experience whereas the indirect impact represents the influence of testing experience.

Figure 4.11 shows the impact of changes in *Experience Level* on the *Cumulative Dfct Generation Rate*. In the simulated development process the defect generation rate from coding remains constant for the duration of the simulation whereas that from design varies depending on the present design quality; consequently, the dynamics of the *Cumulative Dfct Generation Rate* are similar to that of the *Quality of Design*. In correlation with the design quality, the defect generation rate rose at a slower level when the experience level was higher. Consequently, at a higher experience level, the rise in defect generation between two adjacent major refactoring phases was much less. However, unlike the design quality, from the start of the simulation runs there were significant differences in the defect generation rate amongst the three runs.

Figure 4.12 shows the evolution of defects in the final product in the three simulation runs. From the above results on the *Experience Level*, it can be inferred that a higher level of experience reduces the defect generation rate in addition to improving the defect detection rate (due to its

Table 4.7: Simulation Details for Analyzing Effect of Test Granularity

| Group | Simulation | Test Granularity | Test Coverage | Task Size | Experience Level | Test Effort Ratio |
|---|---|---|---|---|---|---|
| GRP 1 | GRAN 1(High) | 0.5 test/story point | 0.95 | 63 use cases | 0.7 | 0.6 |
| | GRAN 2(Medium) | 1 test/story point | 0.95 | 63 use cases | 0.7 | 0.6 |
| | GRAN 3(Low) | 1.5 test/story point | 0.95 | 63 use cases | 0.7 | 0.6 |
| GRP 2 | GRAN 1(High) | 0.5 test/story point | 0.75 | 63 use cases | 0.7 | 0.6 |
| | GRAN 2(Medium) | 1 test/story point | 0.75 | 63 use cases | 0.7 | 0.6 |
| | GRAN 3(Low) | 1.5 test/story point | 0.75 | 63 use cases | 0.7 | 0.6 |

impact on test coverage). Intuitively, a lower defect generation rate and a higher defect detection rate over the project's duration is expected to result in fewer defects in the final product as displayed by the graph in Figure 4.12.

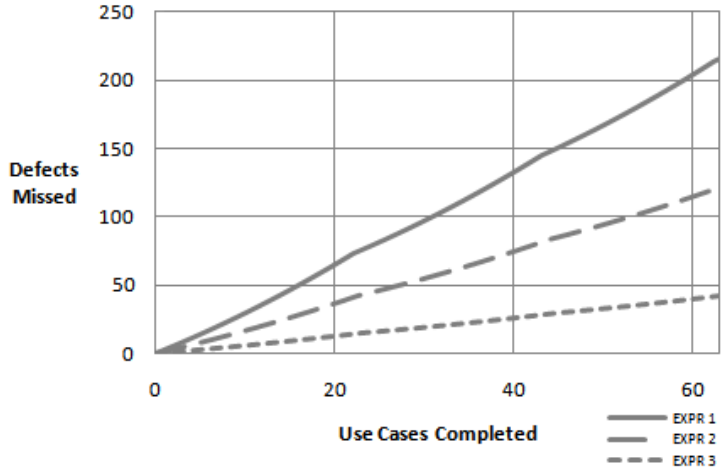



Figure 4.12: *Defects Missed* vs. *Use Cases Completed* for differing Developer Experience Level

*Analyzing the effect of changes in Test Granularity*

For the analysis on *Test Granularity* the model was simulated under three different settings for *Test Granularity* whilst keeping all other variables constant. The entire simulation process was then repeated for a different value of *Test Coverage*; there were 6 simulation runs that were conducted in total. To simplify the simulation process, for only this part of the analysis the model was

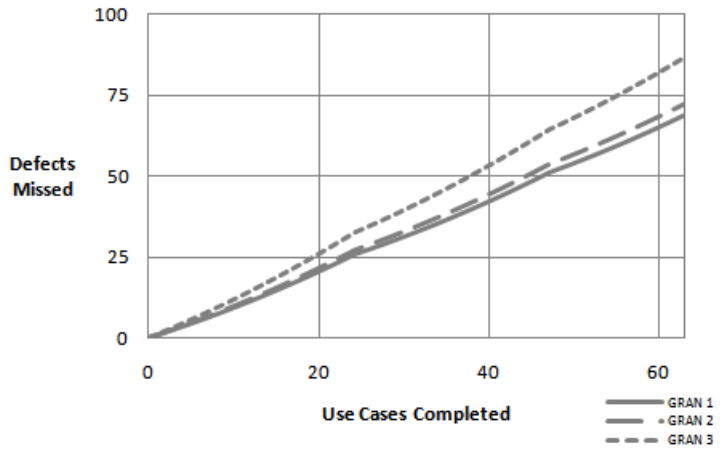



Figure 4.13: *Defects Missed* vs. *Use Cases Completed* for differing Test Granularity at Test Coverage 0.95

altered by removing the dependencies of *Test Coverage* and modeling it as a predefined constant. Accordingly, details of the simulation runs are given in Table 4.7.

Although the meta-analysis in Chapter 3 failed to find a significant correlation between the test granularity and quality, this result cannot be considered to be definite as test granularity was conceptualized differently from study to study and the number of studies used to formulate the claim were also relatively few. Figure 4.13 shows the evolution of defects in the final product i.e. the model element *Defects Missed* for the group of simulation runs where the *Test Coverage* was 0.95. In contrast to the result of the meta-analysis, the simulation process indicates that test granularity has a significant impact on the evolution of defects. This impact can be rationalized by extending the claim made earlier, of a higher test granularity level leading to a narrower development scope, by stating that a narrower development is likely to result in developers spending more time pondering over finer-grained details, and consequently is expected to result in developers making fewer mistakes thus reducing the defect level.

When the results from the simulation runs at *Test Coverage* value 0.95 are compared, the defect level drops by a significant amount upon the shift from a *Test Granularity* of 1.5 to 1.0, however, the drop is not as substantial from the shift from 1.0 to 0.5. This result can be justified using

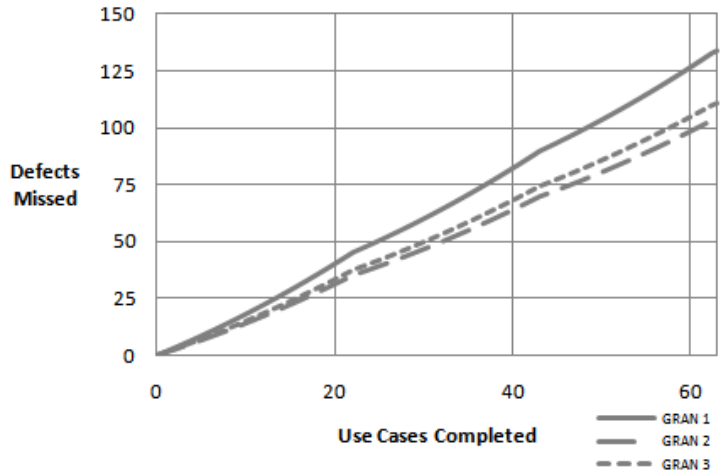



Figure 4.14: *Defects Missed* vs. *Use Cases Completed* for differing Test Granularity at Test Coverage 0.75

the argument that upon the shift from 1.0 to 0.5 the drop in cognitive load, due to a narrower development scope, was not as substantial in comparison to when the shift was made from 1.5 to 1 because the cognitive load when *Test Granularity* was 1.0 was already manageable and not burdensome and as such the drop in development scope, due to the shift in *Test Granularity* from 1.0 to 0.5, did not result in much of a reduction in the cognitive load.

Figure 4.14 shows the evolution of *Defects Missed* at *Test Coverage* value 0.75. Two interesting observations can be derived from the results of this group of simulation runs. Firstly, it can be noted that test coverage has a greater impact on TDD's performance than the test granularity. This can be generalized from TDD's worst performance at *Test Coverage* value 0.95, illustrated by the defect-curve for *GRAN 3* in Figure 4.13, being better than its best performance at *Test Coverage* value 0.75, shown by the defect-curve for *GRAN 2* in Figure 4.14. Secondly, the relative ordering of the defect-level curves has changed with the worst performance now being recorded at the highest level of granularity and the gap between the defect distribution curves at *Test Granularity* 1.0 and 1.5 having narrowed down.

With a lower level of test coverage there is a larger probability of defects lying outside the covered code. In this case it might be preferable to have tests that are more coarsely-grained as

they would help in identifying defects that would otherwise have laid on the boundaries between two tests and would not have been detected due to a lack of coverage. Consequently, one plausible explanation for the re-ordering of the granularity results is that, in comparison to the *Test Granularity* value 1.5, at the value 0.5 the drop in the number of defects generated, due to a reduced scope, is overshadowed by the number of defects escaping due the testing process due to the lack of coverage. However, at the value 1.0 the advantage brought about by the reduced scope is still predominant resulting in a lower defect level than at the value 1.5.

*Analyzing the effects of changes in Test Effort and Domain & Tool Specific Issues*

In the presented model the variable *Test Effort Ratio* impacts both the generation of defects and the detection/removal of defects. With regard to defect detection, as mentioned earlier, an increase in test effort is modeled to result in an increase in the level of test coverage. An improvement in test coverage is expected to result in more of the defects in the system being identified during test suite runs. With regard to defect generation, an increase in test effort would imply that more time is being spent on the preparation of each test. Consequently, as explained earlier, developers are expected have a better understanding of the system being developed and thus would introduce less defects during the coding or implementation of the system. Also, due to the impact of test coverage on the design quality, an increase in test effort is indirectly expected to lower the number of design-related defects. In summary, due to a drop in the number of defects that are generated and an improvement in the fraction of the defects that are detected during testing, an increase in test effort is expected to lower the overall defect count of the end product. Since the impact of test effort is very similar to the impact of experience level, for brevity purposes the results related to this variable have been omitted.

The variable *Domain & Tool Specific Issues (DTSI)*, based on the available empirical research, has been modeled to influence the quality result solely through its impact on test coverage. As mentioned earlier, *DTSI Level* has a negative impact on *Test Coverage*. Due to its impact on test coverage, a higher *DTSI Level* is expected to lower the rate of defect defection. Also, a higher

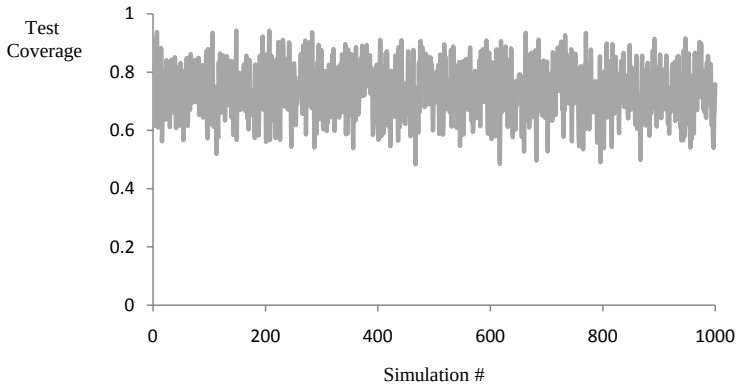



Figure 4.15: Distribution of *Test Coverage* over 1000 runs

value for this variable is expected to increase the rate of defect generation due to its indirect impact on the design quality. Though, in contrast to the *Test Effort Ratio*, a higher *DTSI Level* does not raise the defect generation rate by the same extent as it impacts only the design source of defects. Intuitively, due to a negative impact on defect detection and a positive impact on defect generation, a higher DTSI intensity will increase the defect level of the product. To avoid repitition results from simulation runs at different values of *DTSI Level* have been excluded from this analysis.

*Other Results*

The analysis on the variables *Experience Level*, *DTSI Level* and *Test Effort Ratio* revealed that the test coverage plays a major role in the evolution of defects in the system. As highlighted in the previous section, not only does the test coverage partially determine the fraction of the defects that are detected, but it also influences the number of defects that are generated. Furthermore, when analyzing results on the variable *Test Granularity*, it was found that test coverage can also influence the nature of the impact of other moderator variables. Hence, it would be interesting to see the range of values assumed by this model element when all moderator variables are varied within reasonable limits (i.e., those given in Table B.1 in Appendix B.).

Figure 4.15 shows a plot of the values assigned to the model element *Test Coverage* in the 1000 simulation runs explained in Section 4.4.2.1. It is visible that the coverage value in the simulations ranged from approximately 0.5 to 0.9. This is a contrast to the image portrayed by advocates of

TDD. They claim that by adhering to TDD's principle of only writing code to pass a failing test, the resulting test suite exhibits a high level of test coverage; however, the presented model and its simulation results conclude that the test coverage may vary significantly depending on the values of the moderator variables and in some cases may be inadequately low reducing the ability to leverage the benefits of TDD.

## 4.5 Limitations

Simulation models provide a simplistic view of reality and as such their results should not be accepted without assessing their limitations. The main limitations of the model presented in this chapter are given below.

- The most obvious limitation is that not much emphasis has been placed on the productivity aspect of this model. In particular, constant productivities have been assumed for each of the activities that make up the TDD-cycle which is rarely the case in practice.

- Another limitation is that a simlified view of design was assumed during the modeling process. This was the only option available as the impact of TDD on design still remains unclear.

- Some of the elements, in particular the *Coding Defect Generation Rate*, the *Experience Level* and the *Test Coverage*, were modeled to be constant for the duration of the simulation. In reality, the concepts represented by these elements are expected to change over the project's course; however, the factors that cause these changes are outside the scope of the model and hence this simplifying assumption has been made.

## 4.6 Discussion

This chapter presented a System Dynamics model of a TDD-based development process. Considering that TDD is predominantly an implementation-phase practice, with some effects on system design, the scope of the model was limited to analyzing the impact of the practice on these two phases. As such, the simulation process commences on the assumption that requirements have already been collected and models the development process up to the point where implementation has been completed and the system has been unit tested.

As an application of the model, we simulated the impact of changes in the values of moderator variables. Results from the simulation process helped identify/confirm some observations regarding the the impact of these variables.

With regard to task size, the simulation process affirmed that benefits from applying TDD progressively increase in magnitude, and as a result are expected to be greater in larger-scale systems. However, results from the remaining moderator variables indicated that the task size is not the only factor influencing the extent of the improvement experienced in a particular context.

With regard to experience level, in accordance with intuition that developers' experience level will determine the proficiency with which they can carry out development-related activities, the defect generation rate was found to be inversely correlated with the experience level. Although the impact of experience on the design and coding activities is fairly straightforward, a more subtle impact identified by the simulation process was that of testing experience on design-related defects. In particular, the simulation process revealed that by maintaining a higher coverage level, possibly due to a higher testing experience level, the drop in design quality is slower, which in turn would lead to less design-related defects being generated than would otherwise have been the case. Due to its impact on test coverage, in addition to defect generation, the experience level also plays a role in determining the fraction of the generated defects that are detected during test suite runs.

With regard to test granularity, the simulation process further strengthened the claim of finer-

grained TDD tests resulting in higher system quality. Though, it was found that the magnitude of the benefit achieved by a finer granularity level decreases as the tests become more and more finer-grained. Interestingly, the nature of the impact of test granularity was found to be altered at a lower level of test coverage.

With regard to the remaining variables, Test Effort and DTSI, due to their impact on test coverage, they were observed to impact both the generation and the detection of defects.

# Chapter 5

# Conclusions and Future Work

In the past decade, Agile development has been one of the prime areas of focus within software engineering research. Interest in agile practices/methodologies has partially been fueled by their widespread acceptance in industry. However, to remove misconceptions and to promote further adoption it is important to gain a deeper understanding of these practices/methodologies and validate their benefits as well as identify variables that influence their performance. This research takes a step towards this goal by providing some insight into the TDD practice.

The intent of this thesis was to build on the knowledge acquired from previous empirical research on TDD. A two-phased approach was adopted where the first phase involved a meta-analysis of studies and the second phase involved the development of a simulation model of a TDD-based development process.

The presented meta-analysis examined the impact of TDD on the quality and productivity outcome constructs. Despite considerable differences amongst the studies, some valuable insight can be gained from their collective analysis. At the high level, results from the meta-analysis were similar to those reported in earlier analyses of TDD. In particular, support was found for a small to medium sized improvement in quality but results on productivity were inconclusive. At the low level, the meta-analysis helped identify moderator variables that can be used to rationalize the variation in studies' results. Five potential moderator variables were proposed namely task size, developer experience, test granularity, test effort and domain & tool-specific issues (DTSI). With regard to task size, a positive correlation was found with the magnitude of the improvement in quality. Although no direct correlation was found between experience level and either of the out-

come constructs, too few studies were used in the analysis for this result to be considered definite. Nonetheless, the examined research did reveal that experienced developers maintain higher test coverage and refactor code more. Test granularity was found to be correlated with an improvement in productivity. No significant evidence was found of the impact of test effort, however, following intuition this variable is hypothesized to influence at least the quality construct. Similar to test effort, not much evidence on the quantitative impact of domain and tool-specific issues (DTSI) is available; however, this variable is expected to negatively impact TDD's performance as has been discussed qualitatitvely in multiple studies.

The presented SD simulation model facilitated a detailed examination of the impact of TDD on the quality construct. The model itself was a compound of three co-flows corresponding to the three major subsystems related to the evolution of defects in literature on TDD. As an application of the model, it was simulated under settings for moderator variables discussed earlier. The simulation process affirmed/provided some insight regarding the nature of the impact of the moderator variables. With regard to task size, the simulation process confirmed the claim that benefits from the application of TDD may not be immediately apparent and they progressively increase in magnitude in conjunction with system size. With regard to experience level, an interesting phenomenon that was observed was the impact of testing experience on system design. In particular, higher testing experience was found to reduce the rate at which the design deteriorates during development. This, in turn, is expected to lower the growth rate of design-related defect generation as development progresses and perhaps also improvement overall productivity by reducing the number of 'major refactoring' phases during system development. With regard to test granularity, quality improvements were found to increase as test become more fine-grained. However, simulation results also revealed that these improvements become progressively smaller in magnitude as test become more finer-grained and also, if coverage is not adequately ensured, beyond a certain level of granularity benefits might start to reverse. With regard to test effort and DTSI, due to their influence on test coverage, they were both found to significantly influence both the defect generation process

and the defect detection process. Lastly, the simulation process revealed that although TDD might be practiced, it is highly possible that the test coverage may vary significantly and its actual value is far below 100% due to the influence of the moderator variables.

To conclude this analysis, we suggest some guidelines for future research. One such route is further research into identifying and analyzing possible sources of variation in TDD's performance. Another promising avenue is investigating the effect of TDD on the initial and the continual design effort. Although multiple studies of TDD's impact on design exist, few analyze this internal construct in combination with the external constructs quality and productivity. Thirdly, differences between the impacts of test granularity and test effort could be examined in future empirical trials.

For researchers intending to conduct empirical trials some of the following considerations should be taken into account. Researchers should ensure that adequate training has been given to the TDD group beforehand and that the subjects have fully grasped the technique prior to the actual experiment. Also, it is important that the duration of the entire development process is recorded as opposed to only noting the duration of the implementation phase. Moreover, the term experience has been used broadly and it might be beneficial to distinguish between the different types of experience. Lastly, task difficulty should be reported in a more objective fashion so that tasks can be easily compared across experiments and an analysis into whether this variable is a moderator variable can be conducted.

# Appendix A

# Studies in the Meta-Analysis

The following studies were examined in the meta-analysis in Chapter 3:


T. Bhat, N. Nagappan, Evaluating the efficacy of test-driven development: industrial case studies, in: Proc. of 2006 ACM/IEEE International Symposium on Empirical Software Engineering, 2006, pp. 356-363.

G. Canfora, A. Cimitile, F. Garcia, M. Piattini, C. Visaggio, Evaluating advantages of test driven development: a controlled experiment with professionals, in: Proc. of the 2006 ACM/IEEE international symposium on Empirical software engineering, 2006, pp. 364-371.

C. Desai, D. Janzen, J. Clements, Implications of integrating test-driven development into CS1/CS2 curricula, ACM SIGCSE Bulletin 41 (1) (2009) pp. 148-152.

H. Erdogmus, M. Morisio, M. Torchiano, On the effectiveness of the testfirst approach to programming, IEEE Transactions on Software Engineering 31 (3) (2005) pp. 226-237.

T. Flohr, T. Schneider, Lessons learned from an xp experiment with students: Test-first needs more teachings, in: Product-Focused Software Process Improvement, Vol. 4034 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 305-318.

B. George, L. Williams, A structured experiment of test-driven development, Information and Software Technology 46 (5) (2004) pp.337-342.

A. Gupta, P. Jalote, An experimental evaluation of the effectiveness andefficiency of the test driven development, in: Proc. of 1st International Symposium on Empirical Software Engineering and Measurement ESEM 07, 2007, pp. 285-294.

L. Huang, M. Holcombe, Empirical investigation towards the effectiveness of Test First programming, Information and Software Technology 51 (1) (2009) 182-194.

D. Janzen, H. Saiedian, On the influence of test-driven development on software design, in: Proc. of 19th Conference on Software Engineering Education and Training, Turtle Bay, HI, 2006, pp. 141-148.

K. Lui, K. Chan, Test driven development and software process improvement in china, in: Extreme Programming and Agile Processes in Software Engineering, Vol. 3092 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2004, pp. 219-222.

L. Madeyski, Preliminary analysis of the effects of pair programming and test-driven development on the external code quality, in: Software Engineering: Evolution and Emerging Technologies, Vol. 130 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2005, pp. 113-123.

M. Muller, O. Hagner, Experiment about test-first programming, Software, IEE Proceedings - 149 (5) (2002) pp. 131-136.

N. Nagappan, E. Maximilien, T. Bhat, L. Williams, Realizing quality improvement through test driven development: results and experiences of four industrial teams, Empirical Software Engineering 13 (3) (2008) pp. 289-302.

M. Pancur, M. Ciglaric, M. Trampus, T. Vidmar, Towards empirical evaluation of test-driven development in a university environment, in: Proc. of the International Conference on Computer as a Tool EUROCON 07, 2003, pp. 83-86.

O. Slyngstad, J. Li, R. Conradi, H. Ronneberg, E. Landre, H. Wesenberg, The Impact of Test Driven Development on the Evolution of a Reusable Framework of Components: An Industrial Case Study, in: Proc. of 3rd International Conference on Software Engineering Advances, 2008, pp. 214-223.

J. Vu, N. Frojd, C. Shenkel-Therolf, D. Janzen, Evaluating Test-Driven Development in an Industry-Sponsored Capstone Project, in: Proc. of 6th International Conference on Information Technology: New Generations, 2009, pp. 229-234.

L. Williams, E. Maximilien, M. Vouk, Test-driven development as a defect-reduction practice, in: Proc. of IEEE International Symposium on Software Reliability Engineering, 2003, pp. 34-45.

S. Xu, T. Li, Evaluation of test-driven development: An academic case study, in: Software Engineering Research, Management and Applications, Vol. 253 of Studies in Computational Intelligence, Springer Berlin / Heidelberg, 2009, pp. 229-238.

S. Yenduri, L. Perkins, Impact of Using Test-Driven Development: A Case Study, Software Engineering Research and Practice (2006) pp. 126-129.

L. Zhang, S. Akifuji, K. Kawai, T. Morioka, Comparison between test driven development and waterfall development in a small-scale project,in: Extreme Programming and Agile Processes in Software Engineering, Vol. 4044 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 211-212.

# Appendix B

# Model Details

This chapter provides provides details related to the SD model presented in Chapter 4.

## B.1 The complete SD model

Figure B.1 shows the same model as Chapter 4. However, this figure shows all model elements that were previously hidden for simplicity.

## B.2 Model Equations

The differential equations used in the SD model are given below.

### B.2.1 Model Constants

```
Nominal design restore per unit=0.000165

Duration for Major Refactoring=999

DTSI Level=0.5

Experience Level=0.7

Analyzing Module Productivity=12
```

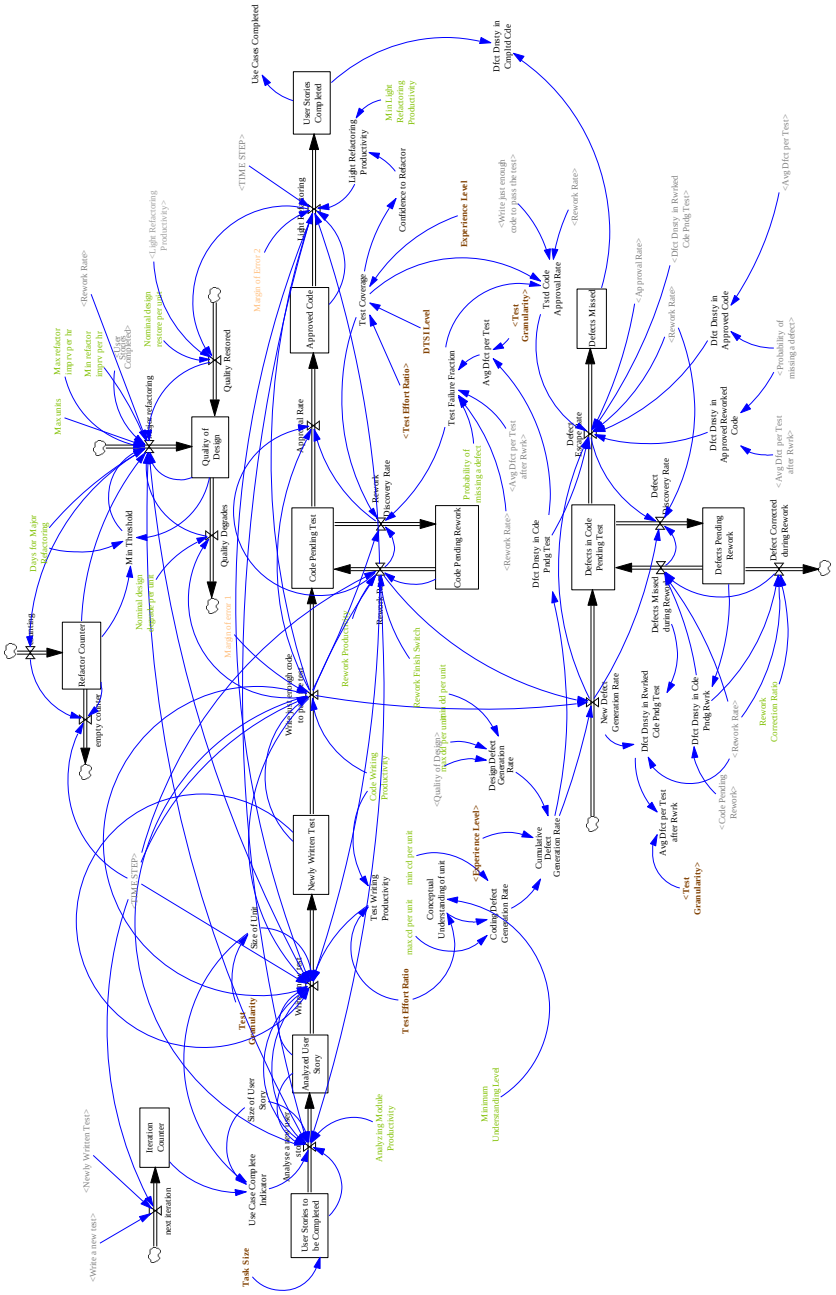



Figure B.1: The TDD SD model with hidden elements

```
Test Granularity=1.5
```

```
Minimum Understanding Level=0.5
```

Margin of Error 2=0.03

Size of User Story=12

Time Step=0.125

Test Effort Ratio=0.8

Max refactor imprv per hr=0.0005

Max units=10000

Min refactor imprv per hr=0.0001

min cd per unit=0.075

min dd per unit=0.125

Nominal design degrade per unit=0.0005

max cd per unit=0.225

max dd per unit=0.375

Task Size=3780

Min Light Refactoring Productivity=1

Probability of missing a defect=0.3

Margin of error 1=0.0001

```
Rework Finish Switch=0.001

Code Writing Productivity=0.8

Rework Productivity=8

Rework Correction Ratio=0.8
```

### B.2.2 Model Auxiliaries

```
Avg Dfct per Test=
 Dfct Dnsty in Cde Pndg Test*Test Granularity

Avg Dfct per Test after Rwrk=
 Dfct Dnsty in Rwrked Cde Pndg Test*Test Granularity

Size of Unit=Test Granularity

Test Coverage=
 ((Test Effort Ratio\^(1/1.5)) *  (1/3))
               + ((Experience Level\^(1/2.5))*  (1/3))
               + ((1-0.75*DTSI Level)*  (1/3))

Test Failure Fraction=
  IF THEN ELSE( Rework Rate>0 ,
     1-EXP(-Avg Dfct per Test after Rwrk
         *(1-Probability of missing a defect)),
     1-EXP(-Avg Dfct per Test
         *(1-Probability of missing a defect)))
```

```
Dfct Dnsty in Cde Pndg Test=
 Cumulative Defect Generation Rate

Min Threshold= SIMULTANEOUS(
IF THEN ELSE(
   INTEGER(Refactor Counter+0.03)=Days for Major Refactoring,
   Quality of Design/2,
   Min Threshold)
,0.5)

Dfct Dnsty in Cmpltd Cde=
 ZIDZ(Defects Missed,Use Cases Completed)

Dfct Dnsty in Rwrked Cde Pndg Test=
  ZIDZ(New Defect Generation Rate+Defects Missed during Rework,
     Rework Rate)

Dfct Dnsty in Approved Reworked Code=
  EXP(-(1-Probability of missing a defect)*Avg Dfct per Test after Rwrk)
  *Avg Dfct per Test after Rwrk*Probability of missing a defect

Use Case Complete Indicator=
  IF THEN ELSE(MODULO(Iteration Counter,
     INTEGER((Size of User Story/Size of Unit)+0.99))<0.01,
     1,0)

Dfct Dnsty in Approved Code=
  EXP(-(1-Probability of missing a defect)*Avg Dfct per Test)
  *Avg Dfct per Test*Probability of missing a defect
```

```
Conceptual Understanding of unit=
  Minimum Understanding Level+(1-Minimum Understanding Level)
  *Test Effort Ratio

Test Writing Productivity=(1/Test Effort Ratio)
 *Code Writing Productivity

Light Refactoring Productivity=
  Min Light Refactoring Productivity/Confidence to Refactor

Confidence to Refactor=
 Test Coverage

Dfct Dnsty in Cde Pndg Rwrk=
 ZIDZ(Defects Pending Rework, Code Pending Rework)
```

### B.2.3 Model Rates

```
Quality Restored=
 IF THEN ELSE( Light Refactoring > 0,
   Nominal design restore per unit
    *(1/Light Refactoring Productivity)*Light Refactoring
   ,0 )

Quality Degrades=
 Nominal design degrade per unit
  *Write just enough code to pass the test
```

```
Major refactoring =
 IF THEN ELSE((Quality of Design<Min Threshold
     :OR:(Refactor Counter>0.1 :AND:INTEGER(Refactor Counter+0.03)
        <=Days for Major Refactoring))
     :AND: Quality Degrades=0
   :AND: Quality Restored=0:AND:Rework Rate=0,
Max refactor imprv per hr-
   (Max refactor imprv per hr-Min refactor imprv per hr)
   *(Use Cases Completed/Max units
,0)

New Defect Generation Rate=
IF THEN ELSE( Write just enough code to pass the test > 0
   :OR: Rework Rate > 0,
  IF THEN ELSE(Write just enough code to pass the test > 0,
       Write just enough code to pass the test
        *Cumulative Dfct Generation Rate,
       Rework Rate*Cumulative Dfct Generation Rate)
  ,0)

Defect Discovery Rate=
IF THEN ELSE( New Defect Generation Rate > 0,
   IF THEN ELSE(Rework Rate > 0,
       New Defect Generation Rate+Defects Missed during Rework
            -Defect Escape Rate,
   New Defect Generation Rate-Defect Escape Rate)
   ,0)

Defect Escape Rate=
 IF THEN ELSE(New Defect Generation Rate > 0,
   IF THEN ELSE(Rework Rate > 0,
```

```
     Tstd Code Approval Rate*Dfct Dnsty in Approved Reworked Code
      +(Approval Rate-Tstd Code Approval Rate)
      *Dfct Dnsty in Rwrked Cde Pndg Test,
     Tstd Code Approval Rate*Dfct Dnsty in Approved Code
        +(Approval Rate-Tstd Code Approval Rate)
        *Dfct Dnsty in Cde Pndg Test)
   ,0)

next iteration=
 IF THEN ELSE( Newly Written Test=0
     :AND:Write a new test>0,
   (1/TIME STEP)
   ,0)

Cumulative Defect Generation Rate=
 (Design Dfct Generation Rate+Coding Dfct Generation Rate)
   *((1-0.95*Experience Level)*(1/2))

Analyse a new user story=
 SIMULTANEOUS(
   IF THEN ELSE( Analyzed User Story<Size of User Story
       :AND:Use Cases to be Completed> 0.01
       :AND: Use Case Complete Indicator =1
       :AND:Write a new test=0
       :AND:Write just enough code to pass the test=0
       :AND:Rework Rate=0
       :AND:Light Refactoring=0
       :AND:Major refactoring=0,
  IF THEN ELSE(Analyzed User Story<Size of User Story,
     Analyzing Module Productivity,0)
  ,0)
```

```
,Analyzing Module Productivity)

Light Refactoring=
IF THEN ELSE(Approved Code>0
     :AND:Write just enough code to pass the test=0
     :AND:Rework Rate=0,
   IF THEN ELSE(Approved Code>
     ((Light Refactoring Productivity*TIME STEP)+Margin of Error 2),
      Light Refactoring Productivity,
      Approved Code*(1/TIME STEP))
,0)

Coding Defect Generation Rate=
  min cd per unit+(max cd per unit-min cd per unit)
   *(1-(2*Conceptual Understanding of unit-1))

Design Defect Generation Rate=
  min dd per unit+(max dd per unit-min dd per unit)
  *(1-Quality of Design)

counting=IF THEN ELSE( Major refactoring>0, 1,0)

empty counter=
  IF THEN ELSE(Refactor Counter>0:AND:counting=0,
    Refactor Counter*(1/TIME STEP)
    ,0)

Write a new test=
SIMULTANEOUS(
  IF THEN ELSE(Analyse a new user story=0
    :AND: Analyzed User Story > 0
```

```
    :AND:Write just enough code to pass the test=0
    :AND:Rework Rate=0
    :AND:Light Refactoring=0
    :AND:Major refactoring=0,
    IF THEN ELSE( Size of Unit - Newly Written Test
      > (Test Writing Productivity*TIME STEP),
      IF THEN ELSE((Test Writing Productivity*TIME STEP)
          >Analyzed User Story,Analyzed User Story
              *(1/TIME STEP),Test Writing Productivity),
        IF THEN ELSE((Size of Unit - Newly Written Test)
                >Analyzed User Story,
           Analyzed User Story *(1/TIME STEP),
         Size of Unit - Newly Written Test)*(1/TIME STEP)))
    ,0)
  ,0)

Approval Rate=
IF THEN ELSE Write just enough code to pass the test>0
   :OR: Rework Rate > 0,
   IF THEN ELSE(Write just enough code to pass the test>0,
     Write just enough code to pass the test-Rework Discovery Rate,
    Rework Rate-Rework Discovery Rate)
,0)



Rework Discovery Rate=
IF THEN ELSE( Write just enough code to pass the test>0
   :OR: Rework Rate > 0,
   IF THEN ELSE(Write just enough code to pass the test>0,
    (Test Failure Fraction*Test Coverage)
        *Write just enough code to pass the test,
```

```
    (Test Failure Fraction*Test Coverage)*Rework Rate)
,0)

Write just enough code to pass the test=
SIMULTANEOUS(
   IF THEN ELSE(Newly Written Test>0
    :AND:Write a new test=0,
    IF THEN ELSE(Newly Written Test
          >((Code Writing Productivity*TIME STEP)+Margin of error 1),
       Code Writing Productivity,
       Newly Written Test*(1/TIME STEP))
   ,0)
 ,0)

Defect Corrected during Rework=
  IF THEN ELSE(Rework Rate>0,
   Rework Rate*
     MIN( Dfct Dnsty in Cde Pndg Rwrk,Rework Correction Ratio)
   ,0)

Defects Missed during Rework=
IF THEN ELSE(Rework Rate>0,
     Rework Rate*
       Dfct Dnsty in Cde Pndg Rwrk-Defect Corrected during Rework
   ,0)

Tstd Code Approval Rate=
IF THEN ELSE( Write just enough code to pass the test>0
  :OR: Rework Rate > 0,
  IF THEN ELSE(Write just enough code to pass the test>0,
    (1-Test Failure Fraction)*Test Coverage
```

```
        *Write just enough code to pass the test,
    (1-Test Failure Fraction)*Test Coverage
         *Rework Rate)
,0)

Rework Rate=
IF THEN ELSE(Code Pending Rework>Rework Finish Switch
  :AND:Write just enough code to pass the test=0 ,
  IF THEN ELSE(Code Pending Rework>(Rework Productivity*TIME STEP)
      ,Rework Productivity,
      Code Pending Rework*(1/TIME STEP))
,0)
```

### B.2.4 Model Levels

```
Quality of Design=
 INTEG (Major refactoring+Quality Restored-Quality Degrades,1)

Iteration Counter=
 INTEG (next iteration, 0)

Use Cases to be Completed=
 INTEG (-Analyse a new user story,Task Size)

Refactor Counter=
 INTEG (counting-empty counter,0)

Use Cases Completed=
 INTEG (Light Refactoring,0)
```

```
Approved Code=
 INTEG (Approval Rate-Light Refactoring,0)


Defects Pending Rework=
 INTEG (Defect Discovery Rate-Defects Missed during Rework
       -Defect Corrected during Rework,0)


Code Pending Rework=
 INTEG (Rework Discovery Rate-Rework Rate,0)


Defects Missed=
 INTEG (Defect Escape Rate,0)


Code Pending Test=
 INTEG (Write just enough code to pass the test+Rework Rate
        -Rework Discovery Rate-Approval Rate,0)


Defects in Code Pending Test=
 INTEG (New Defect Generation Rate+Defects Missed during Rework
        -Defect Discovery Rate-Defect Escape Rate,0)


Newly Written Test=
 INTEG (Write a new test-Write just enough code to pass the test,0)


Analyzed User Story=
 INTEG (Analyse a new user story-Write a new test,0)
```

## B.3 Sensitivity Distributions

The distributions assumed by the moderator variables for the 1000 simulation runs in Section 3.4.2.1 and the beginning of Section 3.4.2.2 are given in Table B.1 below.

Table B.1: Sensitivity Distributions

| *Variable* | *Distribution* | *Mean* | *Std. Dev.* | *Min Value* | *Max Value* |
|---|---|---|---|---|---|
| Test Granularity | Normal | 1 | 0.5 | 0.5 | 1.5 |
| Experience Level | Normal | 0.7 | 0.3 | 0.4 | 1 |
| Test Effort Ratio | Normal | 0.6 | 0.2 | 0.2 | 1 |
| DTSI Level | Uniform | - | - | 0 | 1 |

# Bibliography


[1] T. Abdel-Hamid and S. Madnick. *Software project dynamics: an integrated approach*. Prentice-Hall, Inc., 1991.

[2] P. Abrahamsson, A. Hanhineva, and J. Jäälinoja. Improving business agility through technical solutions: A case study on test-driven development in mobile software development. In *IFIP TC8 WG 8.6 International Working Conference*, pages 227–243, 2005.

[3] S. Ambler. Test-driven development of relational databases. *IEEE Software*, 24(3):37–43, 2007.

[4] D. Astels. *Test driven development: A practical guide*. Prentice-Hall, 2003.

[5] K. Beck. *Test-driven development: by example*. Prentice-Hall, 2003.

[6] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2004.

[7] K. Beck, M. Beedle, A. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for agile software development. 2001. retrieved from `http://agilemanifesto.org/`.

[8] T. Bhat and N. Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. In *Proc. of 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 356–363, 2006.

[9] B. Boehm. Developing small-scale application software products: Some experimental results. *Software World*, 12(1):2–8, 1981.

[10] B. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[11] M. Borenstein, L. V. Hedges, J. P. Higgins, and H. R. Rothstein. A basic introduction to fixed-effect and random-effects models for meta-analysis. *Research Synthesis Methods*, 1(2):97–111, 2010.

[12] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. Visaggio. Evaluating advantages of test driven development: a controlled experiment with professionals. In *Proc. of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 364–371, 2006.

[13] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. Visaggio. Productivity of test driven development: A controlled experiment with professionals. In *7th International Conference on Product-Focused Software Process Improvement PROFES 2006*, pages 383–388, 2006.

[14] L. Cao, B. Ramesh, and T. Abdel-Hamid. Modeling dynamics in agile software development. *ACM Transactions on Management Information Systems*, 1:1–26, 2010.

[15] A. Cau, G. Concas, M. Melis, and I. Turnu. Evaluate xp effectiveness using simulation modeling. In *Extreme Programming and Agile Processes in Software Engineering*, volume 3556 of *LNCS*, pages 1142–1144. Springer, 2005.

[16] A. Causevic, D. Sundmark, and S. Punnekkat. Factors limiting industrial adoption of test driven development: A systematic review. In *Proc. of 4th Intl. Conf. on Software Testing, Verification and Validation (ICST 2011)*, pages 337–346, 2011.

[17] M. Cohn. *User stories applied: for agile software development*. Addison-Wesley, 2004.

[18] C. Desai, D. Janzen, and J. Clements. Implications of integrating test-driven development into CS1/CS2 curricula. *ACM SIGCSE Bulletin*, 41(1):148–152, 2009.

[19] T. Dogsa and D. Batic. The effectiveness of test-driven development: an industrial case study. *Software Quality Journal*, pages 1–19, 2011.

[20] T. Dyba and T. Dingsoyr. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9-10):833–859, 2008.

[21] S. Edwards. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In *Proc. of the International Conference on Education and Information Systems: Technologies and Applications EISTA*, 2003.

[22] S. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. *ACM SIGCSE Bulletin*, 36(1):26–30, 2004.

[23] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–237, 2005.

[24] T. Flohr and T. Schneider. Lessons learned from an xp experiment with students: Test-first needs more teachings. In *Product-Focused Software Process Improvement*, volume 4034 of *Lecture Notes in Computer Science*, pages 305–318. Springer Berlin / Heidelberg, 2006.

[25] J. Forrester. The beginning of system dynamics. *McKinsey Quarterly*, pages 4–17, 1995.

[26] J. W. Forrester and P. M. Senge. Test for Building Confidence in System Dynamics Models. *TIMS studies in the management sciences*, 14:209–228, 1980.

[27] K. Forsberg and H. Mooz. The relationship of system engineering to the project cycle. *Center for Systems Management*, 9:11, 1994.

[28] B. George and L. Williams. An initial investigation of test driven development in industry. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 1135–1139, 2003.

[29] B. George and L. Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337–342, 2004.

[30] R. Grissom and J. Kim. *Effect sizes for research: a broad practical approach*. Lawrence Erlbaum Associates, 2005.

[31] A. Gupta and P. Jalote. An experimental evaluation of the effectiveness and efficiency of the test driven development. In *Proc. of 1st International Symposium on Empirical Software Engineering and Measurement ESEM 07*, pages 285–294, 2007.

[32] J. Hannay, T. Dybå, E. Arisholm, and D. Sjøberg. The effectiveness of pair programming: A meta-analysis. *Information and Software Technology*, 51(7):1110–1122, 2009.

[33] T. Hellmann, A. Hosseini-Khayat, and F. Maurer. Supporting test-driven development of graphical user interfaces using agile interaction design. In *Proc. of 3rd International Conference on Software Testing, Verification, and Validation Workshops*, pages 444–447, 2010.

[34] P. Hodgetts. Refactoring the development process: Experiences with the incremental adoption of agile practices. In *Proc. of Agile Development Conference ADC'04*, pages 106–113, 2004.

[35] A. Höfer and M. Philipp. An empirical study on the tdd conformance of novice and expert pair programmers. In *Agile Processes in Software Engineering and Extreme Programming*, pages 33–42. Springer Berlin Heidelberg, 2009.

[36] D. Houston, D. Buettner, and M. Hecht. Defectivity profiling with dynamic coqualmo: An explication and product quality retrospective. *Journal of Software Maintenance and Evolution: Research and Practice*, 2010.

[37] L. Huang and M. Holcombe. Empirical investigation towards the effectiveness of test first programming. *Information and Software Technology*, 51(1):182–194, 2009.

[38] T. Huedo-Medina, J. Sánchez-Meca, F. Marín-Martínez, and J. Botella. Assessing heterogeneity in meta-analysis: $Q$ statistic or $I^2$ index? *Psychological Methods*, 11(2):193–206, 2006.

[39] B. S. Inc., 2006. `http://www.meta-analysis.com/`.

[40] D. Janzen and H. Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *IEEE Computer*, 38(9):43–50, 2005.

[41] D. Janzen and H. Saiedian. Does test-driven development really improve software design quality. *IEEE Software*, 25(2):77–84, 2008.

[42] D. Janzen, C. S. Turner, and H. Saiedian. Empirical software engineering in industry short courses. In *20th Conference on Software Engineering Education and Training*, pages 89–96, Dublin, Ireland, 2007.

[43] T. Jones. Measuring programming quality and productivity. *IBM Systems Journal*, 17(1):39–63, 1978.

[44] V. Kampenes, T. Dybå, J. Hannay, and D. Sjøberg. A systematic review of effect size in software engineering experiments. *Information and Software Technology*, 49(11–12):1073–1086, 2007.

[45] M. Kellner, R. Madachy, and D. Raffo. Software process simulation modeling: why? what? how? *Journal of Systems and Software*, 46(2/3):91–105, 1999.

[46] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. *Engineering*, 2(EBSE 2007–001), 2007.

[47] S. Kollanus. Test-driven development – still a promising approach? In *Proc. of 7th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 403–408, oct 2010.

[48] S. Kollanus and V. Isomöttönen. Understanding TDD in academic environment: experiences from two experiments. In *2008 Koli Calling International Conference on Computing Education Research*, pages 25–31, Finland, 2008.

[49] S. Kuppuswami, K. Vivekanandan, P. Ramaswamy, and P. Rodrigues. The effects of individual xp practices on software development effort. *ACM SIGSOFT Software Engineering Notes*, 28(6):6, 2003.

[50] P. Lakey. A hybrid software process simulation model for project management. In *Proc. of the 6th Process Simulation Modeling Workshop (ProSim 2003)*, 2003.

[51] C. Larman and V. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, 2003.

[52] M. Lipsey and D. Wilson. *Practical Meta-Analysis*. Sage Publications, Inc., Thousand Oaks, CA, 2001.

[53] R. Madachy. *Software process dynamics*. Wiley-IEEE Press, 2008.

[54] L. Madeyski. Preliminary analysis of the effects of pair programming and test-driven development on the external code quality. In *Software Engineering: Evolution and Emerging Technologies*, volume 130 of *Frontiers in Artificial Intelligence and Applications*, pages 113–123. IOS Press, 2005.

[55] L. Madeyski. The impact of pair programming and test-driven development on package dependencies in object-oriented design - an experiment. In *7th International Conference on Product-Focused Software Process Improvement PROFES 2006*, pages 278–289, Amsterdam, The Netherlands, 2006.

[56] L. Madeyski. Is external code quality correlated with programming experience or feelgood

factor? In *Extreme Programming and Agile Processes in Software Engineering*, pages 65–74. Springer, 2006.

[57] L. Madeyski. The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. *Information and Software Technology*, 52(2):169–184, 2010.

[58] L. Madeyski and Ł. Szała. The impact of test-driven development on software development productivity - an empirical study. In *14th European Conference on Software Process Improvement*, pages 200–211, 2007.

[59] R. Martin and D. Raffo. A model of the software development process using both continuous and discrete models. *Software Process: Improvement and Practice*, 5(2-3):147–157, 2000.

[60] E. Maximilien and L. Williams. Assessing test-driven development at ibm. In *Proc. 25th International Conference on Software Engineering*, pages 564–569, 2003.

[61] M. Melis. *A software process simulation model of extreme programming*. PhD thesis, Universita' di Cagliari, Italy, 2005.

[62] G. Melnik and F. Maurer. A cross-program investigation of students' perceptions of agile methods. In *Proceedings 27th International Conference on Software Engineering*, pages 481–488, 2005.

[63] V. Misic, H. Gevaert, and M. Rennie. Extreme dynamics: towards a system dynamics model of the extreme programming software development process. In *IEE Seminar Digests*, volume 2004, pages 237–242, 2004.

[64] M. Müller and A. Höfer. The effect of experience on the test-driven development process. *Empirical Software Engineering*, 12:593–615, 2007.

[65] M. M. Müller and O. Hagner. Experiment about test-first programming. *IEE Proceedings – Software*, 149(5):131–136, oct 2002.

[66] N. Nagappan, E. Maximilien, T. Bhat, and L. Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, 13(3):289–302, 2008.

[67] W. Opdyke. Refactoring object-oriented software to support evolution and reuse. In *Proc. of the 7 th Annual Workshop on Institutionalizing Software Reuse*, 1995.

[68] R. Ou. Test-driven database development: A practical guide. In *Extreme Programming and Agile Methods - XP/Agile Universe*, volume 2753 of *Lecture Notes in Computer Science*, pages 82–90. Springer, 2003.

[69] S. Palmer and J. Felsing. *A Practical Guide to Feature-Driven Development*. The Coad series. Prentice-Hall, 2002.

[70] M. Pancur and M. Ciglaric. Impact of test-driven development on productivity, code and tests: a controlled experiment. *Information and Software Technology*, 53(6):557–573, 2011.

[71] M. Pancur, M. Ciglaric, M. Trampus, and T. Vidmar. Towards empirical evaluation of test-driven development in a university environment. In *Proc. of the International Conference on Computer as a Tool EUROCON 07*, pages 83–86, 2003.

[72] H. Rahmandad and K. Hu. Modeling the rework cycle: capturing multiple defects per task. *System Dynamics Review*, 26(4):291–315, 2010.

[73] A. Rendell. Effective and pragmatic test driven development. In *Agile 2008*, pages 298–303, 2008.

[74] W. Royce. Managing the development of large software systems. In *Proc. of IEEE Wescon*, volume 26, 1970.

[75] A. Ruiz and Y. W. Price. Test-driven GUI development with TestNG and Abbot. *IEEE Software*, 24(3):51–57, 2007.

[76] J. Sanchez, L. Williams, and E. Maximilien. On the sustained use of a test-driven development practice at IBM. In *AGILE 2007*, pages 5–14, Washington, DC, 2007.

[77] P. Schuh. *Integrating Agile Development in the Real World*. Programming Series. Charles River Media, 2004.

[78] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Series in Agile software development. Pearson Education, 2008.

[79] J. Shore and S. Warden. *The Art of Agile Development*. O'Reilly Inc., 2007.

[80] F. Shull, G. Melnik, B. Turhan, L. Layman, M. Diep, and H. Erdogmus. What do we know about test-driven development? *IEEE Software*, 27(6):16–19, 2010.

[81] M. Siniaalto. The impact of test-driven development on design quality. *Agile Software Development of Embedded Systems*, Public Deliverables(D.5.2.10 v1.0), 2006. retrieved from `http://www.agile-itea.org/public/deliverables/ITEA-AGILE-D5.2.10_v1.0.pdf`.

[82] M. Siniaalto. Test driven development: Empirical body of evidence. Agile Deliverable D.2.7, Information Technology for European Advancement, Eindhoven, The Netherlands, 2006.

[83] M. Siniaalto and P. Abrahamsson. A comparative case study on the impact of test-driven development on program design and test coverage. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007.

[84] M. Siniaalto and P. Abrahamsson. Does test-driven development improve the program code? alarming results from a comparative case study. In *Second IFIP TC 2 Central and East*

*European Conference on Software Engineering Techniques, CEE-SET 2007*, pages 143–156, Poznan, Poland, 2007.

[85] O. Slyngstad, J. Li, R. Conradi, H. Ronneberg, E. Landre, and H. Wesenberg. The impact of test driven development on the evolution of a reusable framework of components: An industrial case study. In *Proc. of 3rd International Conference on Software Engineering Advances*, pages 214–223, 2008.

[86] J. Sterman. *Business dynamics: systems thinking and modeling for a complex world*, volume 53. Irwin McGraw-Hill, 2000.

[87] T. Thayer, M. Lipow, and E. Nelson. *Software reliability: a study of large project reality*. North-Holland Publication Company, 1978.

[88] B. Turhan, L. Layman, M. Diep, H. Erdogmus, and F. Shull. How effective is test driven development? In *Making Software What Really Works, and Why We Believe It*, pages 207–219. O'Reilly Media, 2010.

[89] J. Vu, N. Frojd, C. Shenkel-Therolf, and D. Janzen. Evaluating test-driven development in an industry-sponsored capstone project. In *Proc. of 6th International Conference on Information Technology: New Generations*, pages 229–234, 2009.

[90] P. Wernick and T. Hall. The impact of using pair programming on system evolution a simulation-based study. In *Proc. of 20th IEEE Intl. Conf. on Software Maintenance*, pages 422–426, 2004.

[91] L. Williams, E. Maximilien, and M. Vouk. Test-driven development as a defect-reduction practice. In *Proc. of IEEE International Symposium on Software Reliability Engineering*, pages 34–45, 2003.

[92] R. Wirfs-Brock. Designing with an agile attitude. *IEEE Software*, 26(2):68–69, 2009.

[93] M. Wu and H. Yan. Simulation in software engineering with system dynamics: A case study. *Journal of Software*, 4(10):1127–1135, 2009.

[94] S. Xu and T. Li. Evaluation of test-driven development: An academic case study. In *Software Engineering Research, Management and Applications*, volume 253 of *Studies in Computational Intelligence*, pages 229–238. Springer, 2009.

[95] R. Ynchausti. Integrating unit testing into a software development teams process. In *2nd International Conference Extreme Programming and Flexible Processes in Software Engineering*, pages 79–83, 2001.

[96] Y. Yong and B. Zhou. Evaluating extreme programming effect through system dynamics modeling. In *Proc. of Intl. Conf. on Computational Intelligence and Software Engineering (CiSE 2009)*, pages 1–4, 2009.