QUALITY OF TEST DESIGN IN TEST DRIVEN DEVELOPMENT

Adnan Čaušević

2013
QUALITY OF TEST DESIGN IN TEST DRIVEN DEVELOPMENT

Adnan Čaušević

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid Akademin för innovation, design och teknik kommer att offentligen förvaras tisdagen den 11 juni 2013, 09.15 i Gamma, Högskoleplan 1, Västerås.

Fakultetsopponent: Assistant Professor Henry Muccini, L’Aquila University
Abstract

One of the most emphasised software testing activities in an Agile environment is the usage of the Test Driven Development (TDD) approach. TDD is a development activity where test cases are created by developers before writing the code, and all for the purpose of guiding the actual development process. In other words, test cases created when following TDD could be considered as a by-product of software development. However, TDD is not fully adopted by the industry, as indicated by respondents from our industrial survey who pointed out that TDD is the most preferred but least practised activity.

Our further research identified seven potentially limiting factors for industrial adoption of TDD, out of which one of the prominent factor was lack of developers’ testing skills. We subsequently defined and categorised appropriate quality attributes which describe the quality of test case design when following TDD. Through a number of empirical studies, we have clearly established the effect of “positive test bias”, where the participants focused mainly on the functionality while generating test cases. In other words, there existed less number of “negative test cases” exercising the system beyond the specified functionality, which is an important requirement for high reliability systems. On an average, in our studies, around 70% of test cases created by the participants were positive while only 30% were negative. However, when measuring defect detecting ability of those sets of test cases, an opposite ratio was observed. Defect detecting ability of negative test cases were above 70% while positive test cases contributed only by 30%.

We propose a TDD concept as an approach for achieving higher quality testing in TDD by using combinations of quality improvement aspects and test design techniques to facilitate consideration of unspecified requirements during the development to a higher extent and thus minimise the impact of potentially inherent positive test bias in TDD. This way developers do not necessarily focus only on verifying functionality, but they can as well increase security, robustness, performance and many other quality improvement aspects for the given software product. An additional empirical study, evaluating this method, showed a noticeable improvement in the quality of test cases created by developers utilising TDD concept. Our research findings are expected to pave way for further enhancements to the way of performing TDD, eventually resulting in better adoption of it by the industry.
To my family,
for their love and support
Populärvetenskaplig sammanfattning


En av de mer välkända agila teknikerna är testdriven utveckling (TDD), där testfall skapas av utvecklare innan dess att koden skrivs - allt i syfte att styra eller driva utvecklingsprocessen. Testfall skapade vid användning av TDD kan betraktas som en biprodukt av programvaruutvecklingen. Ett antal vetenskapliga studier har utförts i syfte att undersöka de påstådda förbättringarna av kodkvalitet vid TDD, medan mycket få studier fokuserat på att undersöka eventuella förändringar i kvaliteten i de testfall som metoden producerar.

Denna avhandling undersöker kvaliteten hos testfall producerade vid testdriven utveckling. I våra studier har vi tydligt märkt en tendens hos utvecklare att fokusera på positiva testfall, dvs. testfall som undersöker programvarans beteende i ett tänkt normalfall. Runt 70% av de testfall som skapades i våra studier var positiva testfall, medan endast 30% var negativa. I kvaliteten hos testfallen observerades dock ett motsatt förhållande. Negativa testfall bidrog med över 70% av den totala testkvaliteten, medan positiva testfall bidrog endast med 30%.

Baserat på resultaten i våra studier föreslår vi TDD^HQ, en metod för att uppnå högre testfall av högre kvalitet i testdriven utveckling genom kombination av traditionell testdriven utveckling och testdesign tekniker. Genom
TDDHQ utökas TDD till att inte enbart kontrollera funktionalitet, utan också säkerhet, robusthet, prestanda och andra kvalitetsaspekter. En preliminär utvärdering resulterade i 17% bättre kvalitet i testfallen hos utveckare som använder TDDHQ jämfört med utvecklare som använder traditionell TDD. Våra forskningsresultat förväntas bana väg för ytterligare förbättringar i sättet att utföra TDD, vilket resulterar så småningom kan medföra utökad användning av test-driven utveckling i industrin.
Abstract

One of the most emphasised software testing activities in an Agile environment is the usage of the Test Driven Development (TDD) approach. TDD is a development activity where test cases are created by developers before writing the code, and all for the purpose of guiding the actual development process. In other words, test cases created when following TDD could be considered as a by-product of software development. However, TDD is not fully adopted by the industry, as indicated by respondents from our industrial survey who pointed out that TDD is the most preferred but least practised activity.

Our further research identified seven potentially limiting factors for industrial adoption of TDD, out of which one of the prominent factor was lack of developers’ testing skills. We subsequently defined and categorised appropriate quality attributes which describe the quality of test case design when following TDD. Through a number of empirical studies, we have clearly established the effect of “positive test bias”, where the participants focused mainly on the functionality while generating test cases. In other words, there existed less number of “negative test cases” exercising the system beyond the specified functionality, which is an important requirement for high reliability systems. On an average, in our studies, around 70% of test cases created by the participants were positive while only 30% were negative. However, when measuring defect detecting ability of those sets of test cases, an opposite ratio was observed. Defect detecting ability of negative test cases were above 70% while positive test cases contributed only by 30%.

We propose a TDD\textsuperscript{HQ} concept as an approach for achieving higher quality testing in TDD by using combinations of quality improvement aspects and test design techniques to facilitate consideration of unspecified requirements during the development to a higher extent and thus minimise the impact of potentially inherent positive test bias in TDD. This way developers do not necessarily focus only on verifying functionality, but they can as well increase se-
curity, robustness, performance and many other quality improvement aspects for the given software product. An additional empirical study, evaluating this method, showed a noticeable improvement in the quality of test cases created by developers utilising TDDHQ concept. Our research findings are expected to pave way for further enhancements to the way of performing TDD, eventually resulting in better adoption of it by the industry.
Acknowledgements

I am always inspired by people’s willingness to share their knowledge and very thankful when being taught something new. This is the reason I would like to thanks to many who contributed to the skill set I currently have, from playing accordion to writing scientific publications and everything in-between. However, here I would like to put more focus on the people who greatly helped in making this thesis possible.

A huge thanks goes to my main supervisor Sasikumar Punnekkat, who unselfishly devoted his time and patience to make me not just a better researcher but also a better person. His guidance and experience helped me to move forward when things in my perspective seemed unachievable. Thank you for showing me how to change the perspective and believe more in myself. I am also very thankful to my co-supervisor Daniel Sundmark for often checking how things are going on and making sure that everything is under control. Thank you for great discussions, critical view on my work and being such a fantastic support. My second co-supervisor Ivica Crnković is one of the most responsible persons for my PhD enrolment. Thank you so much for all your effort in making this journey possible in the first place.

During my PhD, I had an opportunity to meet, discuss and collaborate with many professors and senior researchers who directly or indirectly helped me to improve my work. For this I am thankful to Hans Hansson, Damir Isović, Radu Dobrin, Paul Pettersson, Abdulkadir Sajeev, Rikard Land, Frank Lüders, Rakesh Shukla, Iva Krasteva, Sigrid Eldh, Björn Lisper, Stig Larsson, Jakob Axelsson, Kristina Lundqvist, Cristina Seceleau, Thomas Nolte, Dag Nyström, Jan Carlson, Magnus Larsson, Heinz Schmidt, Antonio Cicchetti, Gordana Dodig-Crnković, Mikael Sjödin, Daniel Flemström, and Tiberiu Seceleau. In addition, I would like to thanks Gunnar, Susanne, Carola, Jenny, Anna, Malin, Sofia, and Ingrid for making administrative challenges much easier for me.
Another great thing about PhD is having an opportunity to meet other fellow PhD students. I was sharing my office with some really great roommates during these few years: Abhilash, Andreas, Jiale, Hüseyin, JK, Stefan, Kathrin, Etienne, Aleksandar, and Viji. Thank you guys for being silent, but also cheerful and always ready for a small talk. Outside of my office I had many interesting discussions during the coffee breaks, travels or other social events. For that I am thankful to Ana, Andreas, Aneta, Barbara, Batu, Branka, Eduard, Farhang, Federico, Fredrik, Gabriel, Giacomo, Hang, Hongyu, Irfan, Jagadish, Josip, Juraj, Leo, Luka, Mehrdad, Meng, Mikael, Mohammad, Moris, Nikola, Nima, Omar, Rafia, Raluca, Séverine, Saad, Sara(s), Stefan, Svetlana, and Yue.

Family was and still is an inexhaustible source of motivation for me. I am very thankful to my parents Zuhdija and Šefika as well as to my sister Azra, for their unconditional support and love through all of my life. I would like to thank to my beloved wife Aida for supporting me and for being such a great life’s companion. And last, but most certainly not the least, I would like to thank to my daughter Alina for making me a complete person and making my life journey even more exciting and rewording.

Adnan Čaušević

Västerås, June 11, 2013
List of Publications

Papers Included in the Thesis


**Paper B** *Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review*, Adnan Čaušević, Daniel Sundmark and Sasikumar Punnekkat, In proceedings of the International Conference on Software Testing (ICST), Berlin, Germany, March 2011


**Paper D** *Quality of Testing in Test Driven Development*, Adnan Čaušević, Sasikumar Punnekkat and Daniel Sundmark, International Conference on the Quality of Information and Communications Technology (QUATIC), Lisbon, Portugal, September 2012

---

1The included articles are reformatted to comply with the thesis layout specifications
**Paper E**  *Effects of Negative Testing on TDD: An Industrial Experiment*, Adnan Čaušević, Rakesh Shukla, Sasikumar Punnekkat and Daniel Sundmark, In proceedings of the International Conference on Agile Processes in Software Engineering and Extreme Programming (XP), Vienna, Austria, June 2013

**Paper F**  *TDDHQ: Achieving Higher Quality Testing in Test Driven Development*, Adnan Čaušević, Sasikumar Punnekkat and Daniel Sundmark, In submission
Other relevant publications

Conferences, Workshops and Poster Sessions


- Redefining the role of testers in organisational transition to agile methodologies, Adnan Čaušević, A.S.M. Sajeev and Sasikumar Punnekatt, International Conference on Software, Services & Semantic Technologies (S3T), Sofia, Bulgaria, October, 2009


Technical Reports

## Contents

### I Thesis

#### 1 Introduction

1.1 Motivation and Problem Description ........................................... 27  
1.2 Outline of thesis ................................................................. 27

#### 2 Background

2.1 Agile Development ................................................................. 29  
2.2 Software Testing ................................................................. 31  
2.3 Test-driven development .......................................................... 32  
2.4 Quality of Testing ................................................................. 33  
  2.4.1 Code Coverage ................................................................. 33  
  2.4.2 Mutation Testing ............................................................... 35  
2.5 Negative Testing ................................................................. 36

#### 3 Research Design

3.1 Research Objective ............................................................... 37  
3.2 Research Questions ............................................................... 38  
3.3 Research Methodology ........................................................... 38

#### 4 Research Contribution

4.1 Main Contributions ............................................................... 39  
  4.1.1 Identification of Limiting Factors in TDD Adoption ............... 39  
  4.1.2 Methods for Estimation of Testing Quality ......................... 42  
  4.1.3 TDD\textsuperscript{HQ} - Higher Quality Testing in TDD .......... 44  
4.2 Individual Papers’ Contributions ............................................ 46  
  4.2.1 Paper A ........................................................................... 46  
  4.2.2 Paper B ........................................................................... 47
### 5 Related Work

5.1 General Studies on TDD

5.1.1 Studies on Benefits of TDD

5.1.2 Studies on Quality of Produced Code

5.1.3 Studies on Productivity Improvements with TDD

5.1.4 Studies on Impact of Experience in TDD

5.2 Studies on Quality of Testing in TDD

### 6 Conclusions and Future Work

### Bibliography

### II Included Papers

7 Paper A: An Industrial Survey on Contemporary Aspects of Software Testing

7.1 Introduction

7.2 Research Method

7.2.1 Categorization of Respondents

7.2.2 Question Selection

7.2.3 Scales Used for Answers

7.3 Testing Practices and Preferences

7.3.1 Agile vs. Non-Agile

7.3.2 Distributed vs. Non-distributed

7.3.3 Domain

7.3.4 Safety-criticality

7.3.5 Testers vs. Non-Testers

7.4 Techniques and Tools

7.5 Satisfaction of Current Practice

7.5.1 Satisfaction within Different Categories of Respondents

7.5.2 Satisfaction with Particular Testing Practices

7.6 Conclusion

7.7 Acknowledgments
8 Paper B:  
Factors Limiting Industrial Adoption of Test Driven Development:  
A Systematic Review 93  
8.1 Introduction 95  
8.2 Research Method 96  
8.2.1 Search Process 96  
8.2.2 Paper Exclusion Process 97  
8.2.3 Data Extraction Process 98  
8.2.4 Data Synthesis 99  
8.3 Results and Analysis 99  
8.3.1 Empirical Studies of TDD 99  
8.3.2 Reported Effects of and on TDD 102  
8.3.3 Factors Limiting Industrial Adoption of TDD 103  
8.4 Discussion 109  
8.4.1 Threats to Validity 109  
8.4.2 Implications for Research 110  
8.4.3 Implications for Industry 112  
8.5 Conclusion 112  
8.6 Acknowledgments 113  
Bibliography 115  

9 Paper C:  
Impact of Test Design Technique Knowledge on Test Driven Development: A Controlled Experiment 123  
9.1 Motivation 125  
9.1.1 Problem Statement 125  
9.1.2 Research Objective 125  
9.1.3 Context 126  
9.1.4 Paper Outline 126  
9.2 Related Work 126  
9.2.1 TDD and testing knowledge 126  
9.2.2 Experiments in TDD 127  
9.3 Experimental Design 127  
9.3.1 Goals, Hypotheses, Parameters, and Variables 127  
9.3.2 Experiment Design 131  
9.3.3 Subjects 132  
9.3.4 Objects 132
11 Paper E: Effects of Negative Testing on TDD: An Industrial Experiment
11.1 Introduction ........................................ 173
11.2 Related Work ....................................... 174
11.3 Methodology and Study Design ...................... 175
  11.3.1 Study design .................................. 178
11.4 Experiment Design .................................. 179
11.5 Execution .......................................... 181
11.6 Analysis ........................................... 183
11.7 Interpretation ...................................... 184
  11.7.1 Evaluation of Results and Implications ........ 184
  11.7.2 Threats to Validity - Reservations .............. 186
11.8 Conclusions and Future Work ....................... 187
Bibliography ............................................. 189

12 Paper F: TDDHQ: Achieving Higher Quality Testing in Test Driven Development
12.1 Introduction ........................................ 195
12.2 Research Background ................................ 196
12.3 TDDHQ - Higher Quality Testing in TDD ........... 199
  12.3.1 Combinations of A and B ...................... 202
  12.3.2 Expectations and Threats ....................... 202
12.4 Evaluation of Proposed Approach ................... 203
  12.4.1 Experiment Design ............................. 204
  12.4.2 Execution ................................... 206
  12.4.3 Analysis ................................... 206
  12.4.4 Validity Threats .............................. 210
12.5 Related Work ...................................... 210
12.6 Conclusions and Future Work ....................... 212
Bibliography ............................................. 215
List of Figures

2.1 A sprint overview in the Scrum process ... 31
2.2 Test-driven development practice overview ... 33
2.3 Quadrants of relations between quality of code and tests ... 34

3.1 Flow of Three High-level Generic Questions ... 38

4.1 TDDHQ - Higher Quality Testing in Test Driven Development ... 45

9.1 Design of Experiment. ... 131
9.2 TDD steps for development. ... 133
9.3 Performance mean values with error bars ... 135
9.4 Code coverage mean values with error bars ... 136
9.5 Code complexity mean values with error bars ... 136
9.6 Defects found mean values with error bars ... 137
9.7 How difficult was it to follow steps for development ... 137
9.8 Students perception of TDD ... 138
9.9 Students adherence to TDD ... 140

10.1 Distribution of Test Cases ... 161
10.2 Failing Occurrence for Test Cases ... 161

11.1 Number of Test Cases ... 185
11.2 Defects found by Test Cases ... 186
11.3 Quality of Test Cases ... 187

12.1 Test Driven Development Process Flow ... 198
12.2 TDDHQ additions to standard TDD Process Flow ... 200
12.3 Equivalence Partitioning steps provided to experiment participants ............................... 205
## List of Tables

4.1 Relation between research questions and publications ........ 46

7.1 Categorization of Respondents .................................. 71
7.2 Respondent Demographics ........................................ 73
7.3 Survey data for Agile vs. Non-Agile ................................ 75
7.4 Survey data for Distributed Development .......................... 76
7.5 Survey data for Application Domain .............................. 78
7.6 Survey data for Safety-Criticality .................................. 80
7.7 Survey data for Testers vs. Non-testers ........................... 81
7.8 Respondent Answers on Techniques in use ....................... 83
7.9 Respondent Answers on Tools in use ............................... 84
7.10 Dissatisfaction within categories of respondents .................. 86
7.11 Questions with the Lowest Degree of Dissatisfaction .......... 87
7.12 Questions with the Highest Degree of Dissatisfaction ........... 88

8.1 Searched databases ............................................. 96
8.2 Extracted study details ............................................ 98
8.3 Empirical Studies of TDD ......................................... 101
8.4 Areas of Effect of TDD ............................................ 102
8.5 Limiting Factors for TDD Adoption ............................... 103
8.6 Mapping Between Effect Observations and Primary Studies . . 105

9.1 Research publications on experiments in TDD ................. 129
9.2 Experiment Response Variables .................................. 130
9.3 Mann-Whitney $z$ scores for differences between experiment groups and objects .................................. 139
### List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.4</td>
<td>Testing of null hypotheses of the experiment using the Wilcoxon signed-rank test</td>
<td>139</td>
</tr>
<tr>
<td>10.1</td>
<td>Experiment Response Variables</td>
<td>155</td>
</tr>
<tr>
<td>10.2</td>
<td>Descriptive Statistics of Quality Attributes</td>
<td>156</td>
</tr>
<tr>
<td>10.3</td>
<td>$Q_{testing}$ for Positive and Negative Test Cases</td>
<td>162</td>
</tr>
<tr>
<td>10.4</td>
<td>A Complete Overview of Test Cases</td>
<td>163</td>
</tr>
<tr>
<td>11.1</td>
<td>Distribution of solutions per groups</td>
<td>182</td>
</tr>
<tr>
<td>12.1</td>
<td>A-B Combination for Classical TDD</td>
<td>202</td>
</tr>
<tr>
<td>12.2</td>
<td>A-B Combination for increased robustness in TDD</td>
<td>202</td>
</tr>
<tr>
<td>12.3</td>
<td>Test cases distribution</td>
<td>207</td>
</tr>
<tr>
<td>12.4</td>
<td>Defects detected by test cases</td>
<td>207</td>
</tr>
<tr>
<td>12.5</td>
<td>Defects found in code</td>
<td>208</td>
</tr>
<tr>
<td>12.6</td>
<td>Defect detecting effectiveness of test cases</td>
<td>209</td>
</tr>
<tr>
<td>12.7</td>
<td>Quality score per test case values</td>
<td>209</td>
</tr>
<tr>
<td>12.8</td>
<td>A-B Combination for mutation analysis in TDD</td>
<td>211</td>
</tr>
</tbody>
</table>
I

Thesis
Chapter 1

Introduction

The traditional software development life cycle has become inadequate to preserve quality of software products when organisations attempt to shorten their time-to-market. In many cases the quality control is often reduced or postponed due to the shortened deadlines or overrun of the development phase [1, 2]. Organisations are in need of a new process that will value quality in each stage of their product development without interfering with the product delivery schedule. They are increasingly turning their interest to agile methodologies [3]. However, performing efficient and effective software testing, as required by such methodologies, brings forward many challenges. Increased complexity of software systems, the need for a specific domain knowledge or the lack of testing experience are just a few obstacles a tester could face with. With the presence of Agile methods, quality of the software product becomes everyone’s responsibility, not just the quality assurance or the testing department. But how can we expect quality of software products from team members who are lacking training in quality assurance methods?

The test driven development is one example of how developers can focus on the quality of software by writing executable and automated test scripts before writing the actual code. Test driven development (TDD) was introduced as part of the eXtreme Programming (XP) methodology [4]. By writing test cases before the code, developers are using tests to guide them in making the correct implementation of the required functionality (Section 2.3 provide more details on TDD). Test driven development was identified, in our industrial survey [5], as a preferred but not so often used practice in industry.
One reason for this preference towards TDD, could be that academic research results are often highlighting improvements in the code quality when TDD is being followed instead of the traditional test last approach [6–10]. To better understand potential limiting factors of industrial TDD adoption, a systematic literature review was performed [11]. Seven factors, which are potentially limiting full adoption of TDD, were identified and listed. Developers’ inability to write efficient and effective automated test cases is one of the listed limiting factors.

These results focused our research effort towards investigating and reporting on the quality of test cases design, produced by developers following the TDD approach. We were mostly interested in differences in the design of traditional test cases created for the purpose of testing a software product, and a new paradigm of creating test cases for the purpose of guiding the software development process. Can we efficiently (re)use existing and proven test design techniques for creating developers’ tests in TDD?

We performed several empirical studies to gain better understanding of the quality of testing performed by developers using the TDD approach [12–14]. A common finding in those studies was that participants created significantly less negative test cases than positive ones (concepts explained in 2.5). In the literature, phenomenon of a more positive approach to testing is known as “positive test bias” [15, 16]. Such a result does not come as a surprise considering that TDD is designed in way to lead developers in writing of positive tests which will guide them forward in the implementation of a correct functionality. What we want to investigate is: What is the impact on developers’ testing effectiveness by having such a high number of positive tests in the test suite?

By defining new approaches of quantifying quality of testing (details in section 2.4), we could perform analysis of test cases based on their effectiveness in finding defects in the source code. What we found interesting was that the number of defects detected by using negative tests is significantly higher than those detected by positive ones even if they represented only a smaller portion of a test suite. This was a motivating reason to propose the concept of TDDHQ which is aimed at achieving higher quality of testing in test driven development by augmenting the standard TDD methodology with suitable test design techniques and thus help to overcome its positive test bias. To exemplify this concept, we combined equivalence partitioning test design technique together with the test driven development practice. Initial evaluation of this approach showed a noticeable improvement in the quality of test cases created by the group who followed this approach.
The research presented in this thesis proposes improvements in the process of performing TDD which are expected to result in not just higher quality of produced tests but also in higher quality and productivity in software systems, in terms of robustness and early defects detection.

1.1 Motivation and Problem Description

Today’s business needs are demanding from software organisations to accept a constant pace of change as it reflects the current market and economic demands. According to the agile philosophy delivering an evolving software product without having a predefined set of requirements that will be changed at a later stage is something companies should not fight against, but rather embrace. Agile software development is one representative of the current industrial solutions to this challenge.

But this comes at a price. Adopting agile development for many organisations creates not only a phase shift in thinking on how to develop software, but it also introduces a significant amount of changes to their daily activities [17]. These changes consist of facilitating continuous product integration, ability to prioritise tasks, committing to delivery all the way through daily stand-up meetings and burn-down charts.

In particular, changes affecting testing teams and testers may create additional confusion with respect to understanding who is responsible for the product quality and how to allocate time for this activity. In agile development, quality is everyone’s responsibility and having in mind that traditional testing can consume even more than 50% of the total development time [18], testers do have a concern of ensuring how this time will be allocated in agile development.

1.2 Outline of thesis

This thesis consists of two main parts. The first part is organised as follows: Chapter 2 explain concepts from the main areas of interest for this thesis. Chapter 3 outlines design of research elaborating on its objective, questions and methodology. Chapter 4 presents a summary of the main research contributions. Chapter 5 provides description of related work, while thesis conclusion and guidelines for future work are outlined in Chapter 6. The second part of the thesis consists of Chapters 7 through 12 which represent research publications included in this thesis.
Chapter 2

Background

In this thesis we use several concepts from different research areas, viz., Agile development, software testing, test driven development, quality of testing and negative testing. The following sections present some key concepts from these areas which should be introduced to the reader before providing the details on the contributions of this thesis.

2.1 Agile Development

Agile development is considered a relatively young software engineering discipline that emerged from industrial needs for a software development process where the main focus should be on the customer and their business needs. The idea is to have a constant communication channel with the customer by iteratively providing working software product with currently most needed business values built in. Historically, the idea behind an agile approach is actually not new. It was reported [19] that NASA Project Mercury (first US human spaceflight program in the 1960s) used time-boxed iterations with tests written before each increment - an activity very similar to what is known today as test-driven development (TDD).

Agile is not a software development process by definition, but rather a philosophy based on a set of principles. These principles are listed in the “Agile Manifesto” [20]. Since understanding of agile is relying on those twelve principles, we are listing them here.
Chapter 2. Background

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

7. Working software is the primary measure of progress.

8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9. Continuous attention to technical excellence and good design enhances agility.

10. Simplicity - the art of maximizing the amount of work not done - is essential.

11. The best architectures, requirements, and designs emerge from self-organizing teams.

12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile Manifesto [20]

By following these principles, organisations are committing to have a continuous feedback with customer and provide value to their business needs.

Several software development processes use some of those principles, like: eXtreme Programming (XP), Scrum, Dynamic Systems Development Method (DSDM), Feature Driven Development (FDD), etc. usually referring to them
as *agile software development methods*. Aside from following agile principles, each of those methods contains different *agile practices*. Pair programming (PP), test-driven development (TDD) and continuous integration (CI) are just a few of these agile practices.

![Figure 2.1: A sprint overview in the Scrum process](image)

An overview of a Scrum iteration (sprint), as an example of an agile development process, is shown in Figure 2.1. A prioritised product backlog is used to select user stories for the upcoming sprint. By dividing them into concrete tasks, they become part of the current sprint backlog. During the period of 2-4 weeks items in the current sprint are completed on a daily basis. After each sprint a potentially shippable product increment should exist.

### 2.2 Software Testing

Software testing is a major activity in software development and has two main goals:

- **to confirm** a software solution is behaving as per its requirements, and
- **to find faults** in a software which are leading to its misbehaviour.

It is important to note how testing cannot be used as a proof of fault free software. A famous quote from Edsger Dijkstra [21] reads as follows: “Testing can only show the presence of errors, not their absence”. One of the reasons why we cannot claim there are no faults in software is in the fact that exhaustive testing of input values, which could influence the system outcome, cannot be performed in a realistic time.
Commonly, there are three levels of testing of software systems [18]:

- **System level** - has the purpose of testing overall system functioning from a user perspective.

- **Integration level** - has the purpose of testing interconnections between various components/modules during their integration phase.

- **Unit level** - has the purpose of testing functional and non-functional properties of a single unit/module/component of the system.

Software testing is a widely researched domain of its own with a multitude of techniques and tools proposed for industrial practice. A comprehensive discussion on this vast research domain is beyond the scope of this thesis and hence not attempted.

### 2.3 Test-driven development

Test-driven development (TDD), sometimes referred to as test-first programming [22], is a practice within the extreme programming development method proposed by Kent Beck [4]. TDD requires the developers to construct automated unit tests in the form of assertions to define code requirements before writing the code itself. In this process, developers evolve the systems through cycles of testing, development and refactoring. This process is shown in Figure 2.2. In their experiment, Flohr and Schneider [23] prescribed TDD activities to students as the following list of activities:

1. **Write one single test-case**
2. **Run this test-case. If it fails continue with step 3. If the test-case succeeds, continue with step 1.**
3. **Implement the minimal code to run the test-case successfully**
4. **Run the test-case again. If it fails again, continue with step 3. If the test-case succeeds, continue with step 5.**
5. **Refactor the implementation to achieve the simplest design possible.**
6. **Run the test-case again, to verify that the refactored implementation still succeeds the test-case. If it fails, continue with step 5. If the test-case succeeds, continue with step 1, if there are still requirements left in the specification.**
2.4 Quality of Testing

Quality of software has been an active research area for the past few decades and several software measures ranging from simple lines of code to various complexity measures can be found in the literature to evaluate and improve the software quality from process or product perspectives [24]. However there is no consensus on the universal applicability of any specific software quality metric and their usage had been more context specific and based on the intended objectives.

There are several approaches on how to perceive more information about the quality of the test cases accompanying with the source code of a software product. A commonly used approach in industry is calculation of the code coverage metrics, whilst coverage using mutation testing is considered more preferable by academia for test suite evaluation.

2.4.1 Code Coverage

In its essence, code coverage will monitor which parts of the software system are accessed while a specific test case or set of test cases is executed on
the given system. Based on what parts of the system are monitored we have different code coverage metrics. Here, we list some of them:

- Statement coverage is considered one of the most intuitive coverage metrics. If we consider statements as parts of the system we would like to measure, we basically divide sum of all program statements exercised with test cases with the total number of statements in the program.

- Branch coverage metric calculates to what extent are all the branches in the program accessed once the test suite is executed.

- Condition coverage represents a more rigorous metric compared to the branch coverage, by further evaluating different individual boolean conditions used to define each branch expression.

The main benefit of performing any of the above mentioned code coverage techniques is the ability to provide a very quick and intuitive feedback about which parts of the system are not exercised by a given test suite. However, we have to take the claims of a “100% code coverage” with caution. For example, let us have a look at one solution (consisting of source code and test cases) for which we do not have any information about the quality of code and tests.

![Figure 2.3: Quadrants of relations between quality of code and tests](image-url)
We want to perform a quality of testing analysis on that solution using code coverage metrics and plot the result in the dedicated quadrant within Fig. 2.3. In case we have achieved 100% code coverage for our solution, we cannot be sure which quadrant to choose from. For example, high quality tests executed on both high quality code (quadrant I) and low quality code (quadrant IV) could in return provide 100% code coverage. Executing low quality tests on low quality code (quadrant III) may as well do the same. So, how can we really know the difference between quadrants I, III and IV?

2.4.2 Mutation Testing

One way to distinguish quadrants I, III, and IV is the usage of mutation testing. This technique will simply create faulty versions of our code by deliberately seeding errors in it and thus focusing our attention to quadrants III and IV, since now we know that our code is not of high quality. The mutation testing metric score is calculated by the following steps:

1. Source code is taken as an input
2. Test cases (test suite) are taken as well as an input, but with the pre-requirements that they all passed on the original source code.
3. Using a set of mutation operators, an \( N \) set of variations of the original program is generated (they are called mutants)
4. For each mutant \( n \in N \) the test suite is executed
5. If any test case within the test suite fails, current mutant \( n \) is marked as “killed”.
6. Total number of killed mutants is \( m (m \leq |N|) \).
7. Mutation score is calculated as \( m / |N| \)

The main benefit of using mutation testing is the possibility to obtain a better understanding of the test suite quality as compared to code coverage techniques. There are also specific drawbacks of using this technique:

- Much more computational power is necessary for it to be performed and it will take significantly more time than coverage techniques making it almost inappropriate for obtaining quick feedback.
- Mutation testing can often report false positives. A false positive is a potential bug that turns out not to be a bug and therefore consume unnecessary developers’ time in performing debugging.
However, the main problem in our analysis of test cases is that mutation testing technique relies on the actual code implementation, which could have been wrong in the first place (misunderstood specification, wrong assumptions, etc). This is why we have to approach the problem in a different way (elaborated in Section 4.1.2).

2.5 Negative Testing

By the term negative test case, we refer to a test case that was created for the purpose of exercising a program in a way that was not explicitly specified in the requirements. On the other hand, a positive test case exercises a program behaviour as it is specified in the requirements.

For example, a specification might state: “... numbers are accepted as an input to the program ... ”, and testing such a program with any numerical input is considered as positive testing. For the same program and specification, if testing is performed by providing a non-numerical input, such an approach can be called negative testing.

Even in the case of an implicit specification, for example: “... only numbers are accepted as an input to the program ... ”, testing using non-numerical inputs can still be considered as negative testing, unless it is explicitly stated in the specification how the program should behave upon receiving non-numerical inputs.
Chapter 3

Research Design

In this chapter we present details regarding the design of the research conducted within this thesis, including research objective statement, listing of research questions and discussion of different research methodologies used.

3.1 Research Objective

The main influence on setting up the research focus of this thesis came from the FLEXI project [25], which had the goal of scaling up agile product development in large organisations. As partners in this project, our overall task was to investigate how software testing activities in an agile environment are performed and how efficiently and effectively we can use the set of known test design techniques when following agile development. By reusing already known and well established test design techniques, an organisation could potentially benefit in the form of an increased quality of the software product. This was a basis to setting up the research area for further investigation and we formulated the following overall research objective at the start of our research:

**Investigate testing practices in agile development environments and propose methods for improving the overall quality of software products**
3.2 Research Questions

Since our initial overall research objective represented a more general overview of the agile testing research area, we had to identify a more specific and important challenges which could be practically further investigated. We followed a simple and intuitive approach to define our research questions by focusing on a high-level approach consisting of a set of three generic questions, as listed in Figure 3.1.

![Flow of Three High-level Generic Questions](image)

Figure 3.1: Flow of Three High-level Generic Questions

As a result, an initial set of research questions are formulated as:

- **RQ1**: What is the current agile testing practice, which is not extensively adopted in industry?
- **RQ2**: What are the reasons for not adopting the identified practice in industry?
- **RQ3**: What needs to be improved in order to overcome the identified reasons and contribute to a wider industrial adoption of the identified practice?

Since our approach in performing research is of an iterative nature, refinements are constantly made to research questions as the new results are obtained. By doing such, we can have a more focused and addressable research questions at each level of iteration.

3.3 Research Methodology

The research is based on empirical methodologies including analysis of qualitative and quantitative data. Literature and industrial surveys were performed in order to perceive the state of the art and the state of practice. Experiences from industry on this topic were collected and summarised with the research in a reusable form on a higher level of abstraction intended as guidelines for organisations transitioning into agile.
Chapter 4

Research Contribution

Since this thesis is written as a collection of papers, its contributions are based on the findings from each individual research paper included. In Section 4.2 we have outlined a short summary of each publication and discussed how their results foster in providing answers to the previously defined research questions. However, we would like to give a high level perspective of the main contributions of our research, also elaborating on some of the works which significantly impacted and further defined our research progress.

4.1 Main Contributions

In following sections we present in more details the three main contributions of the overall research results of this thesis: (i) identification of potential limiting factors for industrial adoption of TDD, (ii) proposal of a new method for estimation of testing quality in empirical experiments and (iii) proposal of TDDHQ concept as an approach for achieving higher quality testing in TDD.

4.1.1 Identification of Limiting Factors in TDD Adoption

In our systematic literature review [11] we identified seven potentially limiting factors (LF) for the wide spread industrial adoption of TDD. An overview of these factors is listed here together with the observations from the primary studies as well as motivations for their inclusions.
LF1: Increased development time
By development time, we refer to the time required to implement a given set of requirements. Time required for development of software product is relatively easy to measure. It is however a matter of discussion whether the time for corrective re-work (e.g., based on failure reports from later testing stages) should be included in the development time or not. Depending on the maturity of the organization, an up-front loss (in this case, increased development time) might overshadow a long-term gain (e.g., decreased overall project time, or increased product quality both of which were reported in many of our included studies). Hence, internal organizational pressure might risk the proper usage of TDD.

LF2: Insufficient TDD experience/knowledge
By TDD experience/knowledge, we refer to the degree of practical experience, as a developer or similar, or theoretical insight in TDD. When observing collected data from the included primary studies, we noticed that participants in the experiments (either students or professionals) were mostly provided with some training or tutorial on how to perform TDD. In several cases [26], the knowledge improved as participants would progress with the experiment. We expect that lack of knowledge or experience with TDD could create problems in its adoption.

LF3: Insufficient design
Design, in this context, refers to the activity of structuring (or restructuring) the system or software under development or in evolution in order to avoid architectural problems, and to improve architectural quality. Detailed up-front software design is a common practice of plan-driven development methodologies. TDD emphases on a small amount of up-front design, and frequent refactoring to keep the architecture from erosion. There is no massive empirical support that the lack of design should be considered as a limiting factor for industrial adoption of TDD. However, there are a handful of studies reporting problems regarding lack of design in TDD, particularly in the development of larger, more complex systems. Moreover, the lack of upfront design has been one of the main criticisms of TDD since its introduction and even if the evidence supporting this criticism is sparse, so is the evidence contradicting it [27].
4.1 Main Contributions

**LF4: Insufficient developer testing skills**
By developer testing skill, we refer to the developer’s ability to write efficient and effective automated test cases. Since TDD is a design technique where the developer undertakes development by first creating test cases and then writes code that makes the test cases pass, it relies on the ability of the developer to produce sufficiently good test cases. Additionally, Geras [28] reports on the risk it brings to adopt TDD without having adequate testing skills and knowledge. We find it interesting that there are no explicit investigations of the quality of test cases produced by developers in TDD.

**LF5: Insufficient adherence to the TDD protocol**
By adherence to the TDD protocol, we refer to the degree to which the steps of the TDD practice are followed. For example, are test cases always created and exercised to failure before the corresponding code is written? TDD is a defined practice with fairly exact guidelines on how it should be executed. In the studies it is stated that (1) it is important to adhere to the TDD protocol, and (2) developers do stray from the protocol in several situations. It is however far from certain that there is a clean-cut cause-effect relationship between low TDD adherence and low quality. Not unlikely, confounding factors (e.g., tight development deadlines) might lead to both low TDD adherence and poor quality.

**LF6: Domain- and tool-specific limitations**
By domain- and tool-specific limitations, we refer to technical problems in implementing TDD (e.g., difficulties in performing automated testing of GUIs). Generally, the TDD practice requires some tool support in the form of automation framework for test execution. The single most reported issue is the problem of automatically testing GUI applications, but also networked applications seem to be problematic in terms of automated testing. Proper tool support for test automation is vital for the successful adoption of TDD. With the wide variety of studies reporting domain- and tool-specific issues as a limiting factor in the adoption of TDD, the factor would be difficult to ignore.

**LF7: Legacy code**
By legacy code, we refer to the existing codebase in a development organization. Legacy code often represent decades of development efforts and investments, and serve as a backbone both in existing and future products. TDD, in its original form, does not discuss how to handle
Chapter 4. Research Contribution

legacy code. Instead, the method seems to assume that all code is developed from scratch, using TDD as the development method. As this is seldom the case in large development organization, adoption of TDD might be problematic. A lack of automated regression suites for legacy code hampers the flexibility provided by the quick feedback on changes provided by the regression suites, and may leave developers more anxious about how new changes may unexpectedly affect existing code.

4.1.2 Methods for Estimation of Testing Quality

In most empirical studies performed by researchers, participants work on the solution for the same problem to minimise validity threats and ease the process of analysis. Although this is not a commodity from which industry can benefit, academics do have an opportunity to execute test cases of their participants against each others’ code gaining much more inside information about the test cases’ effectiveness in finding defects. This is of great value especially when test driven development is the subject of investigation, since the final set of test cases that accompany a software solution developed using TDD will show only its correctness but no defects in the same.

In order to realistically measure the quality of testing we need to essentially have access to an ideal test suite which is capable of finding all the defects. Our approach here was to approximate such an ideal test suite by combining all the test suites developed by several individual developers working on the same problem. Given such a set of multiple implementations and associated test suites, we were then able to cross-compare the ability of test cases to find defects. We have defined the following quality metrics used in our studies:

**Defect Detecting Ability**

Defect detecting ability represents the total number of defects a particular test case can find in all the implementations of the same problem created by different developers. This number could be also calculated for all test cases created by a single developer, but more interestingly in the context of this study is to calculate how many defects are detected by negative and positive test cases. Additionally, considering the differences in the expertise levels of the developers, we would like to give a higher quality value to a test case that is capable of finding defects in an implementation of high quality. Hence the evaluation of the quality of test cases will be much more meaningful if we jointly address it together with the quality of the code in which we apply them.
4.1 Main Contributions

**Quality of Code** \( (Q_{\text{code}}) \)

The main reason why we need to calculate quality of code is to support calculation of a *Quality of Tests* attribute. Quality of the code for every developer \( (i) \) is calculated using the next formula:

\[
Q_{\text{code}}(i) = 1 - \frac{N_{\text{FTC}}(i)}{N_{\text{TC}}}
\]

where, \( N_{\text{TC}} \) is a total number of test cases created by all developers and \( N_{\text{FTC}}(i) \) represent total number of failing test cases on the code of a developer \( (i) \) by executing all test cases from all developers. Once we calculated *Quality of Code* value for each developer, we can now fine tune the rewarding of test cases based on their ability to detect defects in codes of varying quality levels.

**Quality of Tests** \( (Q_{\text{tests}}) \)

Quality of test cases for a developer \( (i) \) is calculated as a sum of the quality of each test case \( (j) \) from a set of test cases \( (n) \) of that developer \( (i) \):

\[
Q_{\text{tests}}(i) = \sum_{j=1}^{n} Q_{\text{TC}}(i,j)
\]

To calculate the quality of an individual test case \( (j) \) of a developer \( (i) \) we need to know on which developers’ code this test case is failing \( (m \in M) \). The sum of the *Quality of Code* values \( (Q_{\text{code}}) \) of those developers will define the quality of a particular test case \( (j) \):

\[
Q_{\text{TC}}(i.j) = \sum_{k=1}^{m} Q_{\text{code}}(k)
\]

Once this calculation is done for every developer, we can have a much better understanding of how much each test case contribute to the overall quality of testing. In the context of this thesis, we can also specifically observe how much negative and positive test cases contribute to the quality of testing.
4.1.3 TDD$^{HQ}$ - Higher Quality Testing in TDD

In order to increase the quality of test cases created when using test driven development, we propose a modification to the standard TDD process flow, named TDD$^{HQ}$ - Higher Quality Testing in Test Driven Development, detailed in Figure 4.1. Based on the TDD process flow, a few additional process steps were added, keeping the basic approach of TDD unchanged.

- **A** - Choosing Quality Improvement Aspect

When testing a software product, members of quality assurance teams investigate various aspects of product quality: functionality, performance, security, usability, robustness, etc. For them, it is important to cover both functional and non-functional quality features. However, developers tend to focus mainly on the functional aspects of software quality, as it was noted in our previous empirical studies. This is why it is needed for a developer to explicitly choose an aspect of quality improvement during test driven development which will further guide them to design of additional test cases necessary to achieve the desired quality improvement.

- **B** - Selecting Test Design Technique

After deciding on the quality improvement aspect that should be in the focus of the current iteration, one of the appropriate test design technique should be selected. It is however important that this test design technique directly contribute to the previously chosen quality improvement aspect. In case there is a possibility to select two or more complementary test design techniques, developers could choose to iterate the process flow with the same quality improvement aspect but each time using a different test design technique for the same requirement.

- **Check Whether More TCs for B in A**

Once the quality improvement aspect and the test design technique are determined, a classical red-green phase of TDD is conducted. Since a particular test design technique could require (by design) creation of several test cases, it is important to reflect if more test cases are needed for a given test design technique selected in B to satisfy a quality improvement aspect selected in A. If more test cases are needed, an additional red-green phase should be conducted for each test case individually.
4.1 Main Contributions

Figure 4.1: TDD\(^{HQ}\) - Higher Quality Testing in Test Driven Development
Chapter 4. Research Contribution

- Check Whether All Q.I. Aspects Covered

After fully implementing a feature with the specific quality improvement aspect in mind and using the appropriate test design technique(s), developers could iterate through the implementation of the selected feature with a new A - B combination, or choose to continue with adding new features, if they are needed. Sometimes, the decision on usage of the set of A-B combinations could be taken upfront, for all the features by the individual developer, or it could be enforced at the organisational level as a development policy implied to all development teams.

4.2 Individual Papers’ Contributions

This section describes the contributions of all papers included in the thesis. The relation between contributions from individual research papers and research questions defined in section 3.2 is presented in Table 4.2.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RQ2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RQ3</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.1: Relation between research questions and publications

4.2.1 Paper A


Summary Using data from an industrial survey [29] a state of the practice study was conducted. The survey in addition to confirming some popular beliefs also lists several noteworthy findings from the perspectives of respondent categories such as safety-criticality, agility, distribution of development, and application domain. These findings clearly depict negative discrepancies between the current practices and the perceptions of the respondents. This paper covers RQ1 and provide contribution by identifying test-driven development
(TDD) as a factor with maximum difference between current and preferred practice, making it least used by our industrial respondents.

**My contribution**  I was the main author of this paper contributing by formulating the problem, designing the questions and performing data analysis. The co-authors supervised and helped in the discussion of findings and descriptive statistics.

### 4.2.2 Paper B

*Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review*, Adnan Čaušević, Daniel Sundmark and Sasikumar Punnekkat, In proceedings of the International Conference on Software Testing (ICST), Berlin, Germany, March 2011

**Summary**  As a direct result of investigation from Paper A, a systematic literature review on TDD was performed. After initial keyword search on seven major research databases, results yielded 9462 publications. In several steps we removed publications that are not of interest resulting in 48 publications as the final number of our systematic review. The process of extracting effects areas on TDD or effects areas of TDD from selected research publications and identifying limiting factors contributed to RQ2. Seven limiting factors were identified viz., increased development time, insufficient TDD experience/knowledge, lack of upfront design, domain and tool specific issues, lack of developer skill in writing test cases, insufficient adherence to TDD protocol, and legacy code.

**My contribution**  I was the main author of this paper contributing in obtaining the collection of papers from the search databases, filtering and removal as well as analysis of findings presented in selected collection of papers. The co-authors helped to filter the papers and also performed reading of selected list of publications to validate the findings.

### 4.2.3 Paper C

Summary Among the seven limiting factors identified from the systematic study in Paper B, knowledge of testing was selected to be further investigated as part of a controlled experiment with master students in order to zoom-in on the research question RQ2. The experiment was designed around a Master level course on Software Verification and Validation at Mälardalen University. Participants were divided into two groups solving two problems on two different occasions, before and after the course. The analysis was performed on the collected source code and test scripts created by students, as well as based on a questionnaire survey responses. Results are showing positive improvements of test code coverage but no statistically significant difference exists between pre- and post-course groups. In depth exploratory data analysis revealed lack of negative test cases resulting in students inability to detect bugs related to unspecified behaviours.

My contribution I was the main author of the paper, contributing in setting up the pre-requirements for the experiment (lab instructions, problems user stories, SVN, etc.), collecting data points and performing the analysis. The co-authors provided overall guidance as PhD supervisors and helped in study design and analysis of the data.

4.2.4 Paper D

Quality of Testing in Test Driven Development, Adnan Ćaušević, Sasikumar Punnekkat and Daniel Sundmark, In proceedings of International Conference on the Quality of Information and Communications Technology (QUATIC), Lisbon, Portugal, September 2012

Summary In order to fully understand the impact of lack of negative test cases in test driven development a follow up academic study was performed as another contribution to RQ2. Design of this experiment was focused on having a clear separation of negative and positive test cases created by experiment participants who were divided in groups following two development approaches: (i) test first and (ii) test driven development. To gain better understanding of how valuable test cases of each participant are, they were executed against each other participants’ code providing much more data points for analysis. On an average, the quality of testing in test driven development was almost the same as the quality of testing using test-last approach. However, detailed analysis of test cases, created by test driven development group, revealed that 29% of test
cases were negative tests but contributing as much as 65% to the overall tests quality score of test-first developers.

**My contribution** I was the main author of the paper, contributing in setting up the experiment (instructions, problem description, infrastructure), collecting data points and performing the analysis. The co-authors helped in study design, analysis and supervision of the overall study execution process.

### 4.2.5 Paper E

*Effects of Negative Testing on TDD: An Industrial Experiment*, Adnan Čaušević, Rakesh Shukla, Sasikumar Punnekkat and Daniel Sundmark, In proceedings of the International Conference on Agile Processes in Software Engineering and Extreme Programming (XP), Vienna, Austria, June 2013

**Summary** One of the threats to the validity of the results in Paper D may come from the fact that the experiment was performed in an academic environment using students as participants. Lack of their professional experience could be considered as the main reason why students did not focus on creating higher number of negative tests. Therefore, it was needed to replicate a study using professional developers as experiment participants, verifying if a positive test bias effect does indeed exists in test driven development. Such an experiment was performed at Infosys Ltd. India where the collected data indicated a statistically significant difference between the number of positive and negative test cases, confirming the existence of positive test bias. As a result, similarly to our previous study, 29% of all test cases were negative, but at the same time revealing as much as 71% of all the defects.

**My contribution** I was the main author of the paper and visited Infosys India as part of a student internship to set up and execute the experiment, collect data and perform the analysis. The co-authors helped in modification of the original study design, analysis and supervision of the overall study execution process.

### 4.2.6 Paper F

*TDDHQ: Achieving Higher Quality Testing in Test Driven Development*, Adnan Čaušević, Sasikumar Punnekkat and Daniel Sundmark, (In submission)
Summary  As we observed in our previous studies, the number of defects detected with negative test cases is significantly higher than those detected by positive ones when following a test driven development approach. The problem is that developers do not write a high number of negative tests and thus miss the opportunity of early defect detection. Even when instructing developers to write negative tests, those are mostly based on random test approach, rather than following a precise testing technique. In this paper we propose the concept of TDDHQ which is aimed at achieving higher quality of testing in test driven development by augmenting the standard TDD methodology with suitable test design techniques helping to overcome positive test bias. With this approach we are contributing to RQ3. To exemplify this concept, we present combining equivalence partitioning test design technique together with TDD practice. Initial evaluation of this approach showed a noticeable improvement in the quality of test cases created by developers utilising TDDHQ concept compared to a group of students using regular test driven development practice.

My contribution  I was the main author of the paper, contributing in setting up the experiment (instructions, problem description, infrastructure), collecting data points and performing the analysis. The co-authors helped in study design, analysis and supervision of the overall study execution process.
Chapter 5

Related Work

Empirical studies, performed for the purpose of investigating potential benefits of TDD, focus mainly on the differences in the quality of the produced code. This was one of the findings from our systematic literature review [11], where we listed 48 empirical studies that had effects of TDD as the focus of the investigation. In most cases, TDD was the primary focus of the investigation, but in some studies TDD was used together with a different practice, e.g. pair-programming. Studies presented in this literature survey investigated TDD with relation to: (i) the internal or the external code quality, (ii) performance improvements or (iii) a general perception on TDD. Those studies are summarised in Section 5.1. Several studies with the more specific focus on the test cases created in TDD are presented in more details in Section 5.2.

5.1 General Studies on TDD

In this section we are presenting empirical studies grouped in several subsections based on the aspects of TDD they are investigating with the exception of studies focusing on testing which are described separately in Section 5.2. For that reason it is difficult to relate directly our work and results with these general studies on TDD. However, they did contribute significantly to a better understanding of how empirical studies on TDD could be designed and executed, which metrics could be applied and how data analysis could be performed.
5.1.1 Studies on Benefits of TDD

Müller & Hagner [30] performed an experiment with students divided into two groups, test-first and traditional, with focuses on the programming efficiency, the reliability of the resultant code and program understanding. As a result they conclude that test-first does not accelerate programming, produced programs are not more reliable but test-first support better understanding of program. Flohr & Schneider [23] had an experiment with students divided into two groups (test-first and classical-test) for the purpose of investigating impact of test-first development process. They could not establish any significant differences in their results, but students did show a preference towards test-first approach. Gupta & Jalote [9] performed an experiment with students divided in two groups (TDD and waterfall) evaluating the impact of TDD on designing, coding, and testing where data is obtained by questionnaire and forms. They point out that TDD improves productivity and reduce overall development effort. However, code quality is affected by test effort regardless of the development approach in use. Kollanus & Isomöttönen [31] performed experiment with students on understanding TDD and perception on difficulties of TDD, where again data was collected by the questionnaire. Students expressed difficulties with following TDD approach and designing proper tests. Regardless, students believed in the claimed benefits of TDD.

5.1.2 Studies on Quality of Produced Code

George & Williams [6] had professional developers from three companies in TDD and waterfall-like control groups to investigate code quality improvements. Their results point out that test-driven development produces higher quality code with the tendency of developers spending more time on coding. Another controlled experiment of Janzen & Saiedian [8] examined the effects of TDD on internal software design quality. The experiment was conducted with undergraduate students in a software engineering course. Positive correlation between productivity and TDD was found, but no differences in internal quality. Perception of participants on TDD was more positive after the experiment. Janzen et al. [32] had empirical studies in three industry short courses investigating effects of test-driven development (TDD) on internal software quality. They point out that programmers’ opinions on TDD improved after the experiment but internal code quality had no significant difference between test-first and test-last approach. Vu et al. [10] performed an experiment with students divided in two experimental groups (test-first and test-last) in a year-
5.2 Studies on Quality of Testing in TDD

long software engineering course evaluating productivity, internal and external quality of the product, and the perception of the methodology. They conclude that test-last team was more productive and created more tests. Also, students indicated preference towards test-first approach.

5.1.3 Studies on Productivity Improvements with TDD

Geras et al. [28] executed experiment with professional developers divided in two groups working on two problems using test-first and test-last processes to investigate productivity and software quality. As a result, there were little or no differences in developer productivity but frequency of unplanned test failure was lower for test-driven development. Huang & Holcombe [33] had a controlled experiment with students that investigated the distinctions between the effectiveness of test-first and test-last approaches. They point out that test-first teams spent more time on testing than coding compared to test-last teams. However, there was no linear correlation between effort spent on software testing and the software external quality.

5.1.4 Studies on Impact of Experience in TDD

Müller & Höfer [34] investigated conformance to TDD of professionals and novice TDD developers. Results showed how experts complied more to the rules of TDD and produced test with higher quality. Höfer & Philipp [35] performed an experiment with professionals and students investigating if expert programmers conform to TDD to a higher extent than novice developers. They conclude that experts refactored their code more than novice programmers, but they were also significantly slower.

5.2 Studies on Quality of Testing in TDD

Erdogmus et al. [7] performed an experiment with undergraduate students divided into two groups (test-first and test-last) investigating test per unit effort, quality and productivity. As a result, test-first students created on an average more tests and tended to be more productive. There was no significant difference in quality of produced code between two groups. Madeyski [36] had an experiment with students divided in test-first and test-last groups examining branch coverage and mutation score indicator of unit tests. In this experiment, 22 students were divided in two groups: the test-first and the test-last, with the
task of developing a web based conference paper submission system. This experiment shows no statistically significant differences in branch coverage and mutation score indicators, between the test-first and the test-last groups.

After performing our systematic review, we noticed one additional study [37] with the focus on developers testing ability when following the test driven development approach. This was an industrial observational experiment where developers performed programming tasks in their own offices without the control of researchers. Once developers submitted their code and test cases, researchers performed mutation testing to identify complementary test cases.

In one of our academic studies [38] we measured the code coverage and the mutation score indicators in a very similar way as was done by Madeyski [36] in his study. Just as Madeyski we were not able to notice any differences between the experiment groups. However, we have opted not to measure and analyse these two attributes in our industrial experiment. The reason is that both those indicators are considered as internal quality attributes of a test suite, while we are more interested in measuring the actual effectiveness of a test cases with respect to the defect detecting ability. Specially, this is useful to distinguish the effect that positive and negative test cases have on overall testing effort. This is why our study-design enforces the same programming interface for all participants, allowing us to execute the test cases of one participant on the code of all other participants.
Chapter 6

Conclusions and Future Work

This thesis presents a set of activities conducted as part of our research in order to identify and address potential challenges of software testing in agile development. By performing various empirical studies (questionnaire survey, literature review and controlled experiment) we brought upfront test-driven development as a noteworthy testing research direction, investigating why this practice is not utilised to a higher extent within industrial settings.

During our investigation of the current body of knowledge [11], we identified 18 effect areas out of which 7 were considered as limiting factors on the industrial adoption of TDD, namely, increased development time, insufficient TDD experience/knowledge, lack of upfront design, domain- and tool-specific issues, lack of developer skill in writing test cases, insufficient adherence to the TDD protocol, and legacy code.

We set up a controlled experiment with master students to investigate if developers knowledge of testing can affect adoption of TDD [12]. Two groups of students were using TDD to solve two juxtaposing problems before and after the course on Software Verification and Validation. It is noticeable that code coverage increased in both groups after the course, but we could not identify any statistically significant differences between the groups. Further analysis of students achievements revealed lack of test cases focusing on negative testing.

This was the reason to again conduct several additional empirical studies [13, 14], with master students as well as with professional developers, to investigate if there are any differences in the defect detecting effectiveness of
positive and negative test cases created when following the TDD approach. The difference in defect detecting effectiveness, between positive and negative test cases, was statistically significant. Around 29% of all test cases created by developers were negative, but at the same time contributing in revealing as much as 71% of all the defects found by all test cases.

To increase quality of test design in TDD, we proposed the TDD<sup>HQ</sup> concept, an approach for achieving higher quality testing in test driven development. The TDD<sup>HQ</sup> approach is focused on augmenting the standard TDD methodology with a suitable test design technique, for the purpose of satisfying a criteria of a specific quality improvement aspect we would like to achieve. By using TDD<sup>HQ</sup>, developers do not necessarily focus only on verifying functionality, as it is defined in the given requirements, but they can as well increase security, robustness, performance and many other quality improvement aspects for the given software product, without interfering with the classical TDD flow.

In summary, the main contributions of this thesis are:

- The identification of TDD as a practice which is not used to the extent industry would prefer.
- Listing seven potentially limiting factors for industrial adoption of TDD
- Pointing out student’s inability to write negative test cases during controlled experiment
- Confirming, statistically significant, difference in defect detecting effectiveness of positive and negative test cases
- Proposing the TDD<sup>HQ</sup> concept for achieving higher quality testing in TDD

Concerning future work, the process of identifying limiting factors for industrial adoption of TDD was conducted using peer-reviewed scientific publications that have been addressing validity threats of their empirical study. In order to confirm significance of identified limiting factors our future work will focus on obtaining insights from industrial reports which were not covered in our previous study due to the validity requirements. This will be done in combination with industrial interviews to cover the full scope of obstacles for full utilisation of the test-driven development approach.

The TDD<sup>HQ</sup> concept uses various combinations of quality improvement aspects and test design techniques to achieve higher quality testing in TDD.
In this context, we need to map appropriate test design techniques with quality improvement aspects for different domains to ease the process of utilising TDD\textsuperscript{HQ} concept.

From a long term research perspective, we intend to perform an industrial case study investigating how experienced developers could benefit from TDD\textsuperscript{HQ} and what kind of specific combinations of quality improvement aspects and test design techniques they need in order to increase the quality of the code artefacts they produce.
Bibliography


