Davide Fucci

THE ROLE OF PROCESS CONFORMANCE AND DEVELOPERS' SKILLS IN THE CONTEXT OF TEST-DRIVEN DEVELOPMENT

UNIVERSITY OF OULU GRADUATE SCHOOL, UNIVERSITY OF OULU, FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING
DAVIDE FUCCI

THE ROLE OF PROCESS CONFORMANCE AND DEVELOPERS’ SKILLS IN THE CONTEXT OF TEST-DRIVEN DEVELOPMENT

Academic dissertation to be presented with the assent of the Doctoral Training Committee of Technology and Natural Sciences of the University of Oulu for public defence in Auditorium IT116, Linnanmaa, on 6 May 2016, at 12 noon

UNIVERSITY OF OULU, OULU 2016
Fucci, Davide, The role of process conformance and developers' skills in the context of test-driven development.
University of Oulu Graduate School; University of Oulu, Faculty of Information Technology and Electrical Engineering
University of Oulu, P.O. Box 8000, FI-90014 University of Oulu, Finland

Abstract
Modern software development must adapt to demanding schedules while keeping the software at a high level of quality. Agile software development has been adopted in recent years to meet such a need. Test-driven development (TDD) is one practice that has arisen within the agile software development movement that leverages unit tests to develop software in incremental cycles. TDD supporters claim that the practice increases the productivity of the practitioners who employ it, as well as the internal and external quality of the software they develop. In order to validate or refute such claims, the software engineering research community has studied TDD for the last decade; the results of the empirical studies on the effects of TDD have been mostly inconclusive.

This dissertation has studied two factors that may impact the manifestation of the claimed effects of TDD on software’s external quality and developers’ productivity: the developers’ conformance to the process (i.e., their ability to follow TDD) and their skills. The research was performed in four phases. In the first phase, the literature was reviewed to identify a set of factors that have been considered to affect TDD. In the second phase, two experiments were executed within academia. A total of 77 students at the University of Oulu, took part in the studies. These experiments investigated the quality of the software, as well as the subject’s productivity with respect to their programming and testing skills. A follow-up study, using data collected during the second experiment, explored the relation between the quality, productivity and the subjects’ process conformance. In the third phase, four industrial experiments, involving 30 professional, were performed. Process conformance and skills were investigated in relation to the TDD’s effects on external quality and productivity. The fourth phase synthesized the evidence gathered in the two previous phases.

The results show that TDD is not associated with improvements in external quality, or developers’ productivity. Further, improvements in both external quality and productivity are associated with skills, rather than with the process, at least in the case of professional developers. Hence, process conformance has a negligible impact. The productivity of novice developers, on the other hand, can benefit from the test-first approach promoted by TDD.

Keywords: agile software development, controlled experiment, developers’ productivity, developers’ skills, external software quality, process conformance, quasi-experiment, test-driven development
Fucci, Davide, Prosessin mukaisen toiminnan ja ohjelmistokehittäjien taitojen rooli testivetoisessa kehityksessä.
Oulun yliopiston tutkijakoulu; Oulun yliopisto, Tieto- ja sähköteknikan tiedekunta  
*Acta Univ. Oul. A 671, 2016*  
Oulun yliopisto, PL 8000, 90014 Oulun yliopisto

**Tiivistelmä**


**Asiakirjat:** ketetä ohjelmistokehitys, kontrolloitut koe, näennäiskoe, ohjelmistokehittäjien taidot, ohjelmistokehittäjien tuottavuus, ohjelmiston ulkoinen laatu, prosessin mukainen toiminta, testivetoisen kehitys.
Alla mia famiglia.
Acknowledgements

Completing a Ph.D. degree involves far more than simply writing a doctoral thesis—it is a journey. My Ph.D. journey gave me the chance to grow both intellectually and personally. I would like to thank all of the people who were there during this journey. First, I would like to thank my supervisors, Prof. Burak Turhan and Prof. Markku Oivo, who had the courage to say yes to my ongoing requests for a Ph.D. position in their department. I believe you have done a wonderful job in shaping me as a researcher. Thank you for the encouragement, the successes, and the failures.

Along with professors Turhan and Oivo, I owe much of this dissertation to Prof. Natalia Juristo and Dr. Hakan Erdogmus. Much of this work would simply not exist if it weren’t for Prof. Juristo’s ESEIL Fi.Di.Pro (Finland Distinguished Professor Program). Thank you, Prof. Juristo, for all the ideas we exchanged during our Spanish/Italian/English jabbering. One of Prof. Erdogmus’s papers is the reason I originally began my research on this topic. He has been there despite being half a world away to clarify my doubts and to help me in polishing my ideas. I later had the opportunity to visit him in person at Carnegie Mellon University Silicon Valley. Sitting in the same room with him while we wrote and discussed the dashboard was one of the most formative experiences I have ever had.

I would like to extend a special thank you to members of my follow-up group: Dr. Ayse Tosun-Misirli, Dr. Jouni Markkula, and Prof. Samuli Saukkonen. Year after year, your feedback steered my research in the right direction. I would also like to thank Prof. Daniel Sundmark (Mälardalen University, Sweden) and Prof. Marco Torchiano (Politecnico di Torino, Italy), both of whom dedicated their time to review this thesis; their suggestions helped me to improve its quality. I would also like to acknowledge the EATDD Academy of Finland project, which I have been working for during the last two years, along with the Nokia Foundation, UniOGS, and ISACA Finland; I wish to thank all of these organizations for their support.

Part of the journey is also walking into the office every morning, knowing that you will be greeted by wonderful people. Thanks to (rigorously ordered by seniority) Pilar, Nebojsa, Sandun, Ovais, Ali, Lucy, Rahul, Itir, Adrian, Vitor, Woub, and Amine; with friends like you, the university’s food did not taste that bad after all. Together we built
some very good memories; that is why I wish all the best for each of you. I would also like to thank all of my other colleagues from the M-Group / M3S research group.

During my Oulu years I have met many people from all around the world, though the Italian team (Daniele, Fabio and Francesco) deserves a special thanks; perhaps you guys managed to escape Oulu too soon.

Another person’s Ph.D. journey crossed path with mine, thanks to my girlfriend Irene Lehto for listening to me nagging about this thesis, and thanks for letting me reciprocate it.

This Ph.D. journey of mine wasinterwoven with many other journeys back to my motherland. For this reason, I would like to thank my lifelong friends back home: Antonio, Fabio, Giuseppe, and Saverio were always there for me in the warmth of southern Italy.

Immense thanks go to my family. Mamma, lo so che assecondare la mia volontà di vivere così lontano da casa è stata dura per te. Per questo dedico questo lavoro e gli anni spesi qui al freddo a te e a papà. Vi voglio bene. Grazie.

Finally, at the end of this Ph.D. journey, I can say with all honesty that it was a great journey.

Oulu, March 2016 Davide Fucci
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANOVA</td>
<td>Analysis of Variance</td>
</tr>
<tr>
<td>ASD</td>
<td>Agile Software Development</td>
</tr>
<tr>
<td>AT</td>
<td>Acceptance Test</td>
</tr>
<tr>
<td>C3</td>
<td>Chrysler Comprehensive Compensation System</td>
</tr>
<tr>
<td>CONF</td>
<td>Process conformance</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>ITLD</td>
<td>Iterative Test-last Development</td>
</tr>
<tr>
<td>LOC</td>
<td>(Uncommented) Lines of Code</td>
</tr>
<tr>
<td>LOTC</td>
<td>(Uncommented) Lines of Test Code</td>
</tr>
<tr>
<td>NHST</td>
<td>Null-hypothesis Significance Testing</td>
</tr>
<tr>
<td>PROD</td>
<td>Developer’s Productivity</td>
</tr>
<tr>
<td>QLTY</td>
<td>External Software Quality</td>
</tr>
<tr>
<td>ROI</td>
<td>Return on Investment</td>
</tr>
<tr>
<td>RQ</td>
<td>Research Question</td>
</tr>
<tr>
<td>SLR</td>
<td>Systematic Literature Review</td>
</tr>
<tr>
<td>SQA</td>
<td>Software Quality Assurance</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>SWEBOK</td>
<td>Software Engineering Body of Knowledge</td>
</tr>
<tr>
<td>TDD</td>
<td>Test-driven Development</td>
</tr>
<tr>
<td>TEST</td>
<td>Unit-testing effort</td>
</tr>
<tr>
<td>TLD</td>
<td>Test-last Development</td>
</tr>
<tr>
<td>UT</td>
<td>Unit Testing</td>
</tr>
<tr>
<td>XP</td>
<td>eXtreme Programming</td>
</tr>
</tbody>
</table>
List of original publications

This thesis is based on the following articles, which are referred to in the text by their Roman numerals (I–V):


Contents

Abstract
Tiivistelmä
Acknowledgements 9
Abbreviations 11
List of original publications 13
Contents 15
1 Introduction 19
  1.1 Test-driven Development ........................................ 19
  1.2 Research Gap and Motivation .................................. 21
  1.3 Objectives and Research Questions .............................. 26
  1.4 Scope of the Research ........................................... 27
  1.5 Overview of the Research Design .............................. 28
  1.6 Dissertation Structure ......................................... 30
2 Background 31
  2.1 Agile Software Development and XP ............................. 31
  2.2 Unit Testing ....................................................... 33
  2.3 Test-driven Development ......................................... 34
    2.3.1 TDD Benefits ................................................. 37
    2.3.2 TDD Disadvantages ......................................... 38
  2.4 Empirical Studies on the Effects of TDD ......................... 39
    2.4.1 Effects of TDD on External Quality ........................ 40
    2.4.2 Effects of TDD on Productivity ............................ 41
  2.5 Limitations and Research Gap .................................. 43
    2.5.1 TDD Process Conformance .................................. 44
    2.5.2 Developers’ Skills and Experience ........................ 45
3 Research Design 47
  3.1 Research Design and Research Phases ......................... 47
  3.2 Overall Experimental Design .................................. 49
    3.2.1 Metrics .................................................... 49
    3.2.2 Sampling .................................................. 59
    3.2.3 Experimental Design Strategy .......................... 60

15
3.2.4 Data Analysis ......................................................... 62
3.3 Summary of the Research Approach .................................. 67

4 Original Research Papers ............................................. 69
  4.0.1 Author’s Contributions ........................................... 69
  4.1 Paper I–On the Role of Tests in Test-driven Development: a
        Differentiate and Partial Replication ........................... 71
  4.2 Paper II–A Replicated Experiment on the Effectiveness of Test-driven
        Development .......................................................... 73
  4.3 Paper III–Impact of Process Conformance on the Effects of
        Test-driven Development ............................................. 73
  4.4 Paper IV–Towards and Operationalization of Test-driven
        Development Skills: An Industrial Empirical Study ............ 76
  4.5 Paper V– A Dissection of Test-Driven Development: Does It Really
        Matter to Test-First or to Test-Last .................................. 78

5 Discussion and Limitations ............................................. 81
  5.1 Answer to RQ1: How does process conformance impact the claimed
        effects of TDD? ......................................................... 81
    5.1.1 Answer to R1.1: How does process conformance impact the
            claimed effects of TDD on software external quality? .......... 83
    5.1.2 Answer to R1.2: How does process conformance impact the
            claimed effects of TDD on developers’ productivity? .......... 83
  5.2 Answer to RQ2: How do developers’ skills impact the claimed effects
        of TDD? ................................................................. 84
    5.2.1 Answer to R2.1: How do developers’ skills impact the claimed
            effects of TDD on external software quality? .................. 85
    5.2.2 Answer to R2.2: How do developers’ skills impact the claimed
            effects of TDD on their’ productivity? ......................... 86
  5.3 Threats to Validity .................................................... 86
    5.3.1 External Validity .................................................. 87
    5.3.2 Internal Validity ................................................... 88
    5.3.3 Construct Validity ............................................... 89
    5.3.4 Conclusion Validity .............................................. 90

6 Conclusions and Future Work ........................................ 91
  6.1 Summary of Contributions .......................................... 93
    6.1.1 Relevance to Academia ......................................... 95
6.1.2 Relevance to Industry .................................................. 96
6.2 Future Work ............................................................... 97

References ................................................................. 99
Appendices ................................................................. 105
Original publications ................................................... 107
1 Introduction

Modern software development workflows are currently in the process of adapting to the demanding schedule and resources required by the software industry. Software must be maintained despite the rapidly changing requirements and the short development cycles. In terms of financial and societal impact, the 2015 edition of the Aalto University School of Science survey\(^1\) showed that the Finnish software industry was worth roughly 7.2 billion Euros in revenues that year, which had grown 11.4% compared to the previous year. The cost of software development activities accounted for around 31% of these revenues, which contributed to the erosion of profit margins to a mere 7%. The total cost of software development activities is currently approaching 2 billion Euros. Thus even a 1% improvement in software development productivity within the Finnish software industry would generate an annual savings of around 20 million Euros. The present value of long-term total savings from this small improvement rate when based on the last three years’ long-term interest rate trends in Finland would be roughly 500 million Euros, which could be shifted to innovation activities. A series of software development practices and methodologies—referred to as Agile software development (ASD)—are increasingly being adopted in industry (West et al. 2010) in order to attain such improvements. Accordingly, ASD has become an important topic of investigation for the software engineering research community, as demonstrated by the growing number of conferences and journal issues dedicated to the topic (Dybå & Dingsøyr 2008). ASD is based on the values expressed in the Agile Manifesto (Fowler & Highsmith 2001); in its essence, this approach envision close collaboration with the customers in order to develop a software solution in a short time frame. The evidence collected from organizations that use ASD has shown positive effects in terms of, for example, customer satisfaction, effort estimation, product defects, and learning (Dybå & Dingsøyr 2008).

1.1 Test-driven Development

With the advent of the Agile Manifesto Fowler & Highsmith (2001), the community established several practices. A set of twelve practices, commonly referred to as extreme

\(^1\)http://goo.gl/JPL297
programming (XP), embraced the ASD values. Test-driven Development (TDD) is one of the practices that was integrated into XP in the late 1990s—although NASA was using it already in the 1960s (Beck 2002).

TDD is a programming technique that encourages code development by repeating short cycles consisting of (Beck 2002):

1. writing a unit test (UT) for an unimplemented functionality or behavior;
2. supplying the minimal amount of production code to make the test pass;
3. refactoring the production code;
4. checking that all tests still pass after the changes.

These result in, respectively: i) a shift from a test-last to a test-first approach; ii) developing only enough code to pass the tests; iii) focusing on design quality through refactoring; and iv) a library of regression tests. Thus, some researchers claim that TDD leads to better code quality and improves the developers’ confidence that the code will enhance his or her productivity (Astels 2003, Beck 2002).

Following the ASD values, TDD has been characterized in the practitioners’ literature as a development practice that aids the production of high-quality software solutions in short time iterations (Beck 2002, Astels 2003). If the promises of TDD were to be kept, it would have a major impact on the software industry. In 2014, the State of Agile Survey\(^2\) reported that 34% of the 3,501 IT professional surveyed worldwide claimed to use TDD.

A survey from 2007 (Begel & Nagappan 2007) involving 488 software developers, testers and project managers at Microsoft Corp. showed that 15% had used TDD, while another 10% were willing to adopt it into their development workflow. Nevertheless, TDD, along with pair programming, remains one of the least-used agile development methodologies (Begel & Nagappan 2007). The survey respondents perceived TDD as a factor that mainly contributed to higher code quality, while they ranked the overall benefits tenth among the perceived benefits brought about by ASD methodologies (Begel & Nagappan 2007). Similarly, Rodríguez et al. (2012) reported that only 1.7% of two hundred software-intensive Finnish companies use TDD.

Therefore, academics have been interested in the study of TDD to confirm whether the postulated effects are real and repeatable. The results regarding TDD’s effects are so far contradictory (Turhan et al. 2010).

Empirical studies on the effects of TDD have been synthesized in systematic reviews (Turhan et al. 2010, Munir et al. 2014, Rafique & Misic 2013, Sfetsos & Stamelos 2010, Causevic et al. 2011, Kollanus & Isomöttönen 2008). The secondary studies report ambivalent findings regarding the quality of software systems developed using TDD, as well as the productivity of the developers.

As with many business endeavors, software development involves three main factors: people, process, and product (Cockburn & Highsmith 2001). The focus of this research is on the people factor, i.e., software developers. In this regard, secondary studies report a lack of evidence regarding a) how software developers apply TDD in practice, and b) how developers’ experience and skills affect the application of TDD (Causevic et al. 2011). Very few primary studies (Müller & Höfer 2007, Madeyski & Szala 2007, Latorre 2014) focus on these points.

This thesis analyzes the role of a and b in terms of their effects on external software quality and developers’ productivity.

1.2 Research Gap and Motivation

Software companies are currently embracing ASD, including TDD. Nevertheless, the effects of TDD on external quality and developers’ productivity are not yet fully understood. The existing scientific literature shows inconsistent findings across empirical studies that compare TDD to traditional software-development approaches (see Chapter 2). At the same time, it is well understood that running software engineering experiments is made difficult by the large number of contextual and human factors (Basili et al. 1999, Tichy 1998). Before dismissing TDD as a fluke or embracing it as a panacea, any empirical investigation thus should focus on those factors that might hamper the observation of TDD’s effects.

Although an extensive list of possible factors that restrict the effects of TDD has not been compiled, the existing literature has converged on four main factors:

i) Developer’s skills

This factor refers to the existing skill differences that subjects who participate in studies on TDD might have. It is generally expected in the behavioral sciences that the differences between subjects are equally as important as the treatment that is being studied (Maxwell & Delaney 2004). Nevertheless, only three of the thirty-seven studies included in a meta-analysis on the effects of TDD (Rafique & Misic 2013), explicitly took developers’ experience and existing skills into account. Understanding how
developers’ skills can impact TDD will help its industrial adoption (Causevic et al. 2011).

**ii) Process conformance**

Process conformance is defined as the extent to which the subjects who participate in studies on TDD follow the TDD’s steps. In research fields that rely on experimentation, such as medicine, the study of process conformance—or adherence to the treatment, as it is referred to in the context of clinical trials (Cook & DeMets 2007)—is a factor embedded in the design and execution of empirical investigations (Cook & DeMets 2007, Bosworth et al. 2006). In the field of software engineering, few studies address process conformance (Zazworka et al. 2010, Sørumgård 1997, Silva & Travassos 2004, Zazworka et al. 2009, Basili 1993), and those that do use different names (e.g., correctness, fidelity, compliance are reported by Sørumgård (1997)). According to a systematic review (Munir et al. 2014) that combined forty-one primary studies, only two studies explicitly took TDD process conformance into account. Two secondary studies identified process conformance as one of the limiting factors in the industrial adoption of TDD (Turhan et al. 2010, Causevic et al. 2011).

**iii) Tasks’ size and complexity**

This factor refers to the characteristics of the task performed during the study, such as size, domain, and complexity. Rafique & Misic (2013) found that no empirical research has been conducted that would relate the task size factor to the observed effects of TDD, and only one study thus far has dealt with task complexity. Causevic et al. (2011) found that the nature of the task is also a critical factor for industrial TDD adoption. The ways in which the effects of TDD vary according to the type of task to which it is applied (e.g., legacy code, UI development) has thus far not been studied.

**iv) Supporting tools**

This factor refers to the tools (e.g., integrated development environments, testing frameworks) used during the application of TDD. It would appear that no previous studies have investigated such factors in the context of TDD. Ten primary studies included in Causevic et al. (2011) studied the issue of tooling, but only in terms of specific domains, such as interface or distributed system development.

Factors i and ii deal specifically with software developers as subjects of studies vis-à-vis the effects of TDD. Research gaps 1 and 2 (below) have identified the research
gap associated with each factor.

Research Gap 1: The question of whether or not developers’ skills play a role in explaining TDD’s effects (or lack thereof) on external quality and developers’ productivity has yet to be studied.

In studies involving human subjects, a measure of the relevant individual differences among the subjects is as important as their performance during the study (Maxwell & Delaney 2004). Failing to consider the pre-existing characteristics of the subjects can pose a threat to the conclusion validity (Wohlin et al. 2012). This gap will be filled by assessing how much the developers’ skills account for external quality and developers’ productivity.

Researchers have already studied the effects of subjects’ pre-existing differences in skills on the application of TDD. In a controlled experiment, Müller & Höfer (2007) compared expert and novice software developers who applied TDD; the experts achieved better thoroughness of the test suite, utilized less time to complete the experimental task, and were more conformant to the TDD cycle. The study of developers’ skills in (Müller & Höfer 2007) differs from this dissertation in the following points:

- Developers’ skills are equated with the subjects’ seniority, rather than being assessed with the use of questionnaires;
- The effects on software external quality are not covered;
- Developers’ productivity is simply assessed with the time necessary to complete the task;
- Process conformance is considered as a dependent variable.

Höfer & Philipp (2009) compares pairs of expert and novice software developers (i.e., using pair-programming) who apply TDD. The results contradict those of Müller & Höfer (2007) since the expert developers were slower in finishing the task. The authors argued that this was due to the time the experts spent in performing refactoring activities. In addition, they found no significant differences in terms of the thoroughness of the test suite and the experts’ conformance to the process. Höfer & Philipp (2009) differs from the study of developers’ skills in this dissertation in the following points:

- Developers’ skill is measured by labeling subjects as either students or professionals.
- The effects of developers’ skills are confounded with pair-programming;
- The effects on external software quality are not covered;
– Developers’ productivity is assessed simply by the time necessary to complete the task;
– Process conformance is considered as a dependent variable.

Madeyski (2006a) used TDD and pair-programming together in his study, but he found no correlation between external quality and skills. Madeyski (2006a) differs from the study of developers’ skills in this dissertation in the following points:
– In general it addresses factors relevant for pair-programming (pair experience and the feel-good factor), although in a TDD setting;
– The effects of developers’ skill are confounded with pair-programming;
– The effects on developers’ productivity are not covered.

Latorre (2014) investigated how the conformance to the TDD steps and the development time change at the different levels of seniority through a quasi-experiment spanning several weeks. The results showed that process conformance does not depend on skills, whereas efficiency does. In particular, novices showed the worst performance due to their unfamiliarity with refactoring and evolving architectures, while experts deviated from the process when necessary in order to maximize efficiency. Latorre (2014) differs from the study of developers’ skills in this dissertation in the following points:
– Developers’ skills are associated with the seniority of the subjects, rather than being assessed with the use of questionnaires;
– The effects on external software quality are not covered;
– Developers’ productivity is assessed simply with the time necessary to complete the task;
– Process conformance is considered to be a dependent variable (DV), as opposed to an independent variable (IV).

Finally, a few studies (Pančur & Ciglarič 2011, Madeyski 2010, Erdogmus et al. 2005) have concluded that the lack of investigation regarding the subject’s differences in terms of skills is one reason for the inconclusive results in TDD literature.

Research Gap 2: The question of whether or not developers’ conformance to the TDD process plays a role in explaining TDD’s effects (or lack thereof) on external quality and developers’ productivity has yet to be studied.
Within the medical research field, the study of experimental subjects’ adherence (and

24
non-adherence) to treatment has been the subject of much investigation. It is generally recognized that poor adherence to treatment contributes to the failure of medical interventions; the related costs for non-adherence are in the order of several hundred million dollars in the United States alone (Midence & Myers 1998).

Although not to the same extent as the medical field, adherence to a development process (i.e., process conformance) is of great interest for the software engineering research community. Process conformance is defined as *the degree of agreement between the software development process that is really carried out and the process that is believed to be carried out* (Sørumgård 1997).

Within the software engineering community itself, process conformance is referred to with different names depending on the context, e.g., conformance in Basili’s experience factory (Basili 1993), compliance in the context of SQA (Ince 1994), and fidelity in Humphrey (1995).

In the context of this dissertation, the specific process is TDD. In particular, we refer to the process conformance of software developers to the steps defined in Section 1.1.

Process conformance is of pivotal importance when assessing the threats to the construct validity of an experiment (Wohlin et al. 2012). When studying a process, it is necessary to verify that the process is performed as intended, in order for the knowledge achieved by following that process to be valid. Researchers have identified process conformance (or lack thereof) as one reason for inconclusive experimental results (Pančur & Ciglarč 2011, Müller & Hagner 2002). In this dissertation, the analysis of process conformance is done *a-posteriori* rather than by predicting it from other factors (e.g., as was done by Müller & Höfer (2007), and Höfer & Philipp (2009) in their experiments) as a procedure to assess the process itself. Under such settings, experimental subjects cannot be blocked in their individual conformance to TDD, but the level of conformance is put in correlation with other variables of interest (Sørumgård 1997). In this dissertation, this kind of assessment is done for the TDD process and its postulated effects. The interpretation of the results has led to identifying a possible flaw in how process conformance is assessed, showing that some components of the process are more important than others.

Causevic et al. (2011) identified the lack of process conformance as a limiting factor for the adoption of TDD in industry. The authors took a prescriptive stance regarding the problem of process conformance to TDD by finding and correcting a possible flaw in the execution of the process, i.e., the lack of TDD support for negative tests (Causevic et al. 2012, Causevic et al. 2013).
Considering research gap 1 and research gap 2, one of the motivations for this dissertation is that, although pre-existing skills and process conformance factors are acknowledged in the research community as being potentially interesting in the study of TDD, their effects remain understudied.

Research gap 3: Lack of software engineering experiments conducted in industry. Many researchers have argued that in order to form software engineering theories, professional developers should be observed in the context of their organizations; Votta (1994), among others, claims that experiments performed in industry produce more generalizable results.

A survey of the software engineering experiments conducted between 1993 and 2002 (Sjøeberg et al. 2005) found that although 27 out of 103 experiments reported the involvement of professional software developers, only one study clearly reported industrial settings, i.e., professional software developers working in their company’s contexts. A systematic literature review by Dieste et al. (2013) showed that in the decade 2003–2012, only eleven industrial experiments were published.

Rafique & Misic (2013) meta-analysis of the effects of TDD was based on twenty-five primary studies, although only eight were classified as industrial studies. Out of those eight studies, however, only three were experiments (Dieste et al. 2013); the remainder were considered case studies or surveys using quantitative analyses. Another motivation for this dissertation is therefore the need for experimentation in industry in order to collect evidence and to create a solid understanding of the effects of TDD.

1.3 Objectives and Research Questions

The objectives of this dissertation are formulated as follows:

1. Studying TDD by analyzing the process conformance aspect, i.e., how the technique is applied in practice.
2. Studying TDD by analyzing the developers’ skills aspect.
3. Focusing on the impact of these factors on software quality and developers’ productivity in academic and industrial settings.

Consequently, two research questions (RQ) drove the research:

RQ1 - How does process conformance impact the claimed effects of TDD?
**RQ1.1** - How does process conformance impact the claimed effects of TDD on external software quality?

**RQ1.2** - How does process conformance impact the claimed effects of TDD on developers’ productivity?

**RQ2** - How do the developers’ skills impact the claimed effect of TDD?

**RQ2.1** - How do developers’ skills impact the claimed effects of TDD on external software quality?

**RQ2.2** - How do developers’ skills impact the claimed effects of TDD on developers’ productivity?

RQ1 and RQ2 aim to fill the gaps identified in research gaps 1 and 2. The research questions investigate two factors that could help to understand the effects of TDD: process conformance and developers’ skills. The rationale of the research questions is to detect whether the observed effects of TDD on external quality and developers’ productivity can be subject to the differences in process conformance and subjects’ skills.

RQ1 and RQ2 also address research gap 3. The evidence necessary to answer the questions and sub-questions will be gathered through the design and execution of experiments, some of which take place in industrial settings.

### 1.4 Scope of the Research

The main focus of this work is to understand if and to what extent process conformance and subjects’ skills can impact the effects of TDD. Due to the limited temporal scope of the doctoral training, however, it was not possible to carry out a longitudinal study. Instead, the evidence was gathered in short-term laboratory experiments.

The findings are based on quantitative results from several experiments. Because no triangulation of evidence (e.g., following a multi-method research approach by Creswell & Clark [2007]) was performed, a qualitative investigation of a more explorative nature is beyond the scope of this dissertation. The experience gained while conducting the experiments, however, allowed for establishing a relevant course of action for future work.

The operationalization of the results (e.g., tools to support process conformance, or a checklist for assessing the experience of developers who are willing to use TDD) is
outside the scope of this dissertation, since the results need to be validated in further settings before being used: for example, for developing tools.

The particular factors under investigation—process conformance and development skills—were chosen according to the rationale presented in Section 1.2. The other factors, tools and task size/complexity, are beyond the scope of this dissertation.

The effects of TDD were scoped to external quality and developers’ productivity following the agile motto that advertises ASD and TDD as producing better code and product, in less time. Therefore, other possible effects of TDD are not in the scope of this dissertation. These include, but are not limited to, effects on internal quality (Pančur & Ciglarić 2011), test-case quality (Causevic et al. 2012), test thoroughness (Madeyski 2010), and ROI (Müller & Padberg 2003).

Finally, the scope of this research is limited to the point of view of software developers, although the results of this dissertation could constitute a baseline to be taken into consideration by other stakeholders, such as project managers.

1.5 Overview of the Research Design

This section introduces the research design, which will be further developed in Chapter 3; the design is based on empirical software engineering research principles (Shull et al. 2002). The research approach presented in this dissertation is to be considered of an explanatory or causal nature (Chalmers 1999).

The methodological focus is on experimentation. The experimental approach, following both controlled experimental and quasi-experimental methods, was employed in order to address the research problem during Phase 1 and Phase 2. The research was structured in a path consisting of the following four phases:

**Phase 0**: Identification of research opportunities. In this phase the relevant literature was surveyed to identify any research gaps and opportunities.

**Phase 1**: Experimentation in academia. A first set of controlled and quasi-experiments was conducted within academia. Specifically, these studies were replications of a previous study that investigated similar factors.

**Phase 2**: Experimentation within industry. A second set of experiments was conducted in cooperation with F-Secure, Ltd., and Company X\(^3\) in order to analyze how the factors impact the use of TDD in industry.

\(^3\)Due to a non-disclosure agreement, the real name of the company is not reported.
Phase 3: Research synthesis. The results from the previous two phases were synthesized in order to draw conclusions and outline implications.

This work comprises the contributions the author has made to the research in Phases 1-3, it consists of five original, peer-reviewed publications. Figure 1 presents an overview of the research process, the contribution of each publication, and the research gap addressed by each. A survey of the literature (see Chapter 2) was the starting point for identifying the research opportunities and gaps reported in Section 1.2. The output of Phase 1 consists of papers 1-3; the focus is to study these factors—and their effects on TDD—within academia. In particular, papers 1 and 2 analyze whether or not TDD offers benefits over a more traditional development approach, and they probe what role the skills factor plays. Paper 3 deals with the role of TDD process conformance and its effects. Phase 2 is covered by papers 4 and 5, both of which focus on the study of the same factors in industry. Paper 4 analyzes the effect of the developers’ skills and process conformance combined, using data gathered from industrial experiments conducted at
F-Secure, Ltd., and Company X. Finally, paper 5 delves deeper into the role of TDD process conformance using the data gathered from both industrial experiences.

1.6 Dissertation Structure

The thesis is organized into six chapters. Following this introductory chapter, Chapter 2 reviews the literature on the topic and provides essential definitions for positioning this investigation within the broader research arena. Chapter 3 presents the research framework; includes detailed descriptions of the experimental and quasi-experimental designs, as well as the description of the statistical procedures utilized in each design. Chapter 4 summarizes the original publications included in this dissertation; each contribution is framed according to the research questions that guide this work. Chapter 5 discusses the main results and limitations of the dissertation; it also answers the research questions. Finally, Chapter 6 discusses the implications of the main findings for this research and for industry, and it identifies future research opportunities.
2 Background

The work presented in this thesis is positioned in the areas of software development processes and agile software development methodologies.

Section 2.1 presents an overview of ASD and XP, which TDD has emerged within. Section 2.2 provides an overview of unit testing as the groundwork that TDD is built upon. Sections 2.3 and 2.4 present the concepts behind TDD, its process, and its claimed benefits and disadvantages, as well as providing a review of the empirical evidence on its effects on external quality and productivity. Finally, Section 2.5 summarizes the gap in the literature, focusing on the two main factors of interest: process conformance and developers’ skills.

2.1 Agile Software Development and XP

Agile software development (ASD) was established with the formulation of the Agile Manifesto in 2001 (Fowler & Highsmith 2001). The Agile Manifesto subscribes to the idea that traditional software development methodologies are not fit to the new dynamics of the market. The manifesto puts forward four values (Fowler & Highsmith 2001):

– Individuals and interaction over process and tools
– Working software over comprehensive documentation
– Customer collaboration over contracts negotiation
– Responding to chance over following a plan

The four principles appear to be in opposition to a heavy-process approach. Nevertheless, the original proposers of the Manifesto expressed the view that their new approach could be combined with traditional methods (Fowler & Highsmith 2001).

In response to the Agile Manifesto, a series of agile software development methodologies have been devised, including XP, Scrum, and Dynamic System Development (Highsmith & Cockburn 2001).

In particular, twelve practices that embrace the ideas of ASD. Although the practices included in XP were not new at the time, they were distilled and condensed from software engineering best practices by the US software engineer Kent Beck while he was working on the Chrysler Comprehensive Compensation System (C3) from 1996 until 1999 (Fowler & Highsmith 2001, Beck 2000).
Beck invited other prominent practitioners to work on C3 during those years. In particular, Ward Cunningham and Ross Jeffries helped Beck in refining and disseminating the XP practices (Fowler & Highsmith 2001, Beck 2000). During their experience on C3, the three engineers individuated twelve practices, divided into four areas of focus. Test-driven development is one practice under the fine-scale feedback area.

1. Fine-scale feedback
   - **Pair programming**: in this strategy, the code is produced by two developers working on the same computer at the same time. One is in charge of the big picture (e.g., the problem that they are trying to solve), while the other takes care of the implementation details (e.g., he or she ensures that there are no syntax mistakes).
   - **Planning game**: this is a meeting held between the development team and customers in which the requirements to be included in the short-term release of the software are discussed; the meeting also includes the plan for development activities.
   - **Test-driven development**: this is an approach to software development in which unit tests for a feature of the system are written before the production code; the approach also emphasizes the role of refactoring.
   - **Whole team**: this is the practice of considering the customer as a member of the team in order to leverage his or her domain knowledge.

2. Continuous process
   - **Continuous integration**: this is the practice of integrating the different versions of the source code several times during the day so that the team is always working on the latest version of the software.
   - **Design improvement**: this is the practice of improving the system’s architecture by refactoring the source code in order to make it simpler and more generic.
   - **Small releases**: this means deploying and delivering smaller features of the software system to the customer at a faster pace, rather than in bulk.

3. Shared understanding
   - **Coding standard**: this is a set of rules that the team should follow in terms of (for example) code formatting and development patterns to be followed or avoided.
   - **Collective code ownership**: in this system, every member of the team is responsible for the entire codebase, so that everyone is allowed to change it or fix it when necessary.
– **Simple design**: this is the idea that one should strive to find the simplest solution possible when implementing a new piece of code.

– **System metaphor**: this is the idea that the system’s naming convention should be derived from concepts in the application’s domain that everyone in the team, including the customers and managers, is familiar with.

### 4. Programmer welfare

– **Sustainable pace**: the developers should adhere to a forty-hour hours work week, and should not be required to work overtime.

Although introduced by XP, TDD’s popularity has remained constant over the years, and today is regarded as an independent practice. In Figure 2, the number of Google searches for Extreme programming and Test-driven development are plotted over the course of the last ten years.

![Fig. 2. Google searches (vertical axis) for TDD and XP over the years. Available from http://goo.gl/LTYvP9](http://goo.gl/LTYvP9)

#### 2.2 Unit Testing

TDD leverages a pivotal component of XP: unit testing (UT). One of the main activities in software development is software testing. Software testing has the goal of checking whether the code produced by the developers conforms to the expected behavior, given the requirements. Conversely, software testing is a way to find bugs, i.e., unexpected behaviors in the software (Board 1987). Software testing can be done manually or automatically, according to the level to which it is applied. In general, a software system should be tested at three levels (Runeson 2006):
**Unit:** testing functional requirements and properties of single unit of the system, such as classes or methods.

**Integration:** testing how the components or units interact.

**System:** testing the overall system, including non-functional requirements, usually from the business prospective.

Software testing per se is a wide research field that includes several techniques and tools. Here we focus on UT and its implication for TDD. A discussion of other levels of testing is left out, as it is beyond the scope of this dissertation.

Unit tests are often written and executed by the developers themselves, and are a fundamental feature of XP projects. The SWEBOK[^4] defines UT as:

> Unit testing verifies the functioning in isolation of software elements that are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of highly cohesive units. Typically, unit testing occurs with access to the code being tested and with the support of debugging tools. The programmers who wrote the code typically, but not always, conduct unit testing.

UT helps to catch problems in the codebase early enough so that they do not show up as bugs in the software once it is in use. In turn, the cost of fixing a bug after release is, according to some estimates, one order of magnitude higher than fixing it during development (Boehm et al. 1981, Shull et al. 2002).

At least two studies have examined the efficiency of UT in an industrial context at Microsoft. The practice of UT decreased by 21 both the number of defects found by software quality assurance and customer-reported defects (Williams et al. 2009). The highest resistance toward UT is linked to its effect on productivity; for example, both Williams et al. (2009) and Shull et al. (2002) reported a 30% increase in development time.

### 2.3 Test-driven Development

Test-driven development (TDD) is one methodology, proposed by Kent Beck within XP, that developed from the agile philosophy. TDD is an iterative and fast-paced software

[^4]: http://www.computer.org/web/swebok/v3
development process. A detailed sequence of the TDD steps presented in Section 1.1 is presented as follows:

**Step 0** - Break down a requirement into smaller features
**Step 1** - Write a unit test case that encompasses a feature
**Step 2** - Run the test case to check if such feature already exists (test case pass), or not (test case fails)
**Step 3** - Write just enough production code to make the test pass
**Step 4** - Run the test case to check that it passes
**Step 5** - If necessary, refactor the code (both in the test and in production) to improve and simplify the design
**Step 6** - Run the test case again to check that the refactoring did not break existing code

![TDD cycle diagram](image)

**Fig. 3. TDD cycle**

Figure 3 shows the TDD process workflow. Three different test execution activities can be observed. Each test run is preceded by an activity that implies writing code,
either test or production. The transition between these two associated phases is indicated by a grey arrow. The red and green arrows—exclusively originating from the test run activities—indicate that the test failed or passed, respectively. Each test run maps onto a different phase of the process. According to proponents of TDD, the duration of each iteration is between five and ten minutes (Beck 2002, Astels 2003).

The first phase consists of writing a failing test, referred to as the red phase. It corresponds to steps 1 and 2. The red phase concludes once: after executing the test, it fails. In Figure 3, the test-run activity associated with this phase is presented in red. In a greenfield system, i.e., a system that is being built from scratch, the first test run should already conclude this phase, since no production code that can be exercised by the test exists. This might not be the case with a legacy or “brownfield” system.

The green phase begins by writing the minimal amount of code necessary to make the tests pass (Beck 2002). This means that the developer should not spend too much time trying to write production-ready code, but rather in finding a quick solution that will make the tests pass. Beck (2002) identifies three strategies to tackle this phase:

- Obvious implementation
- Fake it
- Triangulation

The first strategy, obvious implementation, implies that, if the developer is confident about the solution necessary to make the test pass, or the solution is obvious, he or she should go ahead and write the necessary implementation. The second approach, faking it, presents the opposite situation: the developer does not have enough knowledge about the problem. In this case, the developer fakes the implementation by, for example, returning a constant. In other words, new tests—representing other examples of how the feature should work—are necessary. Finally, triangulation builds on two or more examples, for example, tests (as in the way in which radar systems use two or more points to triangulate the location of an object) that encapsulate their variability into a more general solution. The green phase ends once all the tests are passing. This phase is presented with a green box in Figure 3, and corresponds to steps 3–4.

The third phase, the refactoring or blue phase is optional. The goal during this phase is to improve the code that was written in the previous phases. This might be necessary because, for example, the strategies used during the green phase can produce duplication in the code. One or more refactorings (i.e., changes in code that do not change its external behavior) can be applied in this phase (Fowler 2002). It should be noted that
this phase can break the code, i.e., produce unintended changes in the behavior. Hence, it is essential to make sure that all of the tests still pass once the refactoring is complete. In Figure 3, the test run associated with this phase is presented in blue. The refactoring phase is composed of steps 5–6.

TDD is not regarded as a testing practice per se, but it requires knowledge of UT, as it represents the building block of the process. Instead, TDD is mainly regarded as a development and design practice that leverages unit tests. A set of regression tests is thus the by-product of TDD (Erdogmus et al. 2011).

2.3.1 TDD Benefits

Researchers and practitioners have claimed that the use of TDD benefits software development in several ways, as discussed below.

TDD improves developers’ productivity

This is a consequence of the improved developer confidence in the codebase, also referred to as the art of fearless programming (Jeffries & Melnik 2007). In this regard, TDD makes it easier for a developer to try new things on the codebase. This applies to refactoring and the addition of new features. When code is modified or added, running the regression test suite promptly reports the health of the codebase. Developers’ productivity is also improved by their better understanding of the codebase. In particular, properly named unit tests that explain the behaviour of a small chunk of the system aid code comprehension. The TDD approach is more satisfying for developers, since it guides them toward success, i.e., the green state where all tests are passing (Beck 2002, Astels 2003, Jeffries & Melnik 2007).

TDD improves the external quality of the software

TDD reduces the number of defects introduced into the codebase that will eventually manifest once the system goes into production. Due to the granularity of each TDD iteration, it is more difficult to introduce a bug when tackling a really small feature. Conversely, it should be easier to find, and fix, a bug. At the same time, the overall status of the system is controlled by running the regression tests often (Beck 2002, Astels 2003, Jeffries & Melnik 2007).

TDD improves the internal quality of the code

TDD’s test suite represents a safety net as the codebase evolves (e.g., when new features are implemented). A test suite is also necessary when refactoring. Hence, TDD offers the
developer the possibility to simplify the codebase, improve its architecture, and remove
duplication in a safe condition. At the same time, the improved comprehensibility of the

![Diagram of TDD effects]

**Fig. 4. Summary of the claimed effects of TDD. Adapted from Erdogmus et al. (2011)**

Figure 4 summarizes the claimed effects of TDD and the associated causes.

### 2.3.2 TDD Disadvantages

Causevic et al. (2011) and Turhan et al. (2010) have identified the shortcomings of
TDD, as follows.

**Application domain**

Some domains, or components, of a system are more difficult to test than others
(e.g., graphical user interfaces). Some domains, such as embedded systems or highly
decentralized systems, are not suited for incremental testing; the size of the project that
TDD is applied to, and the project’s complexity, also represent hurdles in the adoption
of the practice. In addition, the use of TDD with legacy systems can be complex, since
the system needs to be rewritten to make it more testable.

**Discipline**

On the cognitive side, TDD is among the most difficult agile techniques, due to its steep
learning curve. It requires inverting the approach to development that the developer is
used to; maintaining the rhythm of the process cycle requires a substantial cognitive
effort. Adapting to this way of working requires time, mainly for those developers who
are used to the traditional TLD approach. On the other hand, TDD can be an addictive
practice that can be difficult to relinquish.

**Interaction with other practices**

Two practices are interwoven in TDD: software development and software testing. Depending on the company culture, separate figures take care of such activities. Once
they adopt TDD, the developers should have a certain level of skills and experience
necessary to develop and maintain the test suite. It is not clear how TDD interacts with
other practices. For example, integrating TDD with upfront design has shown positive
effects in studies, whereas the use of pair-programming has not shown any substantial
improvements. Nevertheless, there may be practices that inhibit the effects of TDD.

### 2.4 Empirical Studies on the Effects of TDD

The focus of this review chapter is on the effects of TDD on external quality and
productivity.

The existing body of knowledge, between 2000 and 2014, has been covered by five
systematic literature reviews (Turhan et al. 2010, Munir et al. 2014, Causevic et al.
2011, Kollanus 2010, Sfetsos & Stamelos 2010), one meta-analysis (Rafique & Misic
2013) and one integrative literature review (Mäkinen & Münch 2014).

The primary studies included in the review studies are controlled experiments or
comparative case-studies in which TDD is compared to another development technique
as a baseline. The baseline techniques correspond to either a traditional software
development activity—in which tests are written once one or more features have been
implemented (i.e., TLD)—or a test-last type of activity, in which production code and
unit tests are closely interpolated (ITLD).

Table 1 summarizes the secondary studies reviewed in this chapter.
Although a portion of the primary studies included in the reviewed secondary studies overlap, the set of collected studies offers a comprehensive summary of the relevant literature.

Table 2 reports the secondary studies regarding external quality and productivity, as well as the identified limitations and research gaps.

<p>| Table 1. Summary of the secondary studies context. Note: CE = Controlled experiment; CS = Case study; MA = Meta-analysis; SLR = Systematic literature review. |</p>
<table>
<thead>
<tr>
<th>Source</th>
<th>Type</th>
<th>Primary studies</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turhan et al. (2010)</td>
<td>SLR</td>
<td>32 - CE</td>
<td></td>
</tr>
<tr>
<td>Munir et al. (2014)</td>
<td>SLR</td>
<td>41 - CE - CS - Survey</td>
<td></td>
</tr>
<tr>
<td>Rafique &amp; Misic (2013)</td>
<td>MA</td>
<td>25 - CE</td>
<td></td>
</tr>
<tr>
<td>Mäkinen &amp; Münch (2014)</td>
<td>Integrative review</td>
<td>19 - CE - Case study</td>
<td></td>
</tr>
<tr>
<td>Kollanus (2010)</td>
<td>SLR</td>
<td>40 - CE - CS - Survey</td>
<td></td>
</tr>
<tr>
<td>Stetsos &amp; Stamelos (2010)</td>
<td>SLR</td>
<td>46 - CE - CS - Mixed</td>
<td></td>
</tr>
<tr>
<td>Causevic et al. (2011)</td>
<td>SLR</td>
<td>48 - CE - CS</td>
<td></td>
</tr>
</tbody>
</table>

2.4.1 **Effects of TDD on External Quality**

Turhan et al. (2010) analyzed controlled experiments that assessed the effects of TDD that were published between 2000 and 2008. The synthesis of the thirty-two trials included in the primary studies showed that TDD brought improvement over TLD—e.g., when tests were written after the feature(s) had been implemented—in terms of the number of defects present after development. When considering the entire pool of primary studies, the evidence was strongly in favor of TDD; the results were not as clear when considering only the more rigorous studies.

Munir et al. (2014) reviewed forty-one studies, assessing their rigor and relevance. The result from studies that had high rigor and high relevance markedly showed that TDD was beneficial for external quality. The same result held for studies with high
relevance but low rigor. The result from the category that included rigorous but low-relevance studies showed no difference between the TDD and TLD. The low-rigor, low-relevance studies’ results were divided between being inconclusive and being in favor of TDD. When the studies were considered regardless of their relevance and rigor, the results were inconclusive.

Rafique & Misić (2013) gathered twenty-five primary studies; the authors divided the studies according to setting (industrial vs. academic) and the control to which TDD was compared. In particular, they observed two different treatments: “waterfall” (the traditional approach to software development, in this chapter referred to as TLD) and iterative test-last (ITT), in which small features are first developed and then tested. The result in which academic and industrial studies were compared showed improvements when TDD was applied to industrial projects. When moderator variables were taken into account, however—for example, to account for the difference in task size—no substantial effect was due to TDD. Academic studies that employed a TLD control group were reported to yield larger improvements in quality, whereas such results did not hold for industrial studies, in which neither technique had a significant impact on quality. Nevertheless, the overall results showed minor improvements due to TDD.

Mäkinen & Münch’s review (2014) concluded that TDD was associated with a reduction of defects. Interestingly, they also reported results from more qualitative studies, involving the final users’ judgment of the software. The authors found that TDD did not affect the quality of the product as perceived by users.

Out of the forty studies reviewed by Kollanus (2010), twenty-two report results were related to external quality. The author found weak evidence in support of TDD. When only controlled experiments were taken into account, however, the results were contradictory.

Sfetsos & Stamelos’s systematic review (2010) of agile development techniques showed that TDD was the main driver for improvements in external quality. In particular, the defect rate was reduced from 5%-45%, and up to 50%-90%. The results were consistent between controlled experiments and case studies.

### 2.4.2 Effects of TDD on Productivity

Turhan et al.’s study (2010) showed no particular effect of TDD on productivity when all the available evidence was considered. TDD appeared to improve productivity only
when controlled experiments are taken into account; productivity was found to decrease in industrial studies.

Based on the nine studies with high rigor and high relevance (none of which were controlled experiments), Munir et al. (2014) concluded that productivity was not affected, or slightly declined. For higher-relevance studies (i.e., studies in industry), the authors found TDD to be detrimental to productivity. The more rigorous studies (nineteen experiments, one case study) did not offer substantial evidence either for or against TDD.

In Rafique & Misic (2013) found that the effect of TDD on productivity seemed to be negligible overall. The authors found a substantial drop when comparing studies in an industrial context with those in academia. The authors’ explanation for this was that expert developers (those who are more likely to take part in industrial experiments) put more effort into testing and refactoring activities. Finally, they found the largest drops in productivity in experiments that utilized a waterfall process, both in academia and industry. According to the authors, this might be due to the overhead in testing of TDD.

Mäkinen & Münch (2014) concluded that the negative effect of TDD relates to effort and productivity. They argued that while such results are not necessarily the outcome of the application of TDD, hidden factors may affect productivity.

Kollanus (2010) summarized the evidence from controlled experiments, although the results were unclear: some studies showed positive evidence, while others were negative or inconclusive. The case-study results, on the other hand, seemed to agree that TDD was detrimental to productivity. The author identified the origin of the loss in productivity to be TDD’s focus on UT, which inherently requires more effort; he argued that the effort put into testing might pay back in terms of quality (Kollanus 2010).

Sfetsos & Stamelos (2010) presented contradictory results in their study. Interestingly, one case study that was included in their primary studies found that when TDD was introduced in a company, development costs decreased, due to the reduction in faults in the long term (Damm & Lundberg 2006).

Finally, Causevic et al.’s review (2011) identified the increase of development time (closely related to productivity) as one limiting factor in the industrial adoption of TDD. The authors acknowledged that a trade-off might exist, since an initial loss in productivity could lead to beneficial effects in the long term.
Table 2. Summary of the secondary study results.

<table>
<thead>
<tr>
<th>Source</th>
<th>Effect on quality</th>
<th>Effect on productivity</th>
<th>Research gap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- Positive (High rigor)</td>
<td>- No effects (industrial)</td>
<td>- Process conformance (industrial)</td>
</tr>
<tr>
<td>Turhan et al. 2010</td>
<td>- No effect</td>
<td>- No effects</td>
<td>- Few industrial long-term studies</td>
</tr>
<tr>
<td>Munir et al. 2014</td>
<td>- Positive (high rigor, high relevance)</td>
<td>- Negative (high relevance)</td>
<td>- Impact of ITL - Impact of internal quality on external quality/productivity - Long-term studies - Process conformance</td>
</tr>
<tr>
<td>Rafique &amp; Misic 2013</td>
<td>- Slight positive (industrial)</td>
<td>- No effect (industrial)</td>
<td>- Effects on maintainability</td>
</tr>
<tr>
<td></td>
<td>- No effect (accounting for task size)</td>
<td>- Negative (vis-à-vis waterfall)</td>
<td>- Testing skills and experience - Process conformance</td>
</tr>
<tr>
<td>Mäkinen &amp; Münch 2010</td>
<td>- Positive (defect reduction)</td>
<td>- Negative</td>
<td>N/A</td>
</tr>
<tr>
<td>Kollanus 2010</td>
<td>- Slight positive (experiment)</td>
<td>- Inconclusive (experiments)</td>
<td>- Impact of UT</td>
</tr>
<tr>
<td>Sfeftos &amp; Stamelos 2010</td>
<td>- Positive</td>
<td>- Inconclusive</td>
<td>N/A</td>
</tr>
<tr>
<td>Causevic et al. 2011</td>
<td>N/A</td>
<td>- Negative (development time)</td>
<td>- Testing skills and experience - Process conformance</td>
</tr>
</tbody>
</table>

2.5 Limitations and Research Gap

The secondary studies reviewed in this chapter also surveyed some of the limiting factors associated with TDD. This dissertation focuses on two particular factors: process conformance and developers’ skills. This section reports the literature findings, with a focus on TDD.

43
### 2.5.1 TDD Process Conformance

TDD process conformance is defined as the extent to which a developer follows the TDD cycle (Kou et al. 2010).

Turhan et al. (2010) acknowledged that the majority of the primary studies that were included in their review did not report nor take into account the role played by process conformance. Although they claimed that process conformance should be used and discussed as a factor that may affect the results of TDD studies, they also noted that a high level of process conformance might not be reflected in the results (e.g., improved external quality). Hence, an appropriate level of process conformance might depend on other contextual factors.

Munir et al. (2014) also agreed on the importance of studying process conformance and its confounding effects on TDD studies. They reported that very few studies formally address process conformance, and only one study (Müller & Hagner 2002) fell under the “more rigorous and relevant” category. In particular, primary studies showed that process conformance had a positive impact on external quality and developers’ productivity (Müller & Hagner 2002).

Rafique & Misic (2013) considered process conformance to be one of the criteria for identifying rigorous primary studies. The authors showed that very few studies took process conformance into account in their analyses, or adopted techniques to enforce it during the course of the study. The authors argued that process conformance can be associated with the developers’ experience and skill level. More experienced developers are better able to follow the process, which in turns produces positive effects on quality. On the other hand, the extensive focus on the process comes at the expense of productivity (Rafique & Misic 2013).

Finally, Causevic et al. (2011) identified the “insufficient adherence to the TDD protocol” as one of the limiting factors for industrial TDD adoption. They reported evidence from industrial case studies in which TDD was abandoned after initial adoption due to time pressures, lack of perceived benefits, and difficulty in applying it throughout the whole system. Further evidence associated a low level of conformance with a low level of quality. The authors acknowledged that none of the studies focused on establishing a cause-effect between TDD process conformance and its effects (Causevic et al. 2011).
2.5.2 Developers’ Skills and Experience

From the description of the TDD cycle in Section 2.3, it is evident that TDD requires not only development, but also testing and refactoring skills and experience (Beck 2002, Astels 2003).

Causevic et al. (2011) identified the lack of testing skills as one limiting factor for TDD adoption in industry. Hence, the developers must have sufficient ability to write effective test cases.

Turhan et al. (2010) used the skill factor to categorize the rigor of the primary studies that they included in their systematic review. Although their findings showed that experience and skills allowed an easier adoption of TDD, they did not report which skills were the most relevant.

Similarly, Munir et al. (2014) considered skill and experience in order to characterize the primary studies’ context. They did not report results on the role of skill, however, or its impact on TDD.

In the same vein, Kollanus (2010) argued that developers’ skills must be taken into account when studying TDD, since omitting this factor poses serious limitations to the study validity. Rafique & Misic (2013) used information about developers’ skills and experience, reported in primary studies, as a moderating factor when analyzing the effects of TDD. In particular, only three studies explicitly dealt with such factors. In these studies, the results conflicted with one another and were unreliable due to the scarcity of observations. Although they could not identify a relationship between the skills of developers and the effects of TDD on quality, the authors suggested that skills could play a role in decreased productivity, as shown when considering students (novices) versus professionals (experts). They argued that skilled developers tend to spend more time on activities such as refactoring, whereas students—who may not possess the necessary skillset—tend to skip that phase. In the broader view, the fact that skilled developers tend to have high levels of conformance to the TDD process explains the drop in productivity.
3 Research Design

The work introduced in this dissertation follows an explanatory empirical approach. The main approach is experimental, using both controlled and quasi-experimental designs. The motivations for the design include the following:

1. “Explanatory study” implies that the research aims to explain rather than describe a phenomena. Traditionally, explanatory research is of a quantitative nature—although it has been applied to various types of qualitative research—since it typically involves the testing of prior hypotheses through quantitatively measuring the variables of interest. The data is usually analyzed using statistical techniques. As for the case of TDD, the theories reported regarding its effects have not been demonstrated (Creswell & Clark 2007).

2. This work aims to identify the nature of the relationship between TDD and external quality and developers’ productivity, and how other constructs interact with that relationship. A quantitative analysis can provide systematic evidence of such relationships (or lack thereof).

3. The settings necessary for the experiment’s approach were available in the form of academic courses in topics relevant to this research, as well as industrial partners interested in the same topics.

3.1 Research Design and Research Phases

The work presented in this dissertation was carried out in four phases, as presented in Chapter 1.

**Phase 0:** Identification of research opportunities.

The first step in any research endeavor is to identify a practical and relevant problem. This phase thus identified the related claims, the constructs used by the research and practitioner community interested in TDD (as well as their relationships), and the salient factors that constitute the research gap. The hypotheses that guided the experimental work were formulated during this phase. It should be noted that the literature on the topic was continuously reviewed during the subsequent phases in order to keep them up-to-date.
Phase 1: Experimentation in academia
In Phase 1, the claimed effects of TDD and the role of factors that could impact such effects were analyzed within academia. The context for Phase 1 was academic courses offered within the Information Processing Science curricula at the University of Oulu. Specifically, the experiments were embedded within the Software Quality and Testing course, which provided a good context for the study of TDD. The course touched on relevant topics such as unit testing, testing patterns, and TDD; it also included tutorial and practical classes that allowed the subjects to practice what they learned. The author was responsible for the course’s practical sessions. The data was collected in three replications during the autumn terms of 2011-2013.

Phase 2: Experimentation in industry
In Phase 2, the claimed effects of TDD (and the role of factors that could affect such effects) were analyzed within industry. The context for Phase 2 was the “Empirical Assessment of Test-driven Development” project, funded by the Academy of Finland—in partnership with F-Secure, Ltd., and Company X—in which the author was involved during the years 2012-2015. In both companies, the experiments were embedded in a series of workshops on UT and TDD that took place at the companies; respective premises. Three experiments were conducted between October and December 2013 at three locations of F-Secure, Ltd., and one experiment was conducted in March 2014 at Company X.

Phase 3: Research synthesis
The findings were synthesized inductively, since they were based on a concordance of the evidence that was collected in the experiments. Because the findings are a generalization of the evidence, they are considered to be the most probable findings given the available evidence. That is, since unaccounted-for factors are involved in explaining the effects of TDD, these findings cannot be guaranteed to hold true in all contexts.

In this dissertation, RQ1 is answered using the empirical evidence provided in papers III, IV, and V, whereas RQ2 is answered using the empirical evidence provided in papers I, II, and IV. The different contexts (academia and industry) were considered when synthesizing the findings.
3.2 Overall Experimental Design

This section discusses the constructs that were used (and the metrics used to measure them), as well as the sampling techniques, experimental design strategies, and data analysis techniques that were used during the course of the research.

The research process was designed according to the guidelines for empirical studies in software engineering set forth by Shull et al. (2008). The research method is experimental; experiments are fundamental to scientific and engineering endeavors. Although such experimentation has been used for centuries in fields like medicine and physics, it is important for a dynamic research field such as software engineering to have a way to measure the benefits of (for example) tools or processes in a systematic way (Wohlin et al. 2012). The experimentation paradigm employed in this research is explanatory, since the main concern is to quantify differences and relationships between constructs associated with a specific software development process: test-driven development. In other words, the goal of explanatory research is to prove the existence (or lack thereof) of cause-effect relationships (Wohlin et al. 2012). This type of study is called “fixed design” since factors of interest are known before the research begins, as opposed to qualitative research, in which interesting factors emerge alongside the researchers’ endeavors. Fixed-design research is also referred to as “quantitative research” (Creswell & Clark 2007). It is possible for quantitative and qualitative research to investigate the same phenomena but from different points of view. For example, experiments are valuable to quantifying differences, whereas qualitative methods are necessary in trying to understand the source of such differences (Wohlin et al. 2012). This research employed a fixed design involving the salient factors identified in the body of knowledge on TDD (external quality, developers’ productivity, process conformance, and developers’ skills) in order to identify the relationships between these factors. Table 3 provides a summary of the constructs, the associated metrics, and the types of variables the constructs represented in the experiments.

3.2.1 Metrics

The experiments conducted during the doctoral research relied on several constructs, which represented the dependent and independent variables under consideration. The metrics that were used to quantify the constructs were adopted from the existing literature; others were developed from scratch, due to the novelty of the specific study.
Table 3. Summary of the constructs and associated metrics.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Metric</th>
<th>Type of variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developers’ productivity</td>
<td>PROD</td>
<td>Dependent</td>
</tr>
<tr>
<td>External Software Quality</td>
<td>QLTY</td>
<td>Dependent</td>
</tr>
<tr>
<td>Testing productivity</td>
<td>TEST</td>
<td>Dependent</td>
</tr>
<tr>
<td>TDD process conformance</td>
<td>CONF</td>
<td>Independent</td>
</tr>
<tr>
<td>Developers’ skills in programming language</td>
<td>JAVA</td>
<td>Independent</td>
</tr>
<tr>
<td>Developers’ skills in unit testing</td>
<td>UT</td>
<td>Independent</td>
</tr>
<tr>
<td>Granularity of the development cycles</td>
<td>GRA</td>
<td>Independent</td>
</tr>
<tr>
<td>Uniformity of the development cycles</td>
<td>UNI</td>
<td>Independent</td>
</tr>
<tr>
<td>Sequencing of the development cycles</td>
<td>SEQ</td>
<td>Independent</td>
</tr>
<tr>
<td>Refactoring development cycles</td>
<td>REF</td>
<td>Independent</td>
</tr>
</tbody>
</table>

Specifically, the metrics that were used were a numerical representation of either the property of a software system, or the process that was used to develop such a system. For each metric, this dissertation reports on the rationale behind its usage, along with the formula and the necessary instruments or artifacts necessary to calculate it. The material for calculating the metrics is included in a lab package; see Appendix 1.

**Developers’ productivity**

The productivity of a developer is usually calculated as the output produced over a certain unit of time (Ramírez & Nembhard 2004). Usually, in software engineering, the output associated with the effort of a developer is some measure of the source code produced (Boehm et al. 1976). Traditionally, the number of lines of code (LOC) written was used to measure effort, due to its simplicity. Researchers have demonstrated, however, that LOC is not a good representation of the amount of effort that a developer puts into a task; this might be due to several factors, from the language and integrated development environment (IDE) used to the kind of development task that the developer carries out (Shihab et al. 2013). Previous literature on XP methodologies has used “user stories” to measure effort (Madeyski 2006b, 2010). A user-story is a short description of what a system should do, for whom, and why. Due to their ambiguity, user-stories are usually presented together with one or more examples. A better quantification of the effort is, for example, the number of user-stories completed by a developer (Beck 2000). This, in turn, poses the question of how to define a user-story as completed.
Madeyski (2006a, 2010) and Erdogmus et al. (2005) have suggested acceptance tests to do this; these are usually associated with each user story: for example, by executing the examples given with a user story in the form of a unit test. Hence, a user story is considered to be complete once all the tests (or a given threshold of the tests) associated with the user story pass. Following this approach, for the current study, the researcher divided the development tasks used during the experiments into user stories. The acceptance tests associated with the user stories were not disclosed to the subjects in order not to bias them, and to avoid the situation where they would write code that forcefully passes the tests. Finally, in order to calculate the metric associated with productivity (Formula 1), the number of completed user stories (\(d.u.s\)) was divided by the time, in minutes, the subject took to complete the user stories (\(t\)).

\[
PROD = \frac{d.u.s.}{t}
\]  

A shortfall of the metric as previously formulated is that it does not take the total size of the system into account. Looking at the value for \(PROD\), it is not possible to tell how large (or small) such a value is compared to the whole system. In other words, it is not clear which parts of the system were covered during the development. This is not an issue for controlled experiments, since all of the subjects are working on the same task; thus the relative size of the task is not important. For other experimental designs, in contrast—for example, repeated measures within subjects—it is valuable to have a metric that factors in the relative size of the task. The reason for this is that, for such designs, the measurements are taken from different tasks, each with a fixed duration; hence the size (or complexity) of the task matters (Bergersen et al. 2011). Therefore, another metric for productivity, \(PROD_{(user stories coverage)}\), is defined in Formula 2.

\[
PROD_{(user stories coverage)} = \frac{d.u.s.}{u.s.}
\]  

where \(u.s\) indicates the total number of user stories comprising the system; therefore this measure of \(PROD\) now indicates which percentage of the whole system a subject was able to complete; it is in the range \([0, 1]\).

**Instruments and Operationalization**

In order to calculate the \(PROD\) metric, the necessary artifacts are:

- The source code developed during the experiment,
- The acceptance test suite associated with each user story,
– The recording of the time needed by the given participant to complete the task.

The task is divided into user stories; each user story has a set of acceptance tests associated with it. The steps necessary to calculate the metrics are:

1. Run the acceptance tests against the provided source code.
2. Count the user stories for which the completed criteria are met (e.g., if the passing tests are above a given threshold)
3. Divide the number of completed user stories by the time the subjects used to turn in the experimental task; alternatively (depending on the study design), the total number of user stories may be used.

**External Software Quality**

The quality of a system can be measured at several levels (Boehm 1978). From a business perspective, for example, quality is associated with the value the system generates. From a user-interaction point of view, quality is associated with the system’s usability and understandability for the final user (Bevan 1995). Another level of quality is usually referred to as external quality; this represents to which extent a system matches its requirements, or whether it presents defects once put into use. Yet another level of quality, internal quality, is evaluated at the source-code level; this represents how maintainable, extensible, and comprehensible the codebase is (Basili et al. 1996).

The experiments for this dissertation focus on the external quality of the system developed by the participating subjects. Assessing the defects of the system manually for the dozens of variations of the system produced during the experiment is not optimal, however, since it involves the human factor (for example, experts who use the system and check for possible defects, which might inject unwanted variability into the measurements), it is time-consuming, and it not reproducible by others. The discovery of defects, however, can be partially automated using acceptance unit tests implemented by an expert in the system’s domain. In this case, the test “oracle” is represented by the specifications for the system, written in the form of acceptance tests (Marciniak & Shumskas 1994). The external quality is calculated using a defect-based approach that leverages the existing acceptance tests associated to the user stories. The metric proposed for quality is similar to the one reported in Erdogmus et al. (2005). The value of the quality metric is the sum of the quality of each user story ($QLTY_i$) over the total
number of user stories in the task ($\#u.s.$), as in Formula 3.

\[
QLTY = \frac{\sum_{i=1}^{\#u.s.} QLTY_i}{\#u.s.}
\]  

(3)

In turn, the quality of a single user story is the ratio of acceptance tests that pass over the total number of acceptance tests associated with the user story (see Formula 4). The measure is a ratio in the range [0,1]. Therefore, the metric for the $QLTY$ is also in the range 0,1.

\[
QLTY_i = \frac{\#\text{assert}_i(\text{Pass})}{\#\text{assert}_i(\text{Total})}
\]  

(4)

One shortfall of the metric as previously defined is that all of the user stories are assumed to have the same level of difficulty, so scoring a high value for a complex user story is the same as scoring the same value for a simpler user story. In the studies, the user stories were organized so that they would have the same level of complexity. When that is not possible, one approach is to assign a difficulty score to each user story and weight it in the calculation of the metric (Erdogmus et al. 2005).

**Instruments and Operationalization**

In order to calculate $QLTY$, the necessary artifacts are:

- The source code developed during the experiment;
- The acceptance test suite associated with each user story.

The experimental task is divided into user stories and it is associated with a set of acceptance tests. The following steps are necessary to calculate the value for $QLTY$:

1. Run the acceptance test suite against the source code provided;
2. For each user story, calculate its quality by dividing the number of passing tests for the total tests associated with that user story (see Formula 4);
3. Add the qualities of each user story and divide by the total number of user stories in the task (see Formula 3).

**Testing productivity**

The metric represents the effort necessary for the developer to devise unit tests for the system. Testing productivity involves the quantification of the amount of unit-testing activities done by the subjects. Such effort could be intuitively calculated with a metric such as LOTC (Lines of Test Code), but, in terms of productivity, such a metric has the
disadvantage of being dependent on (for example) the programming language or the IDE that is used. A better and simpler representation of the testing productivity is to use the actual number of test methods written during the development of the system. Using such a coarse-grained measure can be counterproductive, however, since unit-test methods may be empty or only “stubbed” without any significant purpose. Rather than using “counting the unit test” methods, the count of “assert” statements was thus selected, because a valid unit test method should include at least one assert statement. The assert statements were visually inspected for validity; the futile ones were not included in the count. Finally, the number of valid assert statements was normalized by the time the subject took to complete the task in minutes $t$, as presented in Formula 5.

$$\text{TEST} = \frac{\#\text{assert}}{t}$$ (5)

The metric for testing productivity is a ratio that varies in the range $[0, \infty]$ and follows the metric used in previous experimentation (Erdogmus et al. 2005).

**Instruments and Operationalization**

The necessary artifacts for calculating this metric include:

- The source code developed during the experiment;
- The recording of the time needed by the given participant to complete the task.

The following steps are necessary to calculate the metric for testing productivity:

1. Count the number of assert statements in the source code provided by the participant;
2. Divide the number of assert statements by the time used to turn in the task, in minutes (see Formula 5).

**TDD process conformance**

The concept of conformance underlies the idea of comparing an ideal model, in this case a representation of a process, with the actual process that is happening in the real world. Hence, the more the real-world process overlaps with the ideal one, the more conformant it is. The metric used to define process conformance represents the extent to which the experiment’s subjects followed the TDD cycle during the development of the experimental task. The metric is based on a model that represents the ideal

---

5 An assert is a method, usually provided by the testing framework, to verify a given condition.
TDD cycle. A set of heuristics developed by Kou et al. (2010) was used to check whether the developers’ development activities adhered to the red-green-refactor process typical of TDD (see Section 2.3). The heuristics leverage a series of development activities tagged as test-first, test-last, or others (e.g., regression). Kou et al. (2010) validated the heuristics by comparing the judgment of expert TDD developers eighteen in academia, ten) to the automated evaluation performed following the heuristics. The heuristics’ accuracy was between 85 and 90%. The metric used to measure TDD process conformance is therefore the number of activities tagged as TDD conformant \(#activity[TDD]\) over the total number of activities \(#activity[Total]\), as in Formula 6. The metric \(CONF\) (Formula 6) is a ratio in the range \([0,1]\).

\[
CONF = \frac{#activity[TDD]}{#activity[Total]}
\]  

(6)

Instruments and Operationalization

The necessary artifacts to measure process conformance are:

- The sequence of development activities registered by the IDE;
- The heuristics for categorizing sequences of activities.

The two phases—gathering the developers’ activities within the IDE, and categorizing them—are automated by a tool (Becker et al. 2015). Specifically, the tool is an Eclipse IDE plugin 6, which records the activities inside the IDE and outputs a list of the activities’ sequence, with their recognized categories. For example, the sequence below is categorized as a non-conformant TDD sequence.

1. CREATECLASS HelloWorld.java
2. CREATEMETHOD HelloWorld.sayHello()
3. EXECUTE HelloWorld
4. CREATECLASS HelloWorldTest.java
5. CREATEMETHOD HelloWorldTest.testSayHello()
6. RUNTEST HelloWorldTest OK

In the sequence, where each activity is sorted by timestamp, the test is created and executed after the method that is being tested. The tool creates a file that reports the

6http://github.com/dfucci/besouro
sequence of activities and their associated categories. The participants, together with the source code, turn in the file at the end of each experiment session. The metric is calculated by dividing the number of TDD-conformant activities present in the list by the total number of activities in the list, as in Formula 6.

**Developers’ skills**

The skills of the developers were measured over two dimensions:

1. Skill in software development using Java *(JAVA)*;
2. Skills in unit testing *(UT)*.

The Java programming language was selected as the lingua franca to help address the first skills dimension. Although the subjects in academia were accustomed to using Java in their curricula, not all of the professional developers were accustomed to using it during the course of their daily work. Object-oriented programming languages such as Java are a good fit for TDD Beck (2002). Deciding on a single programming language also removed several confounding factors: not only the different languages, but also other sources of variation associated with it, such as IDEs and testing frameworks, among others. In addition, a survey among thirteen thousand software developers Meyerovich & Rabkin (2013) showed that 94% knew an object-oriented programming language, with the percentage increasing to 97% in the case of developers who had a degree in computer science. Among the subjects in the studies included in this dissertation, only two professional developers did not have such a degree. The second dimension of skills deals with UT, a major component of TDD. UT was selected, rather than a more specific dimension (e.g., skill with the use of JUnit), because it encapsulates concepts that are similarly applied across programming languages and frameworks.

**Instruments and operationalization**

Because self-perceived measures are a reliable proxy for measuring programming skills and experience (Siegmund *et al.* 2014), a questionnaire with two four-point Likert-scale items was the tool used for measuring the two skills’ dimensions. An even-numbered scale was used to avoid having neutral answers.

The statements are:

**JAVA** - “Rate your skill level with the Java programming language”

**UT** - “Rate your skill level with unit testing”
The possible answers were: None, Novice, Intermediate, and Expert/Skilled. JAVA and UT are ordinal variables. The four levels were mapped to numerical values (0 = None, 3 = Expert). The questionnaire was administered through an online form, accessible only to the subjects during the training.

**TDD dimensions**

TDD advocates recommend that each cycle or micro-iteration should have a short duration—between five and ten minutes—and the rhythm of the cycles should be steady throughout the development (Beck 2002, Jeffries & Melnik 2007). Taking into account such characteristics of a TDD cycle, four sub-dimensions were identified: granularity, uniformity, sequencing (i.e., the order in which test and production code is written), and refactoring. In particular, the granularity and uniformity dimensions do not apply only to TDD, but to any cyclic process. When sequencing is also considered, some variants of TDD that differ from the textbook version of the process arise. TDD is considered to be the endpoint of the progression a traditional and monolithic approach to a highly iterative one (Jeffries & Melnik 2007). Hence, there is a continuum of processes over the dimensions from which relevant variants of TDD, such as ITL, originate.

Two of the dimensions, granularity (GRA) and uniformity (UNI), deal with the iterative nature of the process. Granularity measures the average duration of the development cycles. The granularity of the micro-iterative development process is defined in Formula 7.

\[
GRA = \frac{\sum_{i=1}^{n} duration(cycle_i)}{n}
\]  

(7)

where each cycle’s duration (duration[cycle_i]) is extracted from the report of the tool used to measure process conformance (see 3.2.1), and \( n \) is the total number of cycles. Therefore \( GRA \in [0, \infty) \). A small value of \( GRA \) identifies a granular development process.

Uniformity measures the dispersion of the duration of the development cycles. Keeping the rhythm of the red-green-refactor mantra of TDD requires uniform development cycles (Beck 2002). The uniformity of the micro-iterative development process is defined in Formula 8

\[
UNI = \sqrt{\frac{\sum_{i=1}^{n} (duration[cycle_i] - GRA)^2}{n}}
\]  

(8)

therefore \( UNI \in [0, \infty) \). A small value of \( UNI \) is a uniform development process. \( GRA \) and \( UNI \) alone are not the only dimensions to characterize a TDD process. Two other
dimensions, sequencing ($SEQ$) and refactoring ($REF$), are peculiar to a TDD process. Sequencing measures the ratio of development cycles that are carried out in a test-first fashion, as the first component of TDD. The sequencing of a development process is defined in Formula 9.

$$SEQ = \frac{\sum_{i=0}^{n} \begin{cases} 1 & \text{type}(cycle_i) = \text{test-first} \\ 0 & \text{otherwise} \end{cases}}{n}, \quad (9)$$

where the type of each cycle is obtained from the tool used to measure process conformance (see Section 3.2.1), and $\text{type}(cycle_i) = \text{test-first}$ is the number of development cycles identified as test-first. Therefore $SEQ \in [0, 1]$. A small value of $SEQ$ is associated with a TLD process.

Refactoring measures the ratio of development cycles that are dedicated to refactoring, as a second component of TDD. It is defined in Formula 10.

$$REF = \frac{\sum_{i=0}^{n} \begin{cases} 1 & \text{type}(cycle_i) = \text{refactoring} \\ 0 & \text{otherwise} \end{cases}}{n}, \quad (10)$$

where $\text{type}(cycle_i) = \text{refactoring}$ indicates the number of refactoring cycles; therefore $REF \in [0, 1]$. A large value of $REF$ is usually associated with a TDD process.

**Instrumentation and Operationalization**

The necessary artifacts for measuring the TDD dimensions are:

- The sequence of the development cycles recognized by the tool, based on the activities registered in the IDE and the tool’s heuristics (see Section 3.2.1);
- A script that calculates the duration of each cycle, and assigns it the right type.

The sequence of development cycles is a tuple (timestamp, type). An excerpt of the plain-text file where the tuples are stored by the tool is shown as follows:

1382698923646 test-first
1382699563036 refactoring
1382699761844 test-first
1382700418572 test-addition
1382700490976 regression

Running the script will generate a CVS file that contains the event itself, as well as the duration of each cycle (in minutes), calculated using the timestamps.
3.2.2 Sampling

Throughout the experiments reported in the original publications, the data was collected from two populations: professional software engineers and students who were taking software engineering / computer science courses.

Students

The students were sampled by convenience among the participants in a course offered by the Department of Information Processing Science, University of Oulu, on the topics of Software Quality and Testing. The course is offered yearly, and includes twenty-four hours of exercise in a laboratory environment. During the exercise sessions, the students learn about testing strategies and techniques, and become familiar with the necessary tools to create test suites. The students learn TDD alongside traditional UT. The students took part in eight exercise sessions of three hours each. Generally, the first sessions were used to train the students in the topics that would be investigated during the experiments. The experiments themselves then took place during the final sessions; the sampling technique was thus by convenience. The subjects were not offered any kind of compensation for participation; they signed a consent form to allow the collection and use of the data for the purposes of the experiments.

Professionals

The professionals who participated in the experiments were recruited from F-Secure, Ltd., and Company X. The two software companies operate in the security and personal entertainment domains, respectively. The subjects were sampled with the help of a “champion”: a figure in the company whose role is to ease industry-academia collaboration (Misirli et al. 2014). For the companies, the design of the experiments was different from those used in the academic settings. Training on unit testing was offered first, then the outcomes of interest (QLTY and PROD) were measured; next, the treatment (TDD) was applied, after which the outcomes were measured again. In order to recruit the experiments’ subjects, the champions advertised the training through the internal company mailing lists, although it was not explicitly mentioned that respondents would be participating in an experiment. The participants were thus
professional software developers with varying degrees of skills, but with potentially limited knowledge of TDD and UT.

### 3.2.3 Experimental Design Strategy

This section defines the research strategies, together with the settings in which they were utilized and the rationale that drove the choice of each strategy.

**Controlled experiment**

The definition of controlled experiment offered by Wohlin *et al.* (2012) is: “an empirical enquiry that manipulates one factor or variable of the studied settings.” In a controlled experiment, the sample gathered from the target population is randomly divided into two (or more) groups associated with each treatment. Each group is exposed to a treatment, where one group is usually given a placebo (i.e., no meaningful treatment is applied, or a baseline treatment is applied). The applied treatment should be the only factor that varies within the studied settings. If there are statistically significant differences between the groups in terms of the outcome of interest after a treatment is applied, such differences have a high probability of being due to the effect of the treatment, rather than chance. Hence, it is possible to infer that the treatment caused the variation. Controlled experiments are executed in laboratory environments, thus allowing a greater level of control when randomly assigning subjects to treatment groups and keeping the other variables at fixed levels. If this level of control is not feasible, quasi-experiments are preferable.

A controlled experiment design was used for the studies that were conducted in academic settings, since randomization was possible. These studies had one treatment (namely the development technique used) with two levels: traditional development or test-driven development. A blocking design is usually used to remove possible sources of variations that are irrelevant in the context in order to increase its precision. No blocking was applied in the controlled experiments.

**Quasi-experiment**

Along with controlled experimentation, another form of experimentation was used: quasi-experimentation. Wohlin *et al.* (2012) define a quasi-experiment
as an empirical inquiry in which the assignment of treatments to subjects cannot be
based on randomization, but emerges from the characteristics of the subjects (or the
subjects themselves). Quasi-experiments differ from controlled experiments mainly
because the assignment of the sample to either group is not random. At times, the
researcher may wish to use a specific criterion to make such assignments; at other times
a randomized assignment is not possible due to the context in which the experiment
takes place (Shadish et al. 2001). In particular, when experiments are performed in
software companies it is difficult to achieve a fully randomized design, whereas such
settings are more feasible for experiments in academia (Sjøberg et al. 2002). There
is also a tradeoff between the number of developers a company is willing to lend for
research purposes, as well as the representativeness of the subjects in the sample. Differ-
ent types of quasi-experimental design were used in the current study, as discussed below.

**Design 1**
This design was two-level, within subjects, also called repeated tests (Wohlin et al.
2012, Shadish et al. 2001). The industrial context in which the experiment was executed
precluded the possibility of having a sample big enough to divide the subjects into two
groups. All of the subjects in the sample thus first applied the control by performing a
task using a traditional approach to software development and testing (TLD). Afterward,
all of the subjects in the sample were instructed on TDD and performed a task using
such treatment. The concurrent execution of control and treatment (TLD and TDD) was
not possible; this in turn required two tasks (one for the TDL and one for the TDD)
in order to avoid a carry-over effect. If the task had been the same for both control
and treatment, the subjects could have performed better with the treatment (TDD):
not because of its effect, but because they were exposed to the same task during the
control (TLD). The measure of quality and productivity achieved for the TLD task was
compared with the corresponding measure achieved for the TDD task, in order to check
whether the usage of TDD had an effect on these two outcomes. Although the tasks
were different, their complexity was similar in order to allow for comparison.

**Design 2**
A quasi-experimental approach was necessary when assessing the effects of TDD
process conformance on quality and productivity. Process conformance was used as
the criterion for assigning the subjects to the different groups, rather than doing so
randomly. This is because it is not feasible to control for process conformance, since it
is a property that can only be measured after the task is completed. The sample was divided into control and treatment groups \textit{a posteriori} once the measure for process conformance was available. A cut-off value to split the sample was chosen according to the guidelines reported by Shadish \textit{et al.} (2001). The groups-dubbed “high conformant” and “low conformant”—were then compared (in terms of external software quality and developers’ productivity) in order to check whether process conformance had an impact. It must be noted that this kind of design, since it is similar to the repeated measure design—the only difference is that the intervention happens at a given cutoff, rather than a given time—has the same limitations in terms of establishing a cause-effect relationship between the constructs (Shadish \textit{et al.} 2001).

\textit{Observational design}

Along with the controlled and quasi-experimental design, another type of design, which is referred to as observational study, was used. Specifically, this design deals with the analysis of existing correlations (i.e., without manipulation) between the constructs that were investigated during the experiments. The objective was not to capture a strong cause-effect relationship, but rather to assess whether or not a construct variation is due to other observable constructs, which can be controlled in order to predict their behavior. The goal of the correlational designs was to create a model that, given the conformance to the TDD process and the measure of the developers’ skills, could assess the level of achievable software quality, and developers’ productivity. The same design was used to investigate the relationship between the same two outcomes, and the low-level dimensions of the process: the granularity of the development iterations and their uniformity, the ratio of test-first cycles, and the ratio of refactoring cycles.

3.2.4 \textit{Data Analysis}

This section introduces the most important analytical techniques used for processing the data collected at different stages, as well as the techniques associated with different experimental designs.
Descriptive statistics

Descriptive statistics are calculated after the data is gathered in order to have a sense of the data. Numerical descriptive statistics, and their graphical representation, can provide a sense of the way in which the data is distributed; this will be essential in deciding which kind of statistical technique to use. The main descriptive statistics include:

- Measures of a central tendency (e.g., mean, median, mode), which estimate a midpoint for the sample.
- Measures of dispersion (e.g., variance, standard deviation, variation interval), which convey the sample’s level of variation from the central tendency in order to indicate how dispersed or concentrated the data is.

Correlations

The data gathered during the experiment usually comes in pairs or sets (e.g., measures for quality and productivity for each subject). It is interesting to examine the dependencies between such variables. Pearson correlation is used in the case of normally distributed samples, whereas the Spearman correlation is used for non-normal data or data measured in an ordinal scale. Correlations are used as a “sanity check” before applying other non-robust statistical techniques for highly correlated data.

Linear regression

Linear models are a particular measure of dependency in which the variables are related through a linear function. Linear models played an important role when analyzing the data for the quasi-experimental design studies of the current project. Specifically, the models presented in these studies were multiple linear regressions, in which one variable depended on two or more other variables; for example, software quality was modeled using process conformance to TDD and developer’s skills. Linear regression models provided an estimate of the coefficients for the parameters from the data: for example, TDD process conformance or developers’ skills. When the estimates were statistically significant, the regression model was used to check the strength of the relationship: in other words, to check if the variables used to construct the model—TDD process conformance and developers’ skills—sufficed, or if there might be other, more compelling, factors. In this sense, the design of such studies was both explanatory and exploratory.
Linear models were one tool that was used to assess the relationship between the level of TDD process conformance the developers achieved, the developers’ skills, and the external software quality or developers’ productivity. At the same time, linear models defined the relationship between the low-level TDD dimensions and similar outcomes.

**Linear regression assumption assessment**

In order to properly apply linear regression, and to have a robust model, the following assumptions should be met (Tarling 2008).

- **Linearity**: the relationship between the IVs and DV should be linear. This can be assessed by looking at the scatterplot that represents the relationship between the DV and each IV individually.

- **No multicollinearity**: the IVs should be independent from one other. This was assessed using the Variance Inflation Factor, or VIF (Kutner *et al.* 2004)\(^7\)

- **Normality**: the distribution of residuals error should be Gaussian; this was assessed using the Shapiro-Wilk test (Shapiro & Wilk 1965), as well as visual inspection of the quantile-to-quantile plot (Kutner *et al.* 2004).

- **Homoscedasticity**: this is a constant variance in the residuals error, which should not depend on the IVs. The assumption can be assessed using the Breusch-Pagan test (Breusch & Pagan 1979), and by visually inspecting the scatterplot of residuals vs. fitted values, which should not present any linear relationship among the two (Kutner *et al.* 2004).

**Stepwise regression**

In a regression model, the IVs can be selected through an automatic procedure called stepwise regression. From an initial model, other models are derived according to a given criterion (Akaike 1974). In the observational study investigating the dimensions of TDD, the proposed model is was simplified using stepwise regression: specifically, backward selection was applied, which removes one variable at each iteration from the model; the new model is then compared with the previous one according to the selected criterion (Akaike 1974). Finally, the model that improves the criterion the most is selected. The criterion applied when selecting the best model for explaining the relationship between the DV and the dimensions was the Akaike Information Criterion,  

\(^7\)A VIF > 10 indicates multicollinearity
or AIC (Burnham & Anderson 2004). AIC represents the loss of information between the models that are built at each iteration for representing the data. When comparing models with the same AIC, the simplest one (i.e., the model with the fewest variables) is selected (Burnham & Anderson 2004). AIC thus offers a relative measure of the model’s goodness of fit. In the observational study, the model that minimized AIC after stepwise backward regression was selected.

**Null hypothesis significance testing (NHST)**

Hypothesis testing was used for the controlled experimental studies in which TDD was compared to TLD. The role of hypothesis testing in controlled experiments is to check if a given null hypothesis can be rejected. For example, in the controlled experimental studies, the properties investigated with hypothesis testing were software quality, and productivity; the null hypothesis stated that the distribution of such DVs came from the same group, whereas the objective of the study was to show that they actually came from different groups (TLD and TDD). The null hypothesis in the studies had a level of significance of 0.05, meaning that the null hypothesis was rejected only when there was at least a 95% probability that it did not observe a false positive (i.e., rejecting the null hypothesis when in reality it was actually true). The design of the controlled experiments was usually one factor (i.e., the development process the subjects used) with two treatments (TDD and TLD). The development process represented the experiments’ IV. Two statistical tests were used during the studies, according to the distribution of the IVs:

- A t-test was used to compare the groups when the DV was normally distributed.
- The Mann-Whitney test was used to compare the groups when the DV was not normally distributed.

Both tests check for differences in the groups using a measure of their centrality.

**Analysis of Variance (ANOVA)**

ANOVA was used in order to account for other variables that might have affected the results of the experiment, but that could not be controlled: for example, the subjects’ skills. ANOVA can take different forms and can be applied to different designs. Its primary use in this case was to align the results of the experiment by assessing whether
or not the different skill levels of the subjects affected the results. By using ANOVA, the differences between TDD and TLD could be identified for different subgroups of subjects: highly skilled, medium-skilled, and low-skilled. ANOVA compares the variance within the subgroups, rather than of the sample mean; thus the results are provided as an F-test (a parametric test used to compare the variances of two or more variables).

**Effect size**

Effect size is a family of measures of the magnitude of the effect of a treatment (Ellis 2010); it is used to evaluate the practical significance of an experiment’s result (as opposed to p-value, which evaluates the experiment’s statistical significance). An analysis of the effect size was used to draw conclusions from one of the quasi-experiments: three groups of developers, with different skills related to TDD, in terms of external quality and productivity. Since the particular experiment involved multiple comparisons, the more robust ANOVA was used instead of a multiple comparison of the groups using a t-test. In such cases, ANOVA would reveal whether or not there were any differences between the groups. In the case of ANOVA, the effect size thus measures the association between the DV and the groups. The specific effect size used was $\eta^2$, which represents the proportion of variation of the DV that can be explained by accounting for membership in the groups defined by the IV (Ellis 2010). Different effect sizes have different interpretations. Conventionally $\eta^2 \sim .2$ is considered small, $\sim .13$ medium, and $\sim .23$ large. The effect size is usually reported with a confidence interval (CI), which quantifies its precision; in other words, a CI is the range of values in which the effect size can vary. A CI is usually reported at 95% precision: i.e., there is a 5% possibility that the effect size will not fall within that the CI (Fritz et al. 2012).

**Clustering**

Using clustering, a set of observations are grouped together (i.e., clustered) so that the more similar ones belong to the same group (Nanda & Panda 2015). Clustering was necessary when forming the groups to be compared in one of the quasi-experiments. Specifically, the clustering algorithm used was k-means (Nanda & Panda 2015). K-means assumes a priori knowledge of the number of clusters, and treats each observation as a vector composed of several features. In the case of the quasi-experiment, the
features were \textit{JAVA, UT,} and \textit{CONF}. The algorithm then iteratively creates the clusters, starting from random centers, by minimizing the Euclidean distance between the centers and the other vectors. The heuristic used to identify the number of clusters was $\sqrt{\frac{n}{2}}$, where \(n\) is the number of observations (Mardia \textit{et al.} 1980).

### 3.3 Summary of the Research Approach

The research approach that was used has deductive, inductive and abductive facets. Figure 5 summarizes the research approach and its phases. The existing body of knowledge on TDD was surveyed during Phase 0. The literature thus far has not provided a definitive answer to the question of whether TDD is superior to TLD in terms of external quality and productivity. In addition, the research community has identified several knowledge gaps in the factors that might confound the effects of TDD (see Chapter 2).
Two hypothesis on the developers’ skills and their ability to conform to the process were selected from the body of knowledge (see Chapter 1). The experimental process deductively generated evidence of the developers’ skill factors (phases 1 and 2). Inductive reasoning was applied to synthesize the evidence and to form a working theory (Phase 3). The theory was inferred inductively, because it followed from the evidence only to a certain extent. The evidence did not completely fit the theory, since the existence of other factors that were not observed in the collected evidence cannot be ruled out (Sloman & Lagnado 2005). The theory on the effect of developers’ skill is referred to as a working theory, because it has not yet been tested to either falsify or confirm it.

In the case of process conformance, the evidence did not support the hypothesis after the execution of the experiments (Phase 1). This presented the opportunity to generate a new hypothesis. Such hypotheses are generated a posteriori, i.e., once the results from the experiments become available. The new hypothesis is then formulated so that the observation of other factors or rules may be taken into account in the theory (Givón 2014).

In this case, evidence that integrated the new hypothesis with the previous observation was obtained during the course of the observational study (Phase 2). The evidence gathered from the two sets of hypotheses was inductively used to generate a working theory on process conformance (Phase 3).
4 Original Research Papers

This section presents the publications included in the dissertation. All of the publications are ranked in the Finnish Publication Forum (JUFO, from the Finnish “Julkaisufoorumi”), which assesses the quality of academic research. Papers I-IV have been published in peer-reviewed international conference proceedings and journals: *Empirical Software Engineering Journal, special issue on experiment replication* (JUFO level 3); the *Seventh ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (ESEM 2013; JUFO level 2); the *Eighth ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (ESEM 2014; JUFO level 2); *Information and Software Technology Journal* (JUFO Level 3). Paper V is currently under review for *IEEE Transactions on Software Engineering* (JUFO level 3). Table 4 shows the contributions that each paper has made to the various research questions. The following subsections provide details on the motivations, findings, and ways in which each original contribution fits into the dissertation.

The paper summaries below follow the same basic structure: first, the rationale that drove the study presented in the paper is briefly introduced; a detailed description of the study is then presented, followed by the salient results. Finally, the papers’ main contributions are reported in the context of this dissertation; a more detailed account of their findings is reported in Chapter 5. The respective publishers granted permission to republish Figures 6–11.

A lab package for the studies presented in this chapter is available in Appendix 1.

4.0.1 Author’s Contributions

The author had major involvement in every phase of the research and is the main author of each publication. In particular, the author conceived, designed, collected, and analyzed the data for the experiments reported in the papers. The author was present and played an active role during the execution of each experiment, with the exception of the experiment conducted with Company X. The author led the writing of the papers, prepared the figures and tables, and prepared the final drafts, addressing the reviewers’ comments for each paper.

---

8http://www.julkaisufoorumi.fi/en
Table 4. Summary of the contribution of each publication. A = Academia, I = Industry.

<table>
<thead>
<tr>
<th>Publication</th>
<th>Purpose</th>
<th>Settings</th>
<th>Main finding</th>
<th>RQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Analyze whether TDD improves external quality and productivity, using the experimental design proposed by Erdogmus et al. (2005). In particular, testing productivity is used as a proxy for improvement in such outcomes.</td>
<td>A</td>
<td>C: Testing productivity does not correlate with external quality, but it positively affects short-term productivity when TDD is used.</td>
<td>RQ2.1, RQ2.2</td>
</tr>
<tr>
<td>I</td>
<td>Identify differences between TDD and TLD in terms of external quality and productivity. Investigate whether testing productivity can be a good proxy measure for the TDD effects, following the experimental design proposed by Erdogmus et al. (2005).</td>
<td>A</td>
<td>D: While TDD does not improve external quality nor productivity when compared to TLD, increased testing productivity is associated with productivity, but not with external quality.</td>
<td>RQ2.1, RQ2.2</td>
</tr>
<tr>
<td>III</td>
<td>Identify the relationship between the effects of TDD on external quality and productivity, and the process conformance.</td>
<td>A</td>
<td>A: The level of external quality and productivity do not depend on the process conformance to the TDD cycle.</td>
<td>RQ1.1, RQ1.2</td>
</tr>
<tr>
<td>IV</td>
<td>Evaluating the impact of developers with different TDD skill-sets—i.e., knowledge of programming, unit testing, and the ability to conform to the TDD process—on external quality and productivity.</td>
<td>I</td>
<td>E: The defined skill-set showed promising results regarding both outcomes.</td>
<td>RQ1.1, RQ1.2, RQ2.1, RQ2.2</td>
</tr>
<tr>
<td>V</td>
<td>Evaluating the impact of low-level characteristic of the TDD process—i.e., the granularity of the development cycles, their uniformity, and their type (test-first or refactoring)—on external quality and productivity.</td>
<td>I</td>
<td>B: The granularity of the development cycles is more important than the test-first approach of TDD for external quality. No significant were found for productivity.</td>
<td>RQ1.1, RQ1.2</td>
</tr>
</tbody>
</table>
4.1 Paper I—On the Role of Tests in Test-driven Development: a Differentiate and Partial Replication

**Rationale:** The first paper of the thesis addressed counterintuitive results produced by an experiment in an academic setting on the effects of TDD, published by Erdogmus et al. (2005).

**Description:** The original study compared TDD to TLD, and conjectured that testing productivity—the capacity of a software developer to write a substantial amount of unit tests to cover the system, which in the context of this dissertation is considered to be a proxy measure for the developer’s skill with UT—should be correlated to TDD, due to its focus on test-first.

On top of not showing a significant difference between TDD and TLD in terms of external quality and productivity, the original study did not show a difference in terms of testing productivity between the two techniques. Testing productivity was positively correlated with productivity, but not with external quality.

In order to understand the impact of testing productivity on the two outcomes of interest for this dissertation, the correlation part of the study was replicated at the University of Oulu. The replication study involved a mix of thirty graduate and undergraduate students. For the purposes of this replication, one additional factor, pair-programming, was considered, due to limitations imposed by the course that the experiment was embedded within.

**Results:** The results confirmed the original findings: in a TDD context, productivity improves with testing productivity ($\beta = .66$, p-value = .004), but not external quality ($\beta = .21$, p-value = .0513). The results are reported in Figure 6.

Such results hold when factoring out pair-programming. This replication differed from the original study due to changes in other contextual factors (for example, the time required to finish the experimental task was notably shorter in comparison with the original study) while yielding the same result. We thus strengthen the conclusion that testing productivity is not associated with better quality, as one might expect.

**Contribution to this dissertation:** In the context of this dissertation, this study provided the basis for setting up the subsequent studies in academia (paper II, a replication of the same original study); it also provided several useful insights into the types of skills that would later be used to answer RQ2.1 and RQ2.1 (paper II and paper IV).
(a) Relationship between developers’ productivity and testing productivity.

(b) Relationship between external quality and testing productivity.

Fig. 6. Paper I results: relationship between testing productivity and the outcomes in a TDD context, in academia
4.2 Paper II–A Replicated Experiment on the Effectiveness of Test-driven Development

**Rationale:** This paper reported an additional replication of Erdogmus et al.’s study (2005). The replication is considered close (Juristo & Gómez 2012), since it followed the investigation framework used in the original study.

**Description:** The study was conducted in academic settings using a sample of forty-seven graduate and undergraduate students. Two contextual factors were included in this replication that differentiated it from the original study: a shorter time frame to complete the experimental task, and the use of pair-programming rather than solo programming. Note that the same factors appear in the study reported in paper I.

**Results:** The results of the replication differed from the original study vis-à-vis the differences between TDD and TLD. In particular, no significant differences were observed between the two development techniques in terms of testing productivity (W = 114.5, p-value = .38), in contrast with the original study. On the other hand, neither external quality (W = 81.5, p-value = .53) nor developers’ productivity (W = 90, p-value = .82) were impacted by the specific technique, thus confirming the original study results. A positive correlation between testing productivity and productivity was found (β = .03, p-value = .001) in this study, which confirmed the result of the original study. As in the original study, a significant correlation between testing productivity and external quality could not be found (β = .17, p-value = .18). Changes in the experimental settings do not seem to have had an impact on the results. The results are presented in Figure 7.

**Contribution to this dissertation:** The results of this paper confirmed that the claimed effects of TDD are not easy to observe. Specifically, the results contributed to RQ2.1 and RQ2.2, as well as driving interest in determining which skills are necessary to follow TDD.

4.3 Paper III–Impact of Process Conformance on the Effects of Test-driven Development

**Rationale:** This paper addressed RQ1.1 and RQ1.2. Although it used the data gathered during the study presented in paper II, it is not considered a replication (Juristo & Gómez 2012), since it dealt with a different construct: TDD process conformance.

**Description:** The purpose of this paper was to analyze the relationship that might exist between the claimed effect of TDD and the extent to which developers follow
(a) Relationship between developers’ productivity and testing productivity for TDD, TLD, and aggregate groups. (LC: low-conformant, HC: high-conformant).

(b) Relationship between external quality and testing productivity for TDD, TLD, and aggregate groups.

Fig. 7. Paper II results: relationship between testing productivity and the outcomes in mixed context (TDD and TLD) in academia
(a) Relationship between developers’ productivity and TDD process conformance sub-grouped by high-conformant and low-conformant subjects.

(b) Relationship between external quality and TDD process conformance sub-grouped by high-conformant and low-conformant subjects.

Fig. 8. Paper III results: relationship between TDD process conformance and the outcomes in academia

the process in reality (i.e., process conformance). A tool known as Besouro\(^9\) was

\(^9\)http://github.com/brunopedroso/besouro

75
used in order to measure the latter (Becker et al. 2015); this tool is able to gather low-level data from the developers’ IDE and to reconstruct their development process. Well-established heuristics were then used to categorize the underlying development process into TDD-conformant and TDD-non-conformant groups.

**Results:** The regression models built for this study used data collected from twenty-two participants from academia. The goal was to model the relationship between conformance to the TDD process and external quality, as well as the productivity observed at the end of the study. Neither regression model was significant. An additional analysis that compared the low-conformant and high-conformant subjects also yielded non-significant results. These results, presented in Figure 8, question whether or not it is worthwhile to enforce the TDD cycle, at least in the short term.

**Contribution to this dissertation:** In the context of this dissertation, the contribution of this paper was to provide evidence that process conformance alone might not be enough to manifest TDD’s claimed effects. Therefore, in the following paper, process conformance was considered together with skills related to what the studies reported in papers I and II observed.

### 4.4 Paper IV–Towards and Operationalization of Test-driven Development Skills: An Industrial Empirical Study

**Rationale** This paper presented the results of an industrial study. The goal was to create a model based on a set of skills in order to explain the claimed effects of TDD on external quality and developers’ productivity. The skills that were considered included programming skills and unit-testing skills, along with process conformance (i.e., the ability to follow the TDD process). This paper thus tackled RQ1.1 and RQ1.2, as well as RQ2.1 and RQ2.2.

**Description:** A group of thirty professional software developers in two domains (security and entertainment) were sampled from two companies. First, the developers’ skills were assessed; next, the TDD-trained developers implemented a task of near-real-world complexity using TDD. The aforementioned tool Besouro was installed in their IDEs in order to gather data on their TDD process conformance.

The subjects were then clustered according to the dimensions of Java development skills, unit-testing skills, and TDD process conformance. These dimensions constitute the TDD skill-set that was investigated in this paper in relation to external quality and
productivity. The clustering resulted in three natural groups: low-level, medium-level, and high-level skills. The groups were then compared.

![Boxplots](image1.png)

(a) Boxplots representing the productivity of the three groups.

![Boxplots](image2.png)

(b) Boxplots representing the external quality of the three groups.

**Fig. 9.** Paper IV result: differences between three groups of professional subjects clustered according to their skills and process conformance

**Results:** A significant difference could not be found in terms of external quality \( (F[2, 27] = 1.44, p = .260) \) and productivity \( (F[2, 27] = 3.02, p = .065) \). Nevertheless, the three dimensions when taken together may play an important role, since the effect sizes and their CIs for external quality \( (\eta^2 = .9, CI = [0, 28]) \) and productivity \( (\eta^2 = .18, CI = \)
[0, 38]) are considered medium to large. The differences among the three groups of subjects (low, medium, and high) can be observed in Figure 9.

**Contribution to this dissertation:** In the context of this dissertation, this paper tied up the skill- and process-conformance factors that were individually considered in papers I-III. The results set a promising path for future research.

### 4.5 Paper V– A Dissection of Test-Driven Development: Does It Really Matter to Test-First or to Test-Last

**Rationale:** This paper investigated the issue of process conformance highlighted in paper III: the idea that strictly following TDD does not seem to improve either of the outcomes. This paper thus addressed RQ1.1 and RQ1.2.

**Description:** The process conformance dataset used in paper IV was enhanced with data collected from other tasks implemented by the same professional software developers. In total, data from eighty-eight software artifacts were collected, along with the underlying IDE data that represented each subject development process. In this paper, the overall process conformance (i.e., sequencing) was considered together with the cycles’ granularity (the duration of the cycles) and their uniformity (the cycles’ degree of similarity throughout the development process); the number of cycles dedicated to refactoring was the fourth dimension included in the study. An example of the representation of a single development cycle is shown in Figure 10. The individual relationship between the four dimensions (TDD sequencing, granularity, uniformity, and refactoring), external quality, and productivity were investigated using a generalized linear model. Taken individually, small granularity, cycle uniformity, and TDD sequencing were associated with improvements in external quality. The dimensions, and their interactions, were considered together in order to represent the underlying development process in the form of a multiple regressions model.

**Results:** The results from this initial model showed that only granularity, uniformity, and their interaction were statistically significant. Specifically, the shorter the cycles, the better the external quality; such a relationship was amplified by having development cycles of uniform duration. Finally, the initial model was simplified using stepwise regression; the resulting model for external quality only included the granularity and refactoring dimensions. The first had positive effects—the shorter the development cycles, the better the quality—while the latter had negative effects—the more the cycles were dedicated to refactoring, the worse the external quality became.
This last result showed an important aspect of TDD that is often neglected: the granularity of the development cycles matters. It might matter more than the actual process of writing test cases before the implementation. Alongside, refactoring can be detrimental if done in the wrong way. In the paper, the importance of isolating refactoring from floss refactoring (i.e., refactoring interleaved with other activities, like adding or modifying code for a feature not related to the current refactoring)
is highlighted. The regression models constructed capture the association between each dimension and external quality are presented in Figure 11. The paper reported partial results for productivity, since the shape of the distribution and the presence of outliers hampered reliable data analysis. Nevertheless, productivity had a significant, negative correlation with refactoring (i.e., the more refactoring was done, the lower the productivity). This result was expected, at least in the short term, because development cycles (and therefore time) are not spent on implementing new features.

**Contribution to this dissertation:** In the context of the dissertation, the paper builds from the results of paper III. It presents a new level of analysis of TDD process conformance that, together with the results of papers III and IV, can provide answers to RQ1.2 and RQ1.2.
5 Discussion and Limitations

This chapter reports a synthesis of the main findings of the empirical studies presented in Chapter 4. Figure 12 presents the five main findings of this dissertation, labeled A to E. Specifically, sections 5.1 and 5.2 answer the research questions, bearing these findings in mind. Finally, Section 5.3 addresses the limitations and threats to the validity of the results.

Due to the diversity of the designs of the studies included in this dissertation, a systematic synthesis—e.g., using meta-analytical methods (Pickard et al. 1998, Hayes 1999, DerSimonian & Laird 1986)—was not attempted.

The findings were attained inductively10 by integrating the evidence collected in the studies included in this dissertation. Using inductive reasoning, the findings followed from the evidence only to a certain extent (i.e., inductive probability). As acknowledged in Section 5.3, other factors that could have influenced the findings may not have been taken into account; these findings thus constitute working theories: theories that follow from evidence but are not tested to either falsify or confirm the theories.

5.1 Answer to RQ1: How does process conformance impact the claimed effects of TDD?

Given Findings A, B, and E, the answer to this research question is that TDD process conformance does not have a straightforward impact on the claimed effects of TDD.

Finding A shows that adhering to the TDD development cycle is not associated with a clear improvement in either external quality nor the productivity of the developers. This result challenges the positive claims associated with TDD (Beck 2002, Astels 2003). While one might expect that the more a developer follows TDD, the more the positive effects of TDD should be visible, no supporting evidence of this claim was found.

Causevic et al. (2011) showed that insufficient adherence to TDD was one factor to impede the adoption of TDD in industry. Finding A challenges this result, because the process-adherence factor (i.e., process conformance) might not be enough to ensure industry adoption, since a clear positive effect was not found. Nevertheless, it should be

---

10Deducing the findings from the evidence was not possible, because the evidence did not always guarantee the findings.
taken into account that the results that support this finding are based only on academic studies.

The result of paper III might indicate that developers tend to balance between following the process and getting things done. Although novice developers could learn and apply the rules of TDD after a short training—a result supported by Latorre (2014)—what it takes to master TDD in terms of quality and productivity is far from being known with certainty.

In contrast, Finding A suggests that following TDD to the letter might not be beneficial. Finding B shows that keeping the development cycle short—whether writing test-last or test-first—could already bring improvements in quality. In this sense, Pančur & Ciglarič (2011) attribute the cause of the inconclusive results of many controlled experiments on TDD’s effects to the technique to which it is compared—TLD—which is also characterized by short development cycles. The authors’ claim is that the effects of TDD’s test-first nature are concealed by the granularity of the cycles, regardless of the order in which the tests are written (Pančur & Ciglarič 2011).
Finding E also supports the idea that conformance to TDD is not enough, since skill also appears to be necessary. Indeed, when the capacity to follow the TDD cycles is accompanied by skills in testing and programming, the improvement in external quality and productivity start to become visible. Given findings B and E, one open point of this dissertation is whether or not the improvements could be a function of the identified skills (paper IV) together with the ability to keep the development cycles short (paper V), as opposed to strictly following TDD. This additional level of investigation will be left for future work (see Section 6.2).

5.1.1 Answer to R1.1: How does process conformance impact the claimed effects of TDD on software external quality?

The answer to RQ1.1 is that process conformance alone has a negligible impact on external quality. Nevertheless, keeping the development cycles short, as well as having experience with skills related to the TDD process, can improve external quality.

Finding A (paper III) shows a positive, although non-significant, relationship between TDD process conformance and external quality. Due to the high dispersion of external quality scores for the most conformant subjects, one conclusion of paper III was that other unaccounted-for variables might influence the results.

Given the results of paper IV, and the subsequent Finding E, it appears that such hidden variables are linked to the developers’ existing skills. In particular, unit testing and programming can positively affect the quality of the software. Nevertheless, it should be taken into account that this result originates from very few studies with low power (see Section 5.3).

Quality improvements are also achieved when the TDD process is skinned from its test-first approach, and the focus is shifted on keeping the development cycles as short as possible, supporting the idea that TDD conformance is not linked to increment in quality.

5.1.2 Answer to R1.2: How does process conformance impact the claimed effects of TDD on developers’ productivity?

The answer to RQ1.2 is that TDD process conformance alone has no significant effects on developers’ productivity.
Finding A shows that there is no relationship between process conformance and productivity; given Finding E, however, the previous result changes when considering additional factors.

In reality, when examining subjects who possessed programming and unit-testing skills, and who better conformed to the rules of TDD, a considerable improvement in productivity was observed in comparison to subjects with lower skill levels. Productivity improvements are thus only secondarily due to process conformance, since such improvements do not play as significant a role as skills do. Such a claim is supported by Latorre’s longitudinal study (2014) of developers at different skill levels who applied TDD. In that study, the most skilled developers’ productivity performance did not significantly change once they switched from a traditional development approach to TDD, or when the same subjects reached a higher level of process conformance.

Finally, Finding B only considers external quality; it does not apply to productivity. In the case of Finding B (paper V), the type of analysis that was done for external quality was not possible for productivity, due to the intractable distribution of the data. It is currently difficult to assess the effects of granular development cycles (as opposed to the entire TDD process) on productivity. According to the secondary studies (Turhan et al. 2010, Rafique & Misic 2013, Munir et al. 2014), the effects of TDD on productivity (if any) are the most controversial. The study of productivity in the TDD context thus offers material for future work.

5.2 Answer to RQ2: How do developers’ skills impact the claimed effects of TDD?

The answer to RQ2 is that developers’ skills have a substantial impact on TDD’s claimed effects.

Causevic et al. (2011) identified the lack of previous testing skill as one factor that limited the adoption of TDD. The findings of this dissertation second this claim. As shown in similar previous studies, the lack of particular skills can hamper the TDD process. For example, Geras et al. (2004) suggested that adopting TDD without having testing skills can be particularly detrimental for productivity, since developers fall back to time-consuming techniques such as debugging.

Finding D shows that the effort of writing unit tests is positively associated with productivity. Considering that two replications (papers I and II), as well as the original
replicated study Erdogmus et al. (2005), yield the same results, this claim is sound, at least in academic contexts.

Studies on the effects of TDD on outcomes that imply skill in testing (e.g., branch coverage, mutation coverage, and test quality) are negligible in the literature. Studies in academic settings based on minor programming tasks (i.e., settings similar to those used in papers I and II) could not show any affects of TDD on branch and mutation coverage metrics (Madeyski 2010, Pančur & Ciglarič 2011, Causevic et al. 2012). The only industrial study to investigate the same metrics in a TDD context is presented in (Alkaoud 2014). The author concludes that “as more tests are written in TDD, branch and the statement coverage improve.” Nevertheless, putting more effort into writing unit tests does not necessarily improve mutation coverage (Alkaoud 2014).

This dissertation shows that substantial improvements may be achieved when testing and programming skills are paid attention to in a TDD context.

5.2.1 Answer to R2.1: How do developers’ skills impact the claimed effects of TDD on external software quality?

The answer to RQ2.1 is that when only testing skill is considered, no impact on external quality is observed, but a positive impact is observed when experience with other skills is also considered.

Finding C, as well as Erdogmus et al.’s study (2005), both support the idea that putting effort into writing unit tests, as in the case of a TDD scenario, does not have an impact on external quality. Desai et al. (2009) reached the same conclusion in their survey of TDD use in academia.

One explanation is that testing skill per se might not directly indicate tests’ thoroughness. Cauvevic et al. (2013) proposed one possible solution; the authors observed a 17% improvement in external quality by extending the TDD cycle by including equivalence partitioning testing Juristo et al. (2012) in order to improve the thoroughness of the test suite. The results of this dissertation, however, show that skill in programming and testing—and only secondarily a focus on TDD process conformance—yield results similar to those found in Cauvevic et al. without modifying the original TDD cycle.
5.2.2 Answer to R2.2: How do developers’ skills impact the claimed effects of TDD on their productivity?

The answer to RQ2.2 is that developers’ skill have an impact on the effects of TDD on their productivity.

Findings D and E provide evidence that the primary factors to take into account in terms of productivity are testing and programming skill, while TDD process conformance is of secondary interest.

Better productivity can be the result of developers having more skills, regardless of the application of TDD. For example, Fucci et al. (2015) reported on a study in an academic context that showed that the magnitude of the correlation between testing productivity and productivity depended on the subjects’ skills in testing and programming. Latorre (2014) compared novice and expert TDD developers over the span of several months and found that skilled developers had a higher level of productivity; their productivity, however, did not differ much when moving from a TLD to a TDD development process. Despite these limitations, both studies concluded that, for novice developers, following TDD can help to establish baseline levels of productivity (Latorre 2014, Fucci et al. 2015).

Rafique & Misic (2013) showed, on the other hand, that more skilled subjects tended to spend more time engaged in TDD-specific activities, which decreased their productivity in comparison with subjects within academia.

5.3 Threats to Validity

The validity of empirical research in general may be threatened by several factors Shadish et al. (2001). Each publication in this dissertation includes a detailed discussion of the respective studies’ validity. The following section summarizes the studies’ validity threats according to type, using the guidelines that Wohlin et al. (2012) devised specifically for empirical studies within the software engineering field.

The types of validity threats considered in this dissertation include:

1. External validity: the generalizability of the findings outside the scope in which they were originally investigafted.
2. Construct validity: the representativeness of the measures that are used to represent the concepts under consideration.
3. Conclusion validity: the soundness of the statistical inference.
4. Internal validity: the correctness of the interpretation of the relationships between the concepts under consideration.

The threats to external validity are addressed according to the different settings of the contributions (academic vs. industrial). The priority of each threat greatly depends on the context, since different priorities reflect different trade-offs (Shadish et al. 2001). In this dissertation, because the trade-off leans toward the threats that are salient for applied research, higher priority is given to the applicability of the results and their correct interpretation.

5.3.1 External Validity

The different settings of the studies included in this dissertation mean that the different threats need to be examined in different ways.

**Findings A, C, and D**

The evidence to support these findings was gathered from experiments that were conducted in university settings. The threat of the interaction of selection and treatments could have taken place, since students could not be generalized to the population of professional software developers (although initial evidence shows that professionals and students tend to perform similarly during experiments, e.g., Müller & Höfer [2007], and Salman et al. [2015]). At the same time, a threat due to the interaction of settings and treatments took place because the tasks that were used during the experiments were not representative of real-world situations. Within the scope of this dissertation, the limitation of the results that arises because the studies were conducted in an academic context should be taken into consideration. Nevertheless, Finding A was replicated in paper V, in which process conformance was not shown to have a relationship with either outcome (as opposed to the granularity of the development). To a lesser extent, the same applies to Finding D, which was replicated in paper IV. Finally, no threat of interaction of history and treatments occurred.

**Findings B and E**

The studies that supported these two findings were conducted with the participation of professional software developers; the tasks and the settings were very similar to those found in real-world conditions. The findings are thus generalizable, at least to similar companies (i.e., those that operate in the same domains). The findings are especially relevant because the evidence was collected from companies located in the Nordic
countries—a region with one of the most efficient software-development industries in the world.

5.3.2 Internal Validity

The threats to internal validity described in this section apply to all of the findings. Due to the studies’ settings (see Chapter 3), the findings reported in this dissertation were not based on a strong causal relationship. The quasi-experimental design that underlies the studies posed a single-group threat—i.e., determining whether the observed results were due to the factors under consideration, or whether other factors have influenced the results. The quasi-experiments were designed to limit or cancel the effects of history, maturation, or testing threats. In addition, no conscious threat to instrumentation took place, as the subjects acknowledged after each experiment.

In terms of the threats that arose from the sampling process, the industrial-context studies suffered from selection threat: the subjects volunteered to participate in the experiment, which was part of a free training on TDD. The subjects who were included in the sample may have been more motivated than engineers who did not participate, due to their willingness to learn something new. Subjects in the academic experiments, on the other hand, were sampled by convenience, because participation in the experiments was a mandatory part of one university course. Using this strategy, subjects with different motivations were included in the sample.

The statistical regression to the mean did not pose a threat, since no pre-test or existing data on the subjects (for example, university grades or level of seniority) was used to compose the sample.

The threat posed by mortality was mainly present for the academic-context experiments, due to students dropping out of the course in which the experiment was embedded. The drop-outs’ impact on the results was assessed in each study.

A threat related to the ambiguity of the direction of the causal influence was embedded in the studies that were designed as quasi-experiments (see Chapter 3).

Determining a strong cause-effect relationship was not the goal of this work, however. For example, rather than being able to claim that process conformance causes improved external quality, the objective of this work was to ascertain whether or not process conformance plays a role in the first place, and if it does, to describe that role. Given the analyses of the effect sizes reported in the studies, the author acknowledges that other factors play a role in the inner workings of TDD. This threat can be addressed by
introducing a control group. Due to the nature of the constructs of interest, however—process conformance and existing skills—it was not possible to artificially create a control group.

In terms of the social threats to internal validity, the diffusion or imitation of treatments was limited by taking ad-hoc precautions in each of the different settings: for example, by controlling the time when the subjects could exchange information and by safeguarding access to the experimental material. No other social threats took place.

### 5.3.3 Construct Validity

The following threats applied to the same extent to all of the studies included in this dissertation. The studies suffer from mono-method and mono-operation bias: only a single task and a single metric was used to measure each construct.

The restricted generalizability across constructs threat may have taken place, since other constructs (closely related to the constructs of interest) were not considered. For example, the results showed that having experience with particular skills could be beneficial for productivity, but possible side-effects (on internal quality, for instance) were neglected.

The interaction of different treatments could have occurred in studies where pair-programming was used together with TDD (papers I and II), although an ad-hoc analysis did not show any significant interaction.

The threats of inadequate pre-operational explication of constructs was addressed by using metrics and material that was closely derived from (or already used in) in previously published scientific literature.

In terms of social threats, hypothesis guessing may have taken place for the later studies, because the participants may have become acquainted with the experimenters’ work; the hypotheses were never disclosed to the subjects before the onset of the experiments, however.

The evaluation apprehension threat may have taken place during the academic experiments, where the subjects were students who were participating in a graded university course. It was made clear to them, however, that their performance during the experiment would not affect their final grade.

Finally, the experimenter expectancies were controlled by assigning at least two researchers to the design, analysis, interpretation, and report of the studies.
5.3.4 Conclusion Validity

The main threat to the conclusion validity was the low statistical power of some of the studies, although the academic studies replicated previous results. This points to a real effect (or lack thereof) in reality (Juristo 2013, Juristo & Vegas 2009), although to a lesser extent, the industrial-context studies suffered from the same threat. Due to the complexity of conducting experiments in industry (Vegas et al. 2015, Misirli et al. 2014), this threat should be accepted; and evidence should be gathered by replicating experiments (Juristo & Gómez 2012, Carver et al. 2014) or triangulating the results with different sources (Creswell & Clark 2007).

The random heterogeneity of subjects was not a threat for the industrial-context studies, in which the differences of existing skills was the factor under investigation. On the other hand, the subjects of the academic-context studies—since they were sampled in the context of a university course—were homogeneous, which could have limited the generalizability of these studies. The goal of the academic studies was not to yield general results, however, but rather to validate previous findings in the same settings and to assess them within the industrial context.

The measures of the different constructs that were investigated in this work were reliable, since they have been validated in the literature and were based on automatically (or semiautomatically) calculated measures.

The reliability of the treatment implementation should not be considered a threat per se, due to the design of the studies. In particular, the measure of process conformance inherently represents different levels of implementation of the treatment.

The statistical tests’ assumptions were assessed thoroughly in each study. At the same time, the error rate was taken into account when necessary, and the appropriate statistical methods were applied.

Finally, no extraordinary events that could have influenced the course of the studies took place.
6 Conclusions and Future Work

This dissertation contributes to an understanding of how factors such as developers’ skills and process conformance influence the effects of TDD. The empirical evidence, gathered through experimentation in both academia and industry, have outlined the relationship between these two factors and two outcomes: the external quality of a software system developed using TDD and the productivity of that system’s developers.

The research contributions from this dissertation suggest that developers’ skills are associated with improvements in both outcomes. While process conformance per se does not seem to be associated with any positive or negative effects, the studies do indicate that focusing on keeping the TDD cycle shorter, rather than focusing on the test-first approach, can be beneficial for external quality. 6.1 summarizes the main conclusions of this dissertation, focusing on the implications they have for both academia and industry. Figure 13, together with Table 5, show how the contributions, the findings of the papers included in the dissertation, the research questions, and the research gaps are linked together. Section 6.2 describes future research directions based on the findings and limitations of this work.

Fig. 13. Overall mapping of the research
Table 5. Summary of the definitions of the dissertation’s Research Gaps, Research Questions, Papers, Findings, and Contributions.

<table>
<thead>
<tr>
<th>Type</th>
<th>ID</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Research Gap</td>
<td>1</td>
<td>The question of whether or not the developers’ skills plays a role in explaining TDD’s effects (or lack thereof) on external quality and developers’ productivity has yet to be studied.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>The question of whether or not developers’ conformance to the TDD process plays a role in explaining the TDD’s effects (or lack thereof) on external quality and developers’ productivity has yet to be studied.</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Lack of software engineering experiments conducted in industry.</td>
</tr>
<tr>
<td>Research Question</td>
<td>1</td>
<td>How does process conformance impact the claimed effects of TDD?</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>How does the developers’ skills impact the claimed effect of TDD?</td>
</tr>
<tr>
<td>Paper</td>
<td>I</td>
<td>On the Role of Tests in Test-driven Development: A Differentiate and Partial Replication</td>
</tr>
<tr>
<td></td>
<td>II</td>
<td>A Replicated Experiment on the Effectiveness of Test-driven Development</td>
</tr>
<tr>
<td></td>
<td>III</td>
<td>Impact of Process Conformance on the Effects of Test-driven Development</td>
</tr>
<tr>
<td></td>
<td>IV</td>
<td>Towards an Operationalization of Test-driven Development Skills: An Industrial Empirical Study</td>
</tr>
<tr>
<td></td>
<td>V</td>
<td>A Dissection of Test-driven Development: Does It Really Matter to Test-First or to Test-Last</td>
</tr>
<tr>
<td>Finding</td>
<td>A</td>
<td>Neither the claimed effects on improved external quality and developers’ productivity are solely affected by process conformance</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>Rather than focusing on the test-first part of the TDD process, external quality improvements are associated with granular development cycles</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>Focusing on writing unit tests in a TDD fashion does not have an impact on external quality</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>An increase in productivity is associated with increased effort in writing unit tests</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>Focusing on process conformance, together with other TDD-related skills (i.e., knowledge of unit testing and programming), has the potential to improve both quality and productivity</td>
</tr>
<tr>
<td>Contribution</td>
<td>a</td>
<td>The test-first approach alone is not associated with improvements in external quality, or developers’ productivity.</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>Professional software developers can improve the external quality of the software they write by focusing on the granularity of the development cycles, rather than writing unit tests before production code during the cycles.</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>Skilled professional developers should also acquire or improve their unit-testing and programming skills, rather than scrupulously applying TDD, to gain benefits in external quality and productivity.</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>Writing unit tests in a test-first fashion is associated with gains in productivity for novice developers.</td>
</tr>
<tr>
<td></td>
<td>e</td>
<td>Refactoring, if not properly applied, can harm the external software quality.</td>
</tr>
</tbody>
</table>
6.1 Summary of Contributions

The synthesis of the studies summarized in points a, b, and c consists of three main contributions; contribution d is based on replication studies that verify the claims that Erdogmus et al. (2005) made vis-à-vis the effects of testing productivity. Finally, contribution e is an ancillary contribution that is not directly associated with the research questions.

a) The test-first approach alone is not associated with improvements in external quality, or developers’ productivity.

The results of the studies reported in papers III and V showed that, contrary to claims that have been made about the test-first nature of TDD, writing unit tests before implementing production code cannot be related with either positive or negative variations in either outcome of interest. This contribution applies to both novice software developers in academic contexts and professionals in industrial contexts. Focusing on writing unit tests, however, can help novices improve their productivity (see contribution d).

b) Professional software developers can improve the external quality of the software they write by focusing on the granularity of the development cycles, rather than writing unit tests before production code during the cycles.

The result from paper V showed that, if the TDD process is dissected in its sub-dimensions, the order in which the tests are written (i.e., sequencing) has no correlation with external quality (see contribution a), unlike the granularity of the cycles, which is associated with gains in external quality. By keeping the development cycles short—for example, by tackling a very small feature in each cycle—the developers could better focus on the feature’s correctness, as the simplicity of the feature makes the sequencing a negligible factor. The association between the sub-dimensions of TDD and developers’ productivity could not be assessed, thus representing material for further investigations. This contribution applies to professional software developers, and was not verified for subjects like university students. Nevertheless, the results from papers I and II suggests that novices should strive to write unit tests in a TDD fashion in order to improve their productivity (see contribution d).

c) Skilled professional developers should also acquire or improve their unit-testing and programming skill, rather than scrupulously applying TDD, to gain benefits in
external quality and productivity.

Paper IV presents the findings related to the role of skills in TDD’s effects. For professional software developers, the conclusion is that skill in unit testing and programming (specifically Java, as it was the language used during the study) has the potential to substantially increase both external software quality and productivity. Although it is not suggested that professional developers should give up on TDD, it is recommended that they avoid focusing on being conformant to the process at all times. Although the paper’s findings do not suggest that professional developers should give up on TDD, the paper does recommend that they should avoid focusing on conformance to the process at all times. The study showed TDD process conformance to have some dimensions that are more important than others (see contributions a and b), at least for skilled software developers. Novices, on the other hand, should focus on unit testing to see improvements (see contribution d).

d) Writing unit tests in a test-first fashion is associated with gains in productivity for novice developers.

The findings from the studies the academic context have shown that productivity and the ability to write unit tests are correlated. When compared to professional software developers, students might not possess the necessary skills to spot bugs and other shortcomings in their code. Without an extensive test suite (e.g., skipping unit tests), novice software developers such as students must debug their source code often in order to find and correct bugs, thus setting back their productivity. Since the TDD process forces developers to start by writing unit tests, a test suite promptly becomes available for regression testing. While the results reported in papers I and II support these claims, it is still unknown whether the same results would hold once a different process—still focusing on unit tests (e.g., ITLD)—was applied. External quality was not correlated with the quantity of unit tests written by the novice developers who used TDD. In this sense, it is possible that the individual skills in other aspects (for example, the particular programming language or programming language model) were more significant.

e) Refactoring, if not properly applied, can harm the external software quality.

In a TDD setting, refactoring was found to be detrimental for sto software quality. In other words, the more refactoring was done, the lower the external quality. The inherent goal of refactoring is to improve the internal quality of a codebase (for example, reducing coupling and redundancy), but it is detrimental to external quality once existing
features are silently broken during refactoring activities: for example in the case of a feature that is not fully covered by tests. When doing floss refactoring—refactoring applied while still working on a feature—faults could be injected into a feature while attempting to refactor part of its code. Such scenarios can manifest when tackling overly large features (e.g., low granularity), and subtle bugs caused by refactoring can easily go unnoticed. This scenario is accentuated when tools that offer proper support for such situations (live code coverage, for example) are unavailable. Although this is a secondary result that is outside the scope of the RQs of this thesis, the role of refactoring in TDD, and which types of refactorings are better suited for the process, represent material for future research.

6.1.1 Relevance to Academia

Academic curricula that are in the process of implementing (or are willing to implement) TDD should factor in the method’s implications in consideration of the results presented in this dissertation.

TDD increases the workload for students due to the counterintuitive activity of writing unit tests before production code (Desai et al. 2008, 2009). Nevertheless, contribution a reveals that the “magic” of TDD is not in its test-first sub-process. When also bearing contribution b in mind, it becomes apparent that teachers should stress the importance of taking on small features (or “baby steps,” as it is referred to in the agile lingo). Such skills cannot be taught by following strict theoretical definitions, since they depend on factors such as the size of the task. In this regard, one practice that can be implemented with students is the “elephant carpaccio”11 proposed by one of the initiators of the agile movement, Alistair Cockburn. These types of practice can help students reach a good level of granularity for the part of the system they are willing to implement. In turn, having a manageable task to implement can in turn facilitate other practices, such as unit testing and refactoring.

Shifting the focus to slicing the requirements into smaller components should not imply that unit testing should be abandoned. On the contrary, because writing sizeable amounts of unit tests has been found to be beneficial for students who are learning unit testing and TDD, it is a practice that should be enforced. In this sense, the test-first component of TDD is beneficial, since it compels students to write tests that would be likely to be left out in a test-last scenario. IDEs’ support for the process is beneficial in

11http://alistair.cockburn.us/Elephant+Carpaccio+exercise
this case, since developers could be promptly informed of the need for testing (or better testing).

Finally, contribution e warned about the bad effects that refactoring can have. Although the nature of the ways in which floss refactoring episodes can have a negative impact on code quality have not been firmly established, it is advisable to take such possibilities into account when teaching refactoring together with TDD.

6.1.2 Relevance to Industry

A software development industry that is already adopting (or is willing to adopt) TDD could benefit from this dissertation’s contributions. While the suggestions presented in this section target software developers, they represent a baseline for managers who are willing to improve the quality of their product, as well as the productivity of the development team.

Given contribution a, software developers and managers alike should be aware that enforcing test-first activities can be an overhead. Rather than striving for to have 100% of tests written in a test-first fashion, they should pivot their interest toward keeping their development iterations (either TDD or not) short: for example by breaking the requirements or user stories down into smaller components. Although having smaller requirements is beneficial for the development team, the activity of breaking down such requirements can be supported by other units, such as product owners and customers. All in all, the development process should include a phase in which requirements are systematically broken down into tasks that could ideally fit into a single TDD iteration: for example, between five and ten minutes, as reckoned in the agile community.

Developers who have skills in unit testing and programming should adopt TDD, since they are more likely to benefit from the practice’s claimed effects on quality and productivity (see contribution c). In contrast, this dissertation’s findings suggest that less-skilled developers should not strictly follow TDD, but rather should focus on other activities: for example, short development cycles. Finally, although researchers have yet to clearly assess the role of refactoring in TDD, this dissertation’s findings recommend that floss refactoring (for example, attempting a refactoring during any other phase of the TDD cycle) should be avoided, since these kinds of activities were found to be detrimental for external quality due to the possibility of injecting bugs.
6.2 Future Work

From a research prospective, the results of this dissertation open several opportunities for further study. The roles of process conformance and developers’ skills for the study of TDD both appear to be promising; replications of the studies that were reported in this work are necessary to either confirm or refute the findings, however. Replications will also help to establish the context and applicability of the results: for example by investigating different domains of expertise of the subjects or utilizing different kinds of tasks.

Although this dissertation has provided a general understanding of the effects of process conformance and developers’ skills on the practice of TDD, both factors are candidates for more detailed study. For example, the set of developers’ skill factors could include other skills that might be salient; developers’ skills could also be evaluated by the more sophisticated tools that are currently being proposed in the literature (Bergersen et al. 2014). While one of the most important results of this work is that keeping short development cycles yields better software quality, the work that was conducted did not investigate the way in which the developers’ skills interact with this factor, nor did it assess the effects of granular cycles on productivity. Such points thus should be taken into consideration for further studies. The findings of this work have demonstrated that refactoring can have a negative impact on external software quality; although such effects were attributed to floss refactoring, more studies are required to support this claim.

In a broader context, future work could include an investigation of outcomes other than external quality and productivity: for example, aspects related to the claim that TDD improves internal quality and test quality. Although TDD was used together with pair-programming in paper I settings, TDD should be studied in a context in which other software development practices are also used in order to make the results more realistic.

Future work that industry could take advantage of might include the creation of guidelines for the adoption and adaptation of the TDD process based on the findings reported here, as well as the findings’ validation and communication to practitioners.

Finally, tools that could monitor the duration of the development cycles and the type of refactorings, and that could suggest corrective actions to developers, should developed to assist practitioners with the process.
References

Chalmers AF (1999) What is this thing called Science? Hackett Publishing Co., Indianapolis, IN.


Appendix 1 Lab Package

A lab package for replicating the studies presented in this dissertation is available at http://dx.doi.org/10.6084/m9.figshare.1595913 (Retrieved 12:17, Feb 15, 2016 - GMT)

The package includes:

**Acceptance tests:** the acceptance test suite, written in Java, used to calculate the external quality and productivity metrics.

**Besouro Eclipse plugin:** a plugin to be installed in the Eclipse environment used during the study. The plugin logs the activities within the IDE and outputs a series of development episodes, along with their conformity to TDD. An installation and user manual for the plugin is also available.

**Besouro log analyzer script:** a Python script to extract the process dimensions, used in paper V, from Besouro’s logs.

**Consent form:** to be filled out by the subjects in order to inform them about the purpose, procedures, risk, and benefit of their involvement in the study.

**Experimental tasks:** description of the tasks used during the study. The Bowling Scorekeeper task was used for the academic experiments, whereas the MusicPhone task was also used for the industry experiments. A skeleton project is also included as a starting point for task development.

**Metrics extraction form:** a spreadsheet to calculate external quality and productivity metrics based on the output of the acceptance tests.

**Questionnaires:** to be administered to the subjects before the study: it is the tool used to measure their experience relative to the two skills of interest in the dissertation, unit testing and programming.

**Training exercises:** programming tasks to be used before the study to familiarize the subjects with the topics, tools, and framework (if necessary). The exercises are in the form of code “katas” of varying difficulties.

**Training material:** used to train the subjects before the study. The topics covered include unit testing, unit testing patterns, and TDD.
Original publications


Reprinted with permission from Springer (I), The Institute of Electrical and Electronics Engineers, IEEE (II and V), Association for Computing Machinery, ACM (III), and Elsevier (IV).

Original publications are not included in the electronic version of this dissertation.

657. Lappalainen, Katja (2015) Modification of native and waste starch by depolymerization and cationization: utilization of modified starch in binding of heavy metal ions from an aqueous solution


663. Pakanen, Minna (2015) Visual design examples in the evaluation of anticipated user experience at the early phases of research and development

664. Hyry, Jaakko (2015) Designing projected user interfaces as assistive technology for the elderly


666. Luukkonen, Tero (2016) New adsorption and oxidation-based approaches for water and wastewater treatment: studies regarding organic peracids, boiler-water treatments, and geopolymers

667. Tolkkinen, Mari (2016) Multi-stressor effects in boreal streams: disentangling the roles of natural and land use disturbance to stream communities

668. Kaakinen, Juhani (2016) Öljyllä ja raskasmetalleilla pilaantuineet maita koskevan ympäristölainsäädännön ja lupamenettelyn edistäminen kemiallisella tutkimuksella


670. Rönkä, Nelli (2016) Phylogeography and conservation genetics of waders
Davide Fucci

THE ROLE OF PROCESS CONFORMANCE AND DEVELOPERS' SKILLS IN THE CONTEXT OF TEST-DRIVEN DEVELOPMENT