

An Industry Experiment on the Effects of Test-Driven Development on External Quality and Productivity

Ayse Tosun

Faculty of Computer Engineering and Informatics,
Istanbul Technical University, Turkey
tosunay@itu.edu.tr

Oscar Dieste

Facultad de Informatica, UPM, Spain
odieste@fi.upm.es

Davide Fucci

Department of Information Processing Science,
University of Oulu, Finland
davide.fucci@oulu.fi

Sira Vegas

Facultad de Informatica, UPM, Spain
svegas@fi.upm.es

Burak Turhan

Department of Information Processing Science,
University of Oulu, Finland
burak.turhan@oulu.fi

Hakan Erdogmus

Carnegie Mellon University, California, USA
hakan.erdogmus@west.cmu.edu

Adrian Santos

Department of Information Processing Science,
University of Oulu, Finland
adrian.santos1987@gmail.com

Markku Oivo

Department of Information Processing Science,
University of Oulu, Finland
markku.oivo@oulu.fi

Kimmo Toro, Janne Jarvinen

FSecure Corporation, Finland
kimmo.toro@f-secure.com, janne.jarvinen@f-secure.com

Natalia Juristo

Facultad de Informatica, UPM, Spain
natalia@fi.upm.es

Abstract

Context: Existing empirical studies on test-driven development (TDD) report different conclusions about its effects on quality and productivity. Very few of those studies are experiments conducted with software professionals in industry. **Objective:** We aim to analyse the effects of TDD on the external quality of the work done and the productivity of developers in an industrial setting. **Method:** We conducted an experiment with 24 professionals from three different sites of a software organization. We chose a repeated-measures design, and asked subjects to implement TDD and incremental test last development (ITLD) in two simple tasks and a realistic application close to real-life complexity. To analyse our findings, we applied a repeated-measures general linear model procedure and a linear mixed effects procedure. **Results:** We did not observe a statistical difference between the quality of the work done by subjects in both treatments. We observed that the subjects are more productive when they implement TDD on a simple task compared to ITLD, but the productivity drops significantly when applying TDD to a complex brownfield task. So, the task complexity significantly obscured the effect of TDD. **Conclusion:** Further evidence is necessary to conclude whether TDD is better or worse than ITLD in terms of external quality and productivity in an industrial setting. We found that experimental factors such as selection of tasks could dominate the findings in TDD studies.

Keywords -- Industry experiment, test-driven development, external quality, productivity

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

An Industry Experiment on the Effects of Test-Driven Development on External Quality and Productivity

Ayşe Tosun
Faculty of Computer Engineering and Informatics,
Istanbul Technical University

Oscar Dieste
Facultad de Informatica, UPM

Davide Fucci
Department of Information Processing Science,
University of Oulu

Sira Vegas
Facultad de Informatica, UPM

Burak Turhan
Department of Information Processing Science,
University of Oulu

Hakan Erdogmus
Carnegie Mellon University

Adrian Santos
Department of Information Processing Science,
University of Oulu

Markku Oivo
Department of Information Processing Science,
University of Oulu

Kimmo Toro, Janne Jarvinen
FSecure Corporation

Natalia Juristo
Facultad de Informatica, UPM

November 18, 2016

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

Abstract

Context: Existing empirical studies on test-driven development (TDD) report different conclusions about its effects on quality and productivity. Very few of those studies are experiments conducted with software professionals in industry. **Objective:** We aim to analyse the effects of TDD on the external quality of the work done and the productivity of developers in an industrial setting. **Method:** We conducted an experiment with 24 professionals from three different sites of a software organization. We chose a repeated-measures design, and asked subjects to implement TDD and incremental test last development (ITLD) in two simple tasks and a realistic application close to real-life complexity. To analyse our findings, we applied a repeated-measures general linear model procedure and a linear mixed effects procedure. **Results:** We did not observe a statistical difference between the quality of the work done by subjects in both treatments. We observed that the subjects are more productive when they implement TDD on a simple task compared to ITLD, but the productivity drops significantly when applying TDD to a complex brownfield task. So, the task complexity significantly obscured the effect of TDD. **Conclusion:** Further evidence is necessary to conclude whether TDD is better or worse than ITLD in terms of external quality and productivity in an industrial setting. We found that experimental factors such as selection of tasks could dominate the findings in TDD studies.

1
2
3
4
5
6
7
8
9 **Keywords**— Industry experiment, test-driven development, external quality,
10 productivity
11

12 **1 Introduction**

13
14
15 Test-driven development (TDD) is the process of writing unit tests before source
16 code so that developers can better understand what the code needs to do, test
17 the preconceptions of how the code will work, and improve the overall design
18 of the code iteratively (Beck, 2003). TDD forces developers think in smaller
19 bits, i.e., in terms of the subtasks of the functionality, which are testable, and
20 redesign the code after implementing each subtask.

21 The key practices of TDD can be summarized as follows (Beck, 2003; Er-
22 dogmus et al., 2005):
23

- 24 1. Decompose the specifications into small, manageable programming tasks.
- 25 2. Write low-level functional tests associated with each task before the pro-
26 duction code.
- 27 3. Get instant feedback from tests to decide whether the task has been fully
28 implemented as intended and whether it interferes with the existing code.
- 29 4. Write minimal pieces of code, necessary to make the tests pass, for the
30 associated task.
- 31 5. Have an up-to-date and complete list of tests in place that are frequently
32 run.
- 33 6. Focus on low-level design and incrementally refactor the production code.

34
35
36
37
38
39 TDD’s industrial relevance is evidenced by its consistently high ranking from
40 software development managers (Emam, 2003). TDD also gained a reputation
41 as a challenging process to apply (VersionOne, 2013). Therefore, TDD has
42 not been adopted much in the industry, despite the fact that companies fre-
43 quently use other agile practices. A recent survey of agile methods usage at
44 the organizational level reports that TDD is not a very popular development
45 methodology (reported as being used by only 1.7% of respondents) among more
46 than 400 professionals from 200 software companies (Rodriguez et al., 2012).
47 Often, professionals select and employ a subset of agile practices (e.g. sprint
48 planning, daily standup, and continuous integration) based on their interests
49 and the effort required to implement a practice. Of those professionals who use
50 agile practices, more than 80% reported that they employed Scrum practices
51 (daily standup, iteration planning, unit testing, self-organizing teams), whereas
52 eXtreme Programming practices were used by 18% of professionals (Rodriguez
53 et al., 2012).
54

55 Yet, there is industry interest in the study of TDD and its effects on cost
56 reduction, productivity, process, and product quality. A longitudinal case study
57
58

1
2
3
4
5
6
7
8
9 at IBM between the years 2001 and 2006 over 10 releases of a software appli-
10 cation revealed that TDD took considerable amount of time at the beginning,
11 but it paid back in terms of more robust code with better quality (Sanchez
12 et al., 2007). Latorre (2014b) reported a successful application of TDD for a
13 period of five and a half months in the industrial environment. The adoption
14 of the developers to this new development approach in (Latorre, 2014b) led
15 to timely product delivery, reductions in the number of residual errors, and
16 easier-to-maintain source code. In terms of productivity, the time spent at the
17 beginning for coordination and requirements analysis were higher in TDD, but
18 the functional testing took much less time due to increased unit testing (Latorre,
19 2014b).

20
21 Causevic et al. (2010) also conducted an industrial survey to examine the
22 expected and actual level of usage for a number of test-related activities and
23 found that “respondents would like to use TDD to a significantly higher extent
24 than they actually do” due to TDD’s claimed benefits in the literature. Our face-
25 to-face meetings with professionals from six software companies in Finland, as
26 part of our on-going project (Juristo, 2016), also revealed that TDD is a highly
27 popular topic, as professionals are interested in learning its key practices and
28 its impact on their development process through experimentation (Tosun-Misirli
29 et al., 2014).

30
31 Empirical studies of TDD look for evidence about the effects of this devel-
32 opment technique on software product and development processes. Researchers
33 have studied TDD through case studies (e.g. Nagappan et al. (2008)), surveys
34 (e.g. Aniche and Gerosa (2010)) and experiments (e.g. Erdogmus et al. (2005)).
35 Existing systematic reviews on TDD grouped the studies with respect to the
36 context (e.g. academia, semi-industrial, industrial (Siniaalto, 2006; Rafique and
37 Mistic, 2013)) and the empirical research strategy (e.g. controlled experiment,
38 case study, pilot study (Kollanus, 2010; Turhan et al., 2010)).

39
40 Even though there are several empirical research studies on the effects of
41 TDD on quality and productivity, the results differ due to contextual factors
42 (Turhan et al., 2010; Rafique and Mistic, 2013). External quality measured
43 during TDD and non-TDD is not significantly different in student-based ex-
44 periments, whereas it improves for TDD in industry experiments (Rafique and
45 Mistic, 2013). Productivity, on the other hand, improves for TDD in academic
46 experiments, whereas it degrades in industrial experiments (Rafique and Mistic,
47 2013). In controlled experiments, different studies also report different results
48 (Kollanus, 2010; Turhan et al., 2010). Thus, the benefits that can be achieved
49 using TDD vary; leading to inconsistent conclusions for both researchers and
50 software professionals. This state of the evidence motivated us to further inves-
51 tigate TDD in the context of software professionals in industry.

52
53 In this research, we report an experiment conducted at three different sites
54 of a software company in order to analyse the effects of TDD on the external
55 quality of the work done and the productivity of the developers who performed
56 the work. 24 professionals from the company attended a three-day training and
57 experimentation session on TDD. We found that TDD did not improve external
58 quality compared to incremental test-last development. Regarding productivity,

1
2
3
4
5
6
7
8
9 the results were confounded by the tasks selected to implement the specified
10 development approach.
11

12 **2 Related Work**

13
14
15 In this section we summarize the results of three literature reviews ((Rafique and
16 Mistic, 2013), (Turhan et al., 2010), (Munir et al., 2014)) on TDD, and in par-
17 ticular discuss the discrepancies among them with respect to how they classify
18 the studies covered. We highlight inconsistent findings with respect to quality
19 and productivity, and lack of industrial experiments in the literature. We then
20 report in detail the only three TDD experiments conducted with professionals
21 in industrial settings.
22

23 **2.1 Literature reviews on TDD**

24
25 Through a systematic literature review, Turhan et al. (2010) summarize empiri-
26 cal studies of TDD (called TDD trials) regarding four aspects. These aspects
27 are the variants of TDD, effort spent on TDD, type of study (pilot, indus-
28 trial use or controlled experiment) and the control technique (test-last or pair
29 programming). They observed the effects of TDD in terms of developer pro-
30 ductivity, test quality, internal code quality, and external (post-release) quality.
31 Based on this review, eight out of 32 empirical studies are classified as con-
32 trolled experiments, four of which were conducted with graduate students or
33 professionals. Due to the differences in study settings and contexts, Turhan
34 et al. (2010) do not formally aggregate the results, but overall findings indicate
35 moderate improvements in terms of external quality using TDD, whereas the
36 effects on productivity are inconclusive.
37

38 Munir et al. (2014) report on the results of a systematic review that identi-
39 fies 41 empirical studies of TDD and analyse them with respect to rigour and
40 relevance scores using an assessment rubric. Based on a prior definition by
41 Ivarsson and Gorschek (2011), rigour concerns adhering to practices of applying
42 and reporting a research methodology, whereas relevance relates to the practi-
43 cal impact and realism of the research setup. Munir et al. (2014) classify 41
44 empirical studies with respect to the research methodology reported in these
45 studies. The majority of the empirical studies covered by the authors (among
46 which 19 are classified as experiments and one is classified as a case study) with
47 high rigour and low relevance scores indicate that there is no difference between
48 TDD and test-last development for a number of variables: productivity, exter-
49 nal quality, internal code quality, number of tests, effort expended for TDD,
50 size of the project and developer opinion. Among the 29 high-rigour studies,
51 only four are identifiable as studies with professionals, three of which are exper-
52 iments with professionals ((Canfora et al., 2006), (George and Williams, 2004),
53 (Geras et al., 2004)), with the remaining one being an empirical study of TDD
54 with a single developer (Madeyski and Szala, 2007). The analysis of the studies
55 classified as case studies and surveys was more conclusive. Based on seven case
56
57
58

1
2
3
4
5
6
7
8
9 studies and two surveys with high rigour and high relevance scores, the authors
10 conclude that the developers perceived an improvement in quality. This finding
11 is consistent with that of Turhan et al. (2010) regarding external quality.

12 Rafique and Misic (2013) conducted a systematic meta-analysis of 27 empir-
13 ical studies that investigate the impact of TDD on external quality and produc-
14 tivity. Eight out of 27 empirical studies were classified as industry experiments.
15 Three out of these eight studies classified themselves as controlled experiments
16 ((Canfora et al., 2006), (George and Williams, 2004), (Geras et al., 2004)), while
17 the rest of the studies classified themselves as “industrial case studies” that re-
18 port on quantitative effects of TDD. Incidentally, Munir et al. (2014) classified
19 the same three controlled experiments ((Canfora et al., 2006), (George and
20 Williams, 2004), (Geras et al., 2004)) as high-rigour experiments with profes-
21 sionals. In this meta-analysis, Rafique and Misic (2013) report that both the
22 quality improvement and productivity drop are much larger in industrial studies
23 than in academic studies. They also observed that improvements in quality are
24 significantly larger when the differences in test effort using TDD and task size
25 were substantial.
26

27 Empirical studies and literature reviews of TDD often use inconsistent ter-
28 minology and a relatively loose definition of experiment. Therefore, the number
29 of controlled experiments in academia and industry varies among different re-
30 views. Upon careful checking, in the lists of empirical studies included in the
31 reviews ((Rafique and Misic, 2013), (Turhan et al., 2010), (Munir et al., 2014)),
32 we identified only three TDD experiments conducted with professionals in an
33 industrial setting: (Canfora et al., 2006),(George and Williams, 2004), (Geras
34 et al., 2004). In the next subsection, we discuss these three TDD experiments
35 in detail since they share the scope with our research.
36
37

38 2.2 TDD experiments with professionals

39 Table 1 summarizes the three TDD experiments in terms of the response vari-
40 ables, subjects, tasks, total effort spent on experimental tasks, training details,
41 metrics, development paradigm used as the control treatment, experimental
42 design, and limitations of these three TDD experiments. The details are as
43 reported by the authors in the respective papers ((Canfora et al., 2006),(George
44 and Williams, 2004), (Geras et al., 2004)).
45

46 Canfora et al. (2006) investigated TDD versus test-after-coding with respect
47 to unit test quality and productivity in a Spanish software house. They did not
48 consider external quality or code quality. Productivity was measured in terms
49 of time (effort) spent per unit test assertion. 28 professionals were involved
50 in the experiment, and they recorded the effort they spent on their own. The
51 participants had on average one year professional experience in the company.
52 The study findings show improvements in unit test quality for TDD but it slows
53 down the development process.

54 George and Williams (2004) investigated TDD versus a waterfall-like method-
55 ology with respect to code quality, external quality and productivity. Three ex-
56 periments were conducted in three different companies. 24 participants with dif-
57
58

	Canfora et al. (2006)	George and Williams (2004)	Geras et al. (2004)	Our experiment
Response variables	Unit test quality and Productivity	Productivity, External and Internal code quality	Effort, Testing, Failure rate from both developer and customer level	External quality and Productivity
TDD is compared with	Test-after-coding (TAC)	Waterfall-like methodology	Test-last	Incremental test-last
Subjects	28 professionals	12 programming pairs	14 corporate IT developers	22 professionals
Demographics	BS in CS, and 5 years experience in Java programming and modelling	Varying experience in TDD and pair programming, employed in three companies	Experience between six months to over six years	BS in CS, MS in CS and EE, more than 6 years of experience in programming (see Section 4.1)
Tasks	TextAnalyzer System	Bowling Alley Scoring Game (adapted from Bowling Scorekeeper)	Project time entry business scenarios	Bowling Scorekeeper, Mars Rover API, MusicPhone
Effort spent on TDD	Both assignments were done in total 5 hours per person (in two consecutive days).	On average 4.3 hours were spent in non-TDD, while 5.2 hours in TDD.	Not mentioned	2.25 hours were spent in non-TDD, TDD tasks and 3 hours were spent in the second TDD task.
Training	Three out of 13 hours (first day)	Only in the third company: Informal training session during whole day with TDD practice in daily work (three weeks prior to experiment)	Training in which half of the participants attended, but quitted without completing the tasks.	Training with all participants including unit testing and basic principles of test-driven development, hands-on exercises using TDD (individual and group-based)
Metrics for the response variables	Productivity: Mean time per assertion, Mean time for writing and executing tests, Total time spent for the assignment Unit test quality: Mean number of assertions per method, Total number of assertions	External code quality: Number of test cases passed Productivity: Total time spent for implementing the tasks Internal code quality: OO metrics, code coverage	Productivity: Time required to develop the task compared to the estimation Testing quality: Test case density(test cases/KLOC), test cases per hour, Test case evaluations per hour Failure: Unplanned failure rate per test case	External quality: Percentage of acceptance tests passed over tackled subtasks Productivity: Percentage of passed assert statements
Design	All subjects implement TDD and non-TDD with randomly assigned tasks in a random sequence (first TDD then non-TDD or vice versa)	Random assignment of each subject in pairs to TDD or non-TDD task	Simple factorial design using two groups	Repeated-treatment design (see Section 3.4)
Limitations	Lack of external quality measures	Small sample size, lack of time constraints, confounding by pair programming	Lack of external quality measures, lack of time constraints during implementation of tasks, confounding by PSP process for process conformance	Task and treatment matching as a confounding factor (see Section 9)

Table 1: TDD experiments in the industry (the information are listed as reported by the study authors.)

ferent levels of experience in programming and TDD participated in these three experiments. There seems to have been no time constraints while completing the tasks. In his thesis George (2002), also reported that the requirements of the tasks were modified after the first experiment in one of the participating companies. The participants worked in pairs, and hence, the study findings might have been affected by the pair programming approach and later modifications in the requirements of the tasks. George and Williams (2004) reported significant improvements in quality for TDD, although it took more time to complete the development tasks.

Geras et al. (2004) investigated TDD versus a test-last approach with respect to productivity, quality of testing process, and failure rate from the customer’s and developer’s perspectives. Fourteen subjects (referred to as corporate IT developers by the authors) with varying programming experience participated in training in the Personal Software Process (PSP). Geras et al. (2004) divided these subjects into two groups (seven subjects in the TDD group and seven subjects in the non-TDD group). The authors do not report any measures relating to product quality, nor do they mention the time spent on completing

1
2
3
4
5
6
7
8
9 tasks. The study findings show that there is little or no difference in productiv-
10 ity, although there are differences in the frequency of unplanned failures. The
11 process that the subjects were supposed to follow in (Geras et al., 2004) was
12 dictated by “scripts” or simple instructions based on PSP. However the authors
13 state that the subjects were not sufficiently familiar with PSP or how to follow
14 these scripts, raising a validity threat related to process conformance and study
15 findings.

16 In this research, we conduct an *industrial experiment with software profes-*
17 *sionals*. We summarize our motivation behind running a TDD experiment with
18 professionals as follows:
19

- 20 • **Inconsistent findings:** As briefly summarized above, the findings on
21 the effects of TDD on developers’ productivity and external code quality
22 differ in academia and industry depending on the subjects, tasks, and
23 treatments used in the experiments (Siniaalto, 2006; Rafique and Masic,
24 2013). The findings in TDD studies with different rigour and relevance
25 scores also come to different conclusions (Munir et al., 2014). The findings
26 of the three TDD industry experiments are also inconclusive, because they
27 all suffer from several validity threats, such as small sample size, lack of
28 detailed information about the sample population, subjective measures of
29 productivity (based on what the participants recorded), and mixed factors
30 confounded with TDD (PSP or pair programming). They did not report
31 the effects of TDD on the external quality of the work, nor did they use
32 the same measures regarding productivity.
33
- 34 • **Few industry studies:** In a recent meta-analysis of the effects of TDD
35 on external quality and productivity Rafique and Masic (2013) classified
36 only eight out of 27 studies as industrial studies. Many case studies and
37 studies with students claim external quality improvements with TDD, but
38 more evidence is needed in industrial contexts (Munir et al., 2014).
39
- 40 • **Very few industry experiments:** We investigated the eight industrial
41 studies listed in (Rafique and Masic, 2013) and found that only three of
42 them were truly controlled experiments, while the rest would be more ap-
43 propriately considered as case studies or surveys with quantitative analy-
44 ses.
45
46

47 Based on these observations, we believe that further evidence is necessary
48 through experimentation on the effects of TDD on quality and productivity.
49 Our study complements and extends previous TDD experiments in industry
50 in several ways. We aimed to increase participation and were successful in recruit-
51 ing more professionals compared to (George and Williams, 2004) and (Geras
52 et al., 2004). We collected detailed demographics data on the participants’ ex-
53 perience in programming, unit testing, and other relevant knowledge prior to
54 experimentation so that we could better characterize the population. We orga-
55 nized a three-day workshop in the company in which two days were dedicated
56
57
58

1
2
3
4
5
6
7
8
9 to training in unit testing and TDD with hands-on individual and group exer-
10 cises to make sure the participants' level of knowledge is appropriate for TDD
11 experimentation. We made sure that all the participants attended the training
12 and completed the control and experimental tasks individually. Compared to
13 the studies by Canfora et al. (2006) and Geras et al. (2004), we set out to in-
14 vestigate more variables, namely the external quality of the work done. These
15 aspects reduced certain internal validity threats related to process conformance
16 (Fucci et al., 2014), mortality, and learning. Our experimental design also differs
17 from the prior studies to fit the context and purpose of the study: We explain
18 our reasoning behind the selection of a repeated-measures design in Section 3.4.
19 Finally, our metrics and their calculations differ from (Canfora et al., 2006)
20 and (George and Williams, 2004) regarding productivity, and from (George and
21 Williams, 2004) regarding external quality. We select more granular measures
22 to increase their reliability. The metric choices are explained in Section 3.2. In
23 the next subsection, we give contextual information about the software company
24 in which the experiment was conducted.
25
26

27 2.3 Context

28
29 The software company in which we conducted our experiment is FSecure. It
30 has been operating at a multinational level for over 25 years and has over 1000
31 employees in 20 sites around the world. It provides online security services and
32 products for protecting digital devices and the digital data of consumers and
33 businesses. The company strives to understand and adopt new trends and tech-
34 nologies in software development as well as in security and digital privacy. FSe-
35 cure has both technical and business-related challenges, such as coping with fre-
36 quently changing requirements, pressure to decrease time-to-market, and pres-
37 sure for increasing quality (Still, 2007). It is one of the leading companies in
38 applying agile development methods (Still, 2007).
39

40 In FSecure, software development teams are highly encouraged to receive
41 training in new development techniques, technologies, and practices that they
42 wish to learn and practice. This openness to innovation and improvement
43 through continuous learning was instrumental in the positive reception by the
44 company of our proposal to conduct an experiment (including training and prac-
45 tice sessions). We conducted our experiment in three different sessions, i.e., at
46 three different FSecure sites: Oulu (Finland), Helsinki (Finland) and Kuala
47 Lumpur (Malaysia).
48

49 3 Experimental Design

50 3.1 Research Objectives

51
52 Using the Goal-Question-Metric-based template suggested by Basili (1992), we
53 define the goal of this research as follows:
54

55 Analyse *TDD and incremental test-last approach*
56
57
58

1
2
3
4
5
6
7
8
9 for the purpose of *comparing them*
10 with respect to the *external quality* of the resulting work and *productivity* of
11 developers
12 from the point of view of the *researchers*
13 in the context of *a software company*.

14 Our research questions are listed below:

- 15 • Does TDD affect external code quality compared to an incremental test-
16 last approach?
- 17 • Does TDD affect developers' productivity compared to an incremental
18 test-last approach?

19
20
21 With these research questions in mind, in the remainder of this section, we
22 discuss the study design and the process followed based on the proposed report-
23 ing structure for experiments by Jedlitschka and Pfahl (2005). We expand the
24 structure in Sections 3.8, 4.2 and 3.10 (Instrumentation, Preparation and Inter-
25 pretation) to clarify our approach regarding the metrics, training, discussion on
26 the results and limitations.
27
28

29 3.2 Variables

30
31 The **independent variable** in this study is the development approach which
32 we tested with two treatments: test-driven development (TDD) and a control.
33 We compare TDD with a closely related process, which we call *incremental*
34 *test-last development* (ITLD). This is the control level. Both TDD and ITLD
35 follow the same, iterative steps except the order of the activities involved in
36 each increment. Both TDD and ITLD follow small steps, such as decomposing
37 the specification into small programming tasks, coding, testing and refactoring.
38 The difference is mainly in the sequencing of coding and testing activities in
39 each increment. TDD prescribes writing tests before writing production code
40 for any piece of new functionality. ITLD prescribes writing production code
41 first, immediately followed by writing tests before moving on to a new, small
42 piece of functionality. We have chosen ITLD as the control since when subjects
43 perform testing at the end (non-incrementally), they tend not to perform it at
44 all (Maximilien and Williams, 2003; George and Williams, 2003). Consequently,
45 we may end up comparing TDD with a process with no testing, and no quality
46 control (Erdogmus et al., 2005). ITLD forces a control that involves testing,
47 making the comparison less biased and fairer.
48

49 The effectiveness of the programming approach can be examined from dif-
50 ferent perspectives. Previous related research has used several dependent vari-
51 ables and metrics, such as productivity, external quality, internal code quality,
52 effort/time and conformance (Munir et al., 2014).
53

54 In this experiment, we adopt the same variables and metrics used in (Er-
55 dogmus et al., 2005) and (Fucci and Turhan, 2013). We study *external quality*
56 (QLTY) and *productivity* (PROD) as **dependent variables**. Using the same
57
58
59
60
61
62
63
64
65

variables reduces the risks of experiment operationalization (the previous studies have successfully vetted and studied the adopted variables) and eases the comparison with relevant previous studies.

We calculate the metric for external quality based on the number of tackled subtasks ($\#tst$) for a given task. We consider a subtask as *tackled* if at least one assert statement in the acceptance test suite associated with that subtask passes. This criterion is used to objectively distinguish subtasks in which a subject put reasonable effort into completing them from other subtasks in which a subject put in little or no effort. We calculate $\#tst$ using Equation 1. In the equation, n is the total number of subtasks comprising the measured task.

$$\#tst = \sum_{i=0}^n \begin{cases} 1 & \#ASSERT_i(PASS) > 0 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

We use $\#tst$ to calculate $QLTY$ in Equation 2 .

$$QLTY = \frac{\sum_{i=1}^{\#tst} QLTY_i}{\#tst} \quad (2)$$

where $QLTY_i$ is the quality of the i^{th} tackled subtask, and is defined as:

$$QLTY_i = \frac{\#Assert_i(Pass)}{\#Assert_i(All)} \quad (3)$$

$\#Assert_i(Pass)$ represents the number of JUnit assertions passing in the acceptance test suite associated with the i^{th} subtask.

The metric for productivity $PROD$ captures the amount of work successfully performed by the subjects. The metric is calculated as follows:

$$PROD = \frac{\#Assert(Pass)}{\#Assert(All)} \quad (4)$$

We consider a passing JUnit assertion to be the smallest quantifiable evidence of work performed. It is therefore the adopted unit of work for measuring productivity.

There are several metrics representing productivity in the literature, such as total time spent implementing a task, number of lines of code (LOC) produced by a developer, number/percentage of user stories implemented, or number/percentage of passing test cases. Bergersen et al. (2014) emphasize the challenge of dealing with the quality of the solution and the time spent on delivering that solution when measuring programmer performance. They mention two main strategies to address the underlying trade-offs: 1) time is fixed, and quality is measured based on successfully implemented steps, 2) time is relaxed, but a high-quality solution is expected and the work is not deemed complete unless that quality expectation is met. In our design, the amount of time allowed to implement the tasks was fixed, and hence, the productivity measure could not be based on the amount of time. If we measure only LOC as the output of the process tested, subjects who produce more LOC but of less quality

1
2
3
4
5
6
7
8
9 could be considered more productive (Madeyski and Szala, 2007); this is not
10 desirable. Similarly, if we choose number/percentage of user stories or subtasks
11 implemented, subjects who claim to have implemented more subtasks could be
12 considered more productive without supporting objective evidence, for example
13 even when most of the acceptance test cases associated with the subtasks do
14 not pass.

15 Therefore, following the recommendations of Bergersen et al. (2014), we
16 chose to base the productivity metric on the number/percentage of passing test
17 cases. Furthermore, to have a more granular and differentiating approach, we
18 modified this metric and measured productivity based on the number of JUnit
19 assert statements (assertions) passing over all assert statements in all test cases
20 associated with the acceptance test suite of a task (see Equation 4).
21

22 3.3 Hypotheses

23 We have two hypotheses, H_Q , H_P , that concern external quality (QLTY) and
24 productivity (PROD), respectively. To use one-tailed hypotheses, knowledge
25 (preferably in the form of theories) of the direction of the effect under study is
26 necessary. Since we do not anticipate the direction of the potential effects, we
27 use two-tailed hypotheses.
28

29 H_{Q0} : $\mu(QLTY)_{ITLD} = \mu(QLTY)_{TDD}$ (Null Hypothesis)

30 H_{Q1} : $\mu(QLTY)_{ITLD} \neq \mu(QLTY)_{TDD}$ (Alternative Hypothesis)

31 H_{P0} : $\mu(PROD)_{ITLD} = \mu(PROD)_{TDD}$ (Null Hypothesis)

32 H_{P1} : $\mu(PROD)_{ITLD} \neq \mu(PROD)_{TDD}$ (Alternative Hypothesis)

33 3.4 Design

34 Our experiment used *repeated-treatment design* (Shadish et al., 2001). In this
35 section, we report on the process of deciding the design of our experiment,
36 explain alternative designs that we ruled out based on validity threats and
37 context characteristics, and justify the selection of the design.
38

39 The most straightforward alternative is the basic randomized between-subjects
40 design (shown in Table 2). Randomized design is conceptually simple and allows
41 for the control of nuisance factors and threats to validity such as maturation
42 and fatigue. The configuration in Table 2 is appropriate if either of the following
43 circumstances hold:
44

- 45 1. All of the subjects are familiar with ITLD, and some of the subjects are
46 already experienced with TDD. We assign a sample of these subjects to
47 the treatment group, G2, and a sample of the remainder (the ones without
48 any experience with TDD) to the control group, G1.
- 49 2. All of the subjects are familiar with ITLD, and none of the subjects are
50 experienced with TDD. In this case, we train a sample of the subjects in
51 TDD and assign them to G2, with the remaining subjects assigned to G1.
52
53
54
55
56
57
58

Group	Experimental Session
G1	ITLD (control treatment)
G2	TDD (experimental treatment)

Table 2: Two-level, basic between-subjects design

In our case, neither circumstance applied: Not all subjects were experienced in TDD, and were sufficiently familiar with ITLD. Incentivizing the experiment as free training (Tosun-Misirli et al., 2014) was necessary to secure our partner’s cooperation and reduce variation, but this measure attracted only participants without any knowledge of TDD. Moreover, the employer of the subjects did not want only a subset of the participants to receive training to prevent compensatory rivalry (a social threat to validity). So we had to provide at least TDD training to all participants, and this prevented us from using a simple randomized between-subjects design: Carry-over may happen when all participants (G1 and G2) are trained in the treatment (TDD) and assigned randomly to two groups (experiment and control), since the control group is also exposed to the treatment even if they are not supposed to use it. Therefore, a non-synchronous execution, for example, running ITLD first (pre-treatment) and TDD after (post-treatment) appeared to be the best design alternative that matched both the researchers’ and industrial partner’s goals.

We had two alternatives for pre-/post-treatment design. The simpler one, still a between-subjects design, is shown in Table 3. A disadvantage of between-subjects designs in general is that, to achieve comparable power, they require larger numbers of participants than alternative designs in which all subjects perform all treatments. However, we had no assurance that a large number of subjects would volunteer in the organization.

The sample size can be reduced by controlling the sources of variation in the sample. One of the most important sources of variation is the skill level. This variation can be controlled using a repeated-treatment design. Table 4 shows the experimental configuration for this type of design in our case.

Group	Temporal sequence			
	Training	Exp. Session	Training	Exp. Session
G1	ITLD	ITLD	TDD	TDD
G2		ITLD		

Table 3: (Pre-Post) Between subjects design with the training course

Group	Temporal sequence			
	Training	Exp. Session	Training	Exp. Session
G1	ITLD	ITLD	TDD	TDD
G2		ITLD		TDD

Table 4: Repeated-treatment design with the training course

1
2
3
4
5
6
7
8
9 In a repeated-treatment design, subjects are all matched with themselves,
10 cancelling out any inherent variability and increasing power greatly (Shadish
11 et al., 2001). A power analysis using a generalized linear model for the design
12 in Table 3 shows that 128 subjects (64 per group) are necessary to generate
13 a medium-sized effect (Cohen’s $d=0.5$) with a resulting chance of Type-I and
14 Type-II errors of $\alpha = 0.05$ and $\beta = 0.2$, respectively (Cohen, 1992). If the
15 assumptions of normality and homoscedasticity are not met, the required sample
16 size would be even larger. Using the same parameters in a repeated-treatment
17 design, on the other hand, the sample size drops to 34 subjects. We decided
18 that this sample size was more reasonable in the context of our industry partner,
19 and in turn, we adopted the design provided in Table 4. We present our final
20 design, with training-treatment sequences and the assigned tasks for treatments,
21 in Section 3.6 (Table 5).
22

23 3.5 Experimental objects/tasks

24 The subjects implemented three programming tasks in total. The first task was
25 called MarsRover API and used with ITLD, and the second and third tasks,
26 Bowling Scorekeeper and MusicPhone, were used with TDD. We discuss each
27 task below.
28

29 MarsRover API (MR) was the only **control task**. It is a greenfield program-
30 ming exercise that requires the development of a public interface for controlling
31 the movement of a fictitious vehicle on a grid with obstacles. MR is a popular
32 exercise used by the agile community to teach and practice unit testing. This
33 task was described in terms of six requirements. We split these requirements fur-
34 ther into 11 fine-grained subtasks, each associated with a set of acceptance tests
35 unknown to the subjects. The description of these requirements also included
36 an example of a simple acceptance test. The complete description of the task
37 provided to the subjects can be found in this link: 10.6084/m9.figshare.3502808.
38

39 MR is an algorithm-oriented task. The implementer needs to handle several
40 edge cases in order to produce the expected results. The implementation of MR
41 leverages an $N \times N$ matrix data structure representing the planet on which an
42 imaginary rover moves. Each matrix cell may store an obstacle on the surface
43 of the planet. Obstacles do not have any behaviour and can be modelled with
44 simple data types (e.g. a Boolean for representing presence/absence). There are
45 six main operations to implement, necessary to move the rover on the surface
46 of the planet. The task can easily be solved using just one class. The possible
47 operations are:
48

- 49 • Matrix initialization and assignment of obstacles to cells
- 50
- 51 • Command parsing
- 52
- 53 • Forward and backward moves
- 54
- 55 • Left and right turns
- 56
- 57
- 58

1
2
3
4
5
6
7
8
9 The forward and backward moves are the most complex operations. Com-
10 mand parsing and left/right turns are straightforward operations. The assign-
11 ment of obstacles to cells upon initialization requires some parsing and type
12 casting. MR is not a particularly difficult task, although it may require a few
13 cycles of debugging.

14 We provided the MR specification document and a Java project template to
15 the subjects in order to get them started easily and to have a common package
16 structure that would make data collection easier to automate. The project
17 template consisted of 17 lines of non-commented code (LOC), a Java class that
18 exposes the signature of the public API required by the task (8 LOC), and a
19 test class containing the stub of a JUnit test case (9 LOC).

20 The **first experimental task** tackled by the subjects using TDD approach
21 was a modified version of Robert Martin’s Bowling Scorekeeper (BSK, 2015).
22 This task has also been popular in the agile community, and was used in previous
23 TDD experiments (e.g. (Erdogmus et al., 2005), (Fucci and Turhan, 2013),
24 (Williams et al., 2003)). The goal of the task is to calculate the score of a
25 single bowling game. The task is algorithm-oriented and greenfield. It does not
26 involve the creation of a UI. The specification was broken down into 13 fine-
27 grained subtasks. The task does not require prior knowledge of bowling scoring
28 rules: this knowledge is embedded in the specification.

29 Each subtask of BSK contained a short, general description, a requirement
30 specifying what that subtask is supposed to do, and an example consisting of an
31 input and the expected output. We instructed the subjects to follow the given
32 order of the subtasks while implementing them. BSK has also four principal
33 operations:
34

- 35 • Add a frame or bonus throws
- 36
- 37 • Detect when a frame is a spare or strike
- 38
- 39 • Calculate a frame score
- 40
- 41 • Calculate the game score
- 42

43 The most complex operation is the calculation of the frame score. It depends
44 on the type of frame (regular, spare or strike), the position of the frame in the
45 game, and whether or not the next frame is a strike.

46 We provided the BSK specification document and a code template to the
47 subjects. The code template contains two Java classes, one with 23 LOC and
48 the other with 28 LOC, each with the method signatures necessary to exercise
49 our acceptance tests. We also provided the stub of a JUnit test class (9 LOC).

50 Both MR and BSK are relatively straightforward greenfield tasks, with some
51 intricate logic that requires attention. In BSK, the complexity was represented
52 by the tricky logic of bonus throws, the handling of frames, and interactions
53 between the scores of subsequent frames. Comparing structural complexities
54 between MR and BSK is, in fact, open to discussion; we believe they are com-
55 parable in many aspects. The process of managing, maintaining and acting
56

1
2
3
4
5
6
7
8
9 upon several possible system states is similar in MR and BSK. Rover move-
10 ments in MR and score calculation in BSK both depend on previous states of
11 the system.

12 The **second experimental task** tackled by the subjects using TDD was
13 a brown-field project called MusicPhone (MP). MP is an application that is
14 intended to run on a GPS-enabled, MP3-capable mobile phone. It resembles a
15 real-world system with a three-tier architecture (graphical user interface, busi-
16 ness logic, and data access). The system consists of three main components that
17 are created and accessed using the Singleton pattern. Event handling is imple-
18 mented using the Observer pattern. We provided a description of the legacy
19 code, including existing classes, their APIs, and a diagram of the system's ar-
20 chitecture to the subjects (see the link: 10.6084/m9.figshare.3502808).
21

22 MusicPhone (MP) has very different characteristics than the other two tasks.
23 It was designed to address the concern of realism, as a counterbalance against
24 a potential bias in favour of TDD when it is applied to greenfield tasks. Our
25 intention was to move the task away from TDD's commonly believed sweet
26 spot of greenfield tasks by embedding it in a more realistic context that involves
27 externalized components and interactions with these components.

28 Due to its architecture, MP's complexity is much higher than that of BSK or
29 MR. The subjects had to implement four operations in the following sequence:

- 30 • Compute the distance between two geographical coordinates (given the
31 formula)
- 32
- 33 • Recommend artists
- 34
- 35 • Find concerts for artist (given the recommendations)
- 36
- 37 • Compute an itinerary for the concerts
- 38

39 All operations of MP are moderately complex. The simplest one, computing
40 the distance between coordinates, is essentially about correctly implementing a
41 mathematical formula, but involves several edge cases that should be addressed.
42 The remaining operations are not self-contained and require the collaboration
43 of existing subclasses.

44 The system provided to the subjects had a working UI. The data access
45 layer was also implemented. In the partial implementation provided, attempting
46 to access a missing function via the UI (e.g. by clicking a button) throws an
47 exception and displays an error message. Subjects had to implement the missing
48 operations in the business logic layer.

49 MP's partial implementation consists of 13 classes and four interfaces (1033
50 LOC) written in Java. The package in which the subjects need to implement
51 the requirements contains three classes (92 LOC): a Singleton class (27 LOC),
52 an exception handling class (7 LOC), and the class where the missing operations
53 should be implemented (58 LOC). Along with the scaffolding of the system, a
54 single smoke test (6 JUnit assertions, 38 LOC) is also provided. The test cases
55 in the smoke test show how different components communicate with each other
56 and how the API of the existing classes should be used.
57
58

3.6 Assignment of tasks to treatments

The design in Table 4 shows that all subjects sequentially apply ITLD and TDD. This experimental configuration suffers from a learning threat (maturation) on the experimental object. The learning effect can be avoided by requiring the implementation of two different tasks (say A and B) during the experiment. Tasks A and B can be assigned to two groups/sessions (G1 and G2) in a counter-balanced manner, e.g., A is assigned to G1-ITLD and G2-TDD, and B is assigned to G2-ITLD and G1-TDD. Counter-balance is the preferred approach because it separates the potential effect of the task from the development approach, or independent variable. But such an experimental configuration would suffer from another threat: maturation. This is because the task definition and other information may be transferred among the groups before TDD, favouring subsequent TDD usage in the experiment. This is a threat to internal validity, and the fact that the experiment would be part of a training course and both groups would be present in the same room during the whole training and experiment would compound the threat.

We can rule out such a threat by assigning A to both groups in ITLD (G1-ITLD, G2-ITLD) and B to both groups in TDD. Information transfer in such a case is impossible, because the subjects implement tasks A and B sequentially. However, this implies an additional cost: tasks and development approaches get confounded and the effects can no longer be separated. In case of an interaction between the task and the treatment, the experiment results could be severely biased. For instance, imagine that task B is more complex than A; then poor results during TDD may be explained either as TDD's poor performance or B's complexity, but we would not be able to decide which is true. This is essentially an instrumentation threat to (internal) validity. We need to make a decision: which threat to validity (learning effect or instrumentation) is more severe in our experimental context?

We considered learning effect to be a critical issue. Although we discouraged the participants from discussing the exercises and tasks during the breaks, they would meet before or after the sessions and work together, and therefore could leak information about the teaching materials, exercises, and tasks. We had no control over this behaviour. However, we could control instrumentation threats to a certain extent.

There were other, more practical reasons for choosing to accept an instrumentation threat over a learning threat in this experiment. The company did not set any restrictions on experimental tasks, but expressed a strong desire to get maximum benefits from their participation. This implied that we should expose the volunteers to as many tasks and with as much variety as possible. The managers also did not want certain volunteers to be excluded from certain sessions, because this could make them feel that they were getting fewer benefits and become less motivated. The organization did not want to risk these kinds of resentment-based perceptions, which also pose an internal threat, known as *compensatory rivalry*. We wanted to avoid compensatory rivalry as well. These practical considerations combined with the risk of uncontrollable learning effects

made us choose two tasks of comparable complexity for alternative treatments.

TDD has often been criticized as being suitable mostly for self-contained, *greenfield* tasks that can be broken down into smaller subtasks easily (Erdogmus et al., 2005; Rafique and Misis, 2013). These kinds of tasks are also ideal for experimental settings, since subjects do not need to familiarize themselves with the design of the system and can proceed immediately. Detractors of TDD claim that it is unclear how TDD could perform in *brownfield* tasks, under conditions of increased realism (Causevic et al., 2011).

The uncertainty around the task effects on TDD called for a more conservative approach to the assignment of tasks to the treatments. Instead of assigning only one type of task during TDD, we decided to use both a greenfield and a brownfield task. TDD would be applied first to the greenfield task, and later on to the brownfield task.

In our design, the natural choice was to introduce the tasks that help subjects master incremental testing since this aspect is common to both treatments. These tasks would then serve as baselines. MR and BSK are, by design, suitable for this purpose. Therefore, we selected them as the control (ITLD) and first experimental task (TDD), respectively.

For the second experimental task, on the other hand, we chose MusicPhone as it is not a task in the sweet spot of TDD. We took this position to avoid a potential bias in favour of the investigated phenomenon. The caveat is the risk of introducing a reverse bias, that is, the risk of being too conservative. Our decision was to err on the side of caution.

Hence we ended up with three tasks in total: a greenfield task for ITLD, a comparable greenfield task for TDD, and a third task of a different nature, a brownfield task, for TDD again. The resulting configuration is illustrated in Table 5.

Training	1 st Exp. Session	Training	2 nd Exp. Session	3 rd Exp. Session
ITLD	ITLD <i>greenfield task</i> MarsRover	TDD	TDD <i>greenfield task</i> Bowling Scorekeeper	TDD <i>brownfield task</i> MusicPhone

Table 5: Repeated-treatment design with the training course and the assigned tasks

3.7 Selection of subjects

We used convenience sampling to select the subjects of our experiment, since our industry partner preferred that the developers could register for the training/experiment voluntarily. All the volunteered subjects implemented all the tasks, so there was no randomization involved in our experiment.

3.8 Instrumentation

We provided a technological infrastructure to the subjects that includes the tools that were used during the training and experimentation sessions. This

1
2
3
4
5
6
7
8
9 infrastructure was embedded into a virtual machine (VM) image through Ora-
10 cle VM Virtual Box (Vir, 2014). The image included the Windows 7 operating
11 system, a web browser, Eclipse Helios (Ecl, 2014) as the development environ-
12 ment, JUnit 4.3 as the unit testing framework installed inside Eclipse (Gamma
13 and Beck, 2014), and a plug-in called Besouro that allowed us to collect process
14 conformance data in order to assess whether the subjects followed the TDD
15 process to a reasonable extent (Becker et al., 2014). Each subject was asked to
16 install Oracle VM VirtualBox and download the virtual machine we prepared
17 before the experimentation/ training week. They used a VM for three reasons:
18 (1) To isolate the development environment used for the experiment from the
19 real environment that the subjects used for their daily work; (2) to control
20 the technology that the subjects used; and (3) to collect data more easily and
21 uniformly.
22

23 We also provided the subjects three Java project templates (one for ITLD
24 and two for TDD tasks) inside the Eclipse environment. These project templates
25 helped the subjects to ramp up more easily and reduce the time required. They
26 included project setup configurations, some class and method signatures, debug
27 and test configurations). Furthermore, we distributed hard copy versions of the
28 project specification documents to all subjects before each task.
29

30 The pre-test instrument for collecting the demographic information about
31 the subjects was provided in the form of a Google survey whose link was shared
32 with the subjects on the first day of the training.

33 The main instrument used for extracting the QLTY and PROD metrics was
34 acceptance tests. A set of acceptance tests was written for all tasks. For the
35 MarsRover API, the acceptance tests were written by the researchers, whereas
36 we adapted the tests for MusicPhone and Bowling Scorekeeper from a previous
37 experiment (Erdogmus et al., 2005). MusicPhone’s original test suite was writ-
38 ten in the C# language and hence, we translated it to Java. The acceptance
39 test suites of two tasks (MarsRover API and MusicPhone) had 11 JUnit tests
40 cases each; the suite for the third task (Bowling ScoreKeeper) had 13 JUnit
41 test cases. Each test case contained a varying number of JUnit tests, and each
42 test contained a varying number of assert statements. Table 6 summarizes the
43 composition of the test suites.
44

45 3.9 Data Collection & Measurement Procedures

46 We designed the experiment as a three-day training course with a planned
47 **schedule** shown in Figure 1. The schedule and the content of the training
48 were shared with the participants prior to the training/experimentation days.
49 We informed them about the possibility of changing the break times for coffee
50 and lunch depending on their preferences. However we clearly stated that there
51 would be no break during the implementation of control and experimental tasks,
52 and asked them follow this rule.
53

54 As shown in Figure 1, the training sessions included several exercises. These
55 exercises are called “katas” (Draper, 2006) and were implemented in a solo or
56 “randori” (a sequence of pair programming sessions rotated every 5-10 minutes
57
58
59
60
61
62
63
64
65

Subtask	MarsRover API		MusicPhone		Bowling Scorekeeper	
	Tests	Assert	Tests	Assert	Tests	Assert
ST1	1	1	4	4	3	3
ST2	7	11	3	12	3	3
ST3	4	8	3	12	2	2
ST4	4	7	4	4	3	10
ST5	4	8	10	26	5	5
ST6	8	8	8	12	6	6
ST7	6	15	7	17	8	8
ST8	4	8	1	1	5	5
ST9	3	8	2	11	5	5
ST10	3	7	2	13	4	4
ST11	8	8	4	20	2	2
ST12	NA	NA	NA	NA	3	3
ST13	NA	NA	NA	NA	2	2
<i>Total</i>	52	89	48	132	51	58

Table 6: Summary of acceptance tests used to calculate the metrics for each task used.

(Draper, 2006)) fashion by the participants during the training. We carefully selected each of these katas to present how ITLD and TDD could be applied to different kinds of problems (e.g. when there is legacy code that needs to be modified for a new subtask, when the subtasks are not clearly defined, etc.). The time allocated to implement the second experimental task, MusicPhone, was higher (180 minutes) than the time allocated to implement the control task, MarsRover API (135 minutes). The main reason behind this was the complexity of the experimental task.

As described in the technological infrastructure in Section 3.8, we planned to collect data from the subjects. The data included project folders consisting of production and test code written by the subjects, activity logs from Besouro (Becker et al., 2014), and daily snapshots of the VM image. The raw data collected from each subject’s computer were stored on hard drives located on-site. The pre-test questionnaire for collecting the demographics information was online, and the responses were automatically collected and stored in the cloud.

After we collected the raw data from the VM images, we processed the data to calculate the QLTY and PROD metrics for each subject and task. This process worked as follows: We executed the acceptance test suite associated with each task on the production code of the subjects. We stored the number of assert statements passing and failing during this execution as base measures. There was a one-to-one mapping between the JUnit test cases and subtasks. This mapping allowed us to compute the number of tackled subtasks for each subject and each task, and finally the QLTY metric according to Equation 1 and the PROD metric according to Equation 4.

	DAY 1	DAY 2	DAY 3
15 minutes	Intro	TDD Training	TDD Task - MR
15 minutes	Demographics questionnaire		
60 minutes	UT Training	Coffee Break	
15 minutes	Coffee Break		
75 minutes	ITLD Kata (Random)	TDD Kata (Random)	
60 minutes	Lunch		
60 minutes	ITLD Kata (Random)	TDD Kata (Random)	TDD Kata (Random)
15 minutes	Coffee Break	Coffee Break	
135 minutes	ITLD Task - MR API	TDD Task - BSK	Retrospective
30 minutes			

Figure 1: The experimental schedule (planned)

3.10 Analysis Approach

We perform a two-stage analysis to answer our hypotheses. **Stage 1** aims to check the effect of the development approach on QLTY and PROD. We use IBM SPSS Statistics Version 22 to apply a repeated-measures General Linear Model (GLM) procedure. This procedure allows us to perform a repeated-measures ANOVA on three levels of the independent variable corresponding to two treatments: ITLD and TDD. The repeated measures GLM assumes that the form of the covariance matrix of the dependent variables is circular/spherical, i.e., that the covariance between any two elements is equal to the average of their variances minus a constant (Winer, 1971). To check this assumption, we use Mauchly's test of sphericity. If the sphericity assumption is met, we use the univariate test, as it is statistically more powerful than the multivariate test. If the sphericity assumption is not met, we use two tests: (1) a multivariate test, and (2) a univariate with Huynh-Feldt correction (Field, 2007) to check whether both of them yield the same results.

If the repeated-measures GLM indicates a significant difference between the treatments, we examine pairwise comparisons between ITLD and TDD while keeping the task complexity constant. Thus, we compare the two treatments with the simple tasks (MR versus BSK). We hypothesize that the two treatments

1
2
3
4
5
6
7
8
9 differ when applied to comparable tasks and the simpler of the tasks.

10 We also suspect that the change in task complexity may affect the results as
11 increasing the task complexity in the experimental treatment (TDD) may cause
12 its performance to decline and partially or wholly cancel its benefits. Therefore
13 we examine pairwise comparisons between TDD with the simple (BSK) and
14 complex (MP) tasks. If there is a significant difference in the performance of
15 TDD with two tasks, it is more likely that the task complexity is confounded
16 by the development approach. If there is no significant performance difference
17 between the two TDD tasks, we could say that the difference is due to TDD or
18 ITLD. If we observe a difference between the two tasks (BSK versus MP) at the
19 same treatment level (TDD), we move to the second stage analysis.

20 **Stage 2** is performed only if the first stage analysis warrants it, i.e., if
21 the task complexity impacts TDD’s performance. In this second stage, we
22 use *task type* as an additional factor in our treatment (development approach)
23 and build a marginal model. We use SPSS to apply a Linear Mixed Effects
24 (MIXED) procedure. This procedure allows us to include the task type as a
25 factor nested in the experimental treatment and to build a population-averaged
26 model. The difference between the MIXED procedure and GLM is that there
27 is no constant variance, and the form of covariance matrix of the dependent
28 variables is assumed to be independent of their variances (C.E. McCulloch and
29 Searle, 2000). If the MIXED model confirms that task is a significant factor over
30 the development approach, it is not possible to conclude that the difference is
31 solely due to the treatments, i.e., contextual factors matter.

32 After running the statistical tests, we report the statistical significance and
33 the observed power (of an apparent relationship) (Ellis, 2010). We also report
34 the effect size in terms of a partial eta-squared statistic and Cohen’s *d* statistic
35 to gauge the magnitude of the apparent relationship (Coe, 2002).
36
37
38

39 3.11 Evaluation of design validity

40 Jedlitschka and Pfahl (2005) recommend that potential threats to the experi-
41 ment validity due to the choice of the design are discussed and reported sepa-
42 rately. We cover them in this section.

43 A repeated-treatment design may be exposed to the following threats to
44 validity (Shadish et al., 2001): fatigue, carry-over/order, period effects, and
45 practice effects. We believe that these threats were negligible, beneficial, or
46 likely cancelled out each other:
47

- 48 • **Fatigue:** We conducted experiments on each site over three days during
49 regular working hours. During this period, the developers could not per-
50 form normal work activities; their only responsibility was participation in
51 the experiment and the related training. Therefore, the experiment did
52 not induce any extra effort on the subjects. In fact, their schedule during
53 the experiment was more relaxed than a regular workday. The participants
54 may have felt more energetic closer to the beginning of the training, and
55 as the training progressed, could get more fatigued and less motivated af-
56

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

fecting their performance. We could not avoid this natural reaction to the working week, and we do not believe it could have had a significant effect. If any, it would have only biased the findings slightly against TDD, but this effect is likely to have been cancelled out by learning and carry-over.

- Carry-over/Order: A carry-over between the control, ITLD, and the treatment, TDD, is a possibility, and may normally raise a threat. In this case, the carry-over and order threats coincide, but they were required and beneficial in our case. ITLD is normally the baseline case for learning TDD, since it more closely reflects the traditional way of thinking. After mastering ITLD, developers can learn TDD more easily. After learning TDD, unlearning it to revert to a more traditional way of thinking could be difficult. Progressing to TDD through ITLD is thus the natural order. The design in Table 4 reproduces this order: The subjects can build upon the ITLD expertise when they apply TDD. Carry-over from ITLD to TDD is therefore required and desirable in the design used.
- Period effects: Beyond fatigue and carry-over, we could not identify any plausible interaction between the treatments and the periods of administration of the treatments, i.e., the fact that ITLD was applied on the first day and TDD on the second and third days does not seem to make any difference, since there were no notable differences between the days of the training.
- Practice effects: As a consequence of a repeated-treatment design, carry-over and practice threats may normally be confounded. Practice threat is plausible when subjects are inexperienced and the repeated execution of treatments implies learning. In our experiment, the participants stated that they were familiar with traditional test-last development, but they still improved their abilities in incremental development and unit testing with hands-on exercises (one randori exercise, one task) during the first day of the experiment. Hence any practice effect of ITLD on TDD, as in the case of carry-over, was beneficial, and likely contributed to the experiment success. This is also a natural consequence of accepting instrumentation threat over learning threat, as we discussed in Section 3.6.

4 Experiment Execution

The experiments took place between September and December 2013 at three different sites. As planned earlier, three days were allocated for each site (Oulu, Helsinki and Kuala Lumpur). One trainer, one observer, and one data collector were present in all three experiments.

4.1 Sample

We had 24 subjects: Seven in Oulu, 11 in Kuala Lumpur, and six in Helsinki, who attended the whole training and experimentation. We conducted a demo-

graphics survey to get more detailed information about the subjects' academic background, professional career, experience in programming, programming language used in the experiments (Java), JUnit, and TDD. Table 7 presents a summary of the survey responses. More than 80% of the participants had six or more years of programming experience, whereas around half of the participants had two to ten years of experience in the programming language used in the experiment. All of them were familiar with JUnit, and half had no knowledge on TDD. The other half of the subjects indicated that they attended training/workshops on TDD, but never applied this practice in the professional development.

Table 8 also provides more information about the education and level of degree, and the most frequently used programming languages, unit testing tools, development environments, and development methodologies. We observed that the majority of the participants had a bachelor's degree in computer science, computer engineering or electrical engineering. Around 40% (ten out of 24) also had a master's degree. The most commonly used programming language and unit testing tool were Java and JUnit, respectively. We see that the subjects also used other testing frameworks (e.g., Jasmine (15%) and Nose (13%)) and IDEs (e.g., IntelliJ, 22%, and PyCharm, 18%), which reflects their preferences for languages other than Java. Regarding the development methodology, 17 out of 24 subjects (71%) indicated they used agile practices in their daily work.

# Years	Prof. career	Prog. exp.	Prog. lang. exp.	JUnit exp.	TDD exp.
0	0	0	0	0	50.0
≤ 2	9.1	0	18.2	45.5	31.8
$3 \leq 5$	18.2	18.2	27.3	36.4	18.2
$6 \leq 10$	45.5	40.9	36.4	13.6	0
> 10	27.3	40.9	18.2	4.5	0

Table 7: Summary of demographics for the subjects (in percentage)

4.2 Preparation

Schedule. At the beginning of the training, we presented the planned schedule (Figure 1) to the participants. Start and end times for each day as well as the breaks were discussed and agreed upon by all.

Training. In order to make sure that all participants shared a common baseline understanding of the testing and TDD-related concepts, we provided crash courses at all sites. These training sessions included: lectures (two hours), hands-on group exercises (four hours) and hands-on individual exercises (12 hours). As shown in Figure 1, the training sessions alternated with the treatments.

- **Lectures:** We delivered two lectures, each one hour long. The first lec-

Education	Degree	Prog. lang.	Unit testing tool	IDE	Methodology
Computer Science & Eng. (11)	BS (11)	Java (10)	JUnit (16)	Eclipse (6)	Agile (17)
Electrical Eng. (5)	MS (10)	C++ (5)	Jasmine (6)	IntelliJ (5)	Waterfall (4)
Other (6)	MA (1)	Python (4)	Nose (5)	PyCharm (4)	Iterative (1)
		Other (3)	Other (13)	Text editors (4)	
				Other (3)	

Values in parentheses indicate the number of subjects selected the corresponding category.

Table 8: Detailed demographics for the subjects

ture took place at the very beginning of the study (morning of Day-1) and covered the basic principles of unit testing. We compiled the training content from a combination of our own software testing course materials and practitioner-focused books. We emphasised and discussed the following principles of unit testing: simplicity, readability, maintainability, self-documentation, avoidance of non-determinism and redundant assumptions, execution independence, focus on a single function, focus on external behaviour, ability to provide quick feedback, coverage of positive and negative behaviours, the four-phase test design (Setup-Execute-Verify-Teardown), and importance of test refactoring. The second lecture took place at the beginning of the second day where TDD was conceptually introduced (morning of Day-2). We slightly modified our own lecture material used in our software testing courses. In particular, we introduced the TDD way of working (e.g. the red-green-refactor cycle), pointed out the differences from incremental test-last development, and discussed the pros and cons of TDD in terms of the expected effects on programmer productivity and software quality. However, we took care to introduce TDD in an impartial manner, explaining that there are proponents and opponents of TDD. We did not present the existing empirical evidence in this lecture in order to avoid introducing biases.

- **Hands-on Group Exercises:** We conducted four hands-on group exercises following the randori session format. Two of the sessions took place during the first day with a unit testing and incremental test-last focus and the other two took place during the second day with a focus on TDD. We browsed through code-kata exercises available on the Web¹ and selected four tasks, after trying them out ourselves, for use in these sessions. In a randori session, a group of developers work on a single task with the

¹e.g. <http://craftsmanship.sv.cmu.edu>

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

goal of learning from each other, both from good and bad practices. The session is run by a “sensei” who leads the activity by asking questions, but not providing solutions. We, the researchers, acted as the sensei. The randori setting uses a single computer whose display is projected onto a big screen. A pair of developers leads the development activity, one pair at a time. One developer acts as the driver and the other as the co-pilot in a pair programming session, and they are asked to think and act aloud. The rest of the audience are seated facing the projected screen and are encouraged to make suggestions to the driver and co-pilot. However, it is up to the driver whether to follow the suggestions or not. The whole group is considered as one collective mind and silence is not allowed; the sensei starts asking questions when there is silence. The pairs rotate every five to ten minutes, i.e., the driver goes back to the group, the co-pilot becomes the driver and a new subject becomes the co-pilot. All participants ideally experience the driver role at least once during a session. We conformed to this format while encouraging the participants to follow incremental development. After each session, we conducted a short retrospective on the lessons learned, again led by us, to summarise the good practices and common mistakes that took place in the session.

- **Hands-on Individual Exercises:** Between the second and third days of the training, we gave participants two days break to take care of their work responsibilities. During these days, we suggested that they practise TDD on their work-related software development tasks. We think that this kind of personal practice may give subjects more hands-on experience with TDD than artificial exercises do. We also reserved these days for potential schedule slips, since we could use them as buffer zones as needed. We did not collect data from these practice days due to the company’s privacy policy.

Data collection. Data was collected on-site by the researchers. We collected the responses to the demographics questionnaire using an online form. After the questionnaire session, the answers were instantly available in a spreadsheet.

As planned, we extracted the software artefacts produced by each subject for each of the tasks at the end of the sessions. First, we helped the subjects copy the folders that contained the production code and test code to an external storage drive. Second, as a fall-back plan, the VM images the subjects used were exported to an external storage drive. The data collection took 30 to 60 minutes. At one site, we had to deviate from our data collection plan due to the time required to copy some subjects’ output. Instead, we first exported all the data to an internal server, and we then moved the data to the external drives.

5 Results

We analysed the data collected from all 24 subjects. The data analysis is performed as follows. First, descriptive statistics for all metrics are presented. This

1
2
3
4
5
6
7
8
9 is followed by the boxplots for all the metrics to visualize variations in the data.
10 Then we apply the two-stage analysis described in Section 3.10 to test the hy-
11 potheses related to the research questions. We report statistical significance,
12 observed power, and effect size for all tests. We also check for normality test
13 assumptions.

15 5.1 Quality (QLTY)

16 We present the descriptive statistics, box plots, and statistical test results for
17 QLTY in the next subsections.

20 5.1.1 Descriptive statistics for QLTY

21 Table 9 presents central values (mean along with its 95% confidence interval,
22 trimmed mean, and median), dispersion (variance, standard deviation, mini-
23 mum, maximum, range and interquartile range) and symmetry (skewness and
24 kurtosis) for the QLTY metric for both ITLD and TDD on the greenfield and
25 brownfield tasks respectively.

26 **TDD with the greenfield task has the highest mean in QLTY (76%)**
27 **with a confidence interval between 66.4% and 85.6%.** The mean for TDD
28 with the brownfield task has the lowest mean (39.1%) with a confidence interval
29 between 28.4% and 49.%. The mean for ITL is higher than TDD with the
30 brownfield task (53.9%) with a confidence interval between 35.4% and 72.5%.
31 The trimmed means are almost the same as the means in the three tasks (54.4%
32 for MR, 39.4% for MP and 78.5% for BSK). In terms of dispersion, ITL has the
33 greatest variance, followed by TDD with the brownfield task. TDD with the
34 greenfield task has the smallest variance.

35 In an interval of [0, 100], the range of variation for ITL and TDD with the
36 greenfield task is 100% (0% - 100%), and 72.9% for TDD with the brownfield
37 task (0% - 72.9%). It is important to note that a 100% QLTY value means all
38 the subtasks that the subject tackled have been correctly implemented. The
39 percentage varies among tasks depending on how correct each delivered subtask
40 is.

41 Normality tests for QLTY residuals ($p=0.002$) and z -scores for skewness
42 and kurtosis values ($z_{skewness} = -1.677$ and $z_{kurtosis}=-1.021$) show that the
43 residuals do not depart much from normality (Kim, 2013).

44 The box plot in Figure 2 shows an outlier in the TDD treatment with the
45 greenfield task. If this outlier is removed, the range of variation reduces to 52%
46 - 100%.

50 5.1.2 Hypothesis testing for QLTY

51 We performed the repeated-measures GLM on the QLTY metric. Mauchly's
52 sphericity test showed that we cannot reject the null hypothesis for sphericity
53 assumption ($p = 0.23$); therefore we could use the univariate test. The test
54 confirms that there is a significant difference between the mean QLTY obtained
55
56
57
58

	ITLD	TDD-greenfield	TDD-brownfield
Mean	53.9 (9.0)	76.0 (4.6)	39.1 (5.2)
Lower Bound*	35.4	66.4	28.4
Upper Bound*	72.5	85.6	49.8
5% Trimmed Mean	54.4	78.5	39.4
Median	66.4	80.6	42.9
Variance	1843.9	496.3	610.3
Std. Deviation	42.9	22.3	24.7
Minimum	0.0	0.0	0.0
Maximum	100.0	100.0	72.9
Range	100.0	100.0	72.9
Interquartile Range	100.0	34.7	40.9
Skewness	-0.3	-1.9	-0.5
Kurtosis	-1.7	5.1	-0.9

*Bounds are given for 95% confidence interval of the mean.

Table 9: Descriptive statistics for QLTY

from the two treatments with the tasks of different complexity, i.e., among BSK, MR, MP ($F(2; 44) = 7.887$, $p = 0.001$). The observed power of the univariate test (0.94) shows that the analysis has sufficient power. Figure 3 depicts the profile plot for QLTY.

The GLM test indicates a significant difference among the three tasks of different complexity, but this does not necessarily indicate that the difference is between ITLD and TDD treatments. So we need to perform pairwise comparisons between ITLD and TDD. Pairwise comparison between ITLD and TDD with the greenfield task using Bonferroni adjustment suggests that the observed differences in the marginal means of the two treatments (see Table 10) are not significant ($p = 0.36$). Thus we conclude that **the development approach does not have a significant impact on the quality of the work produced**. Pairwise comparison between the two TDD tasks reveals that TDD with the greenfield task is significantly different from TDD with the brownfield task in terms of QLTY.

	Mean	Std. Error
ITLD	53.9	8.95
TDD-greenfield	76.0	4.64
TDD-brownfield	39.1	5.15

Table 10: The estimated marginal means for QLTY

5.1.3 Effect size

The partial eta-squared measure of GLM shows that the development approach explains around 26.4% of the total variability in the model for QLTY. The

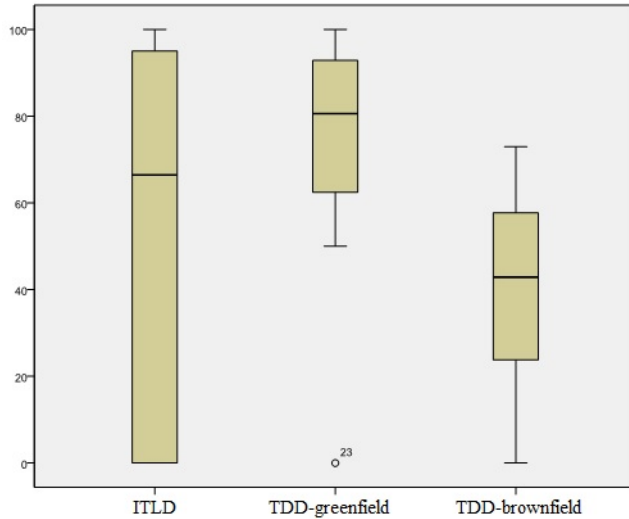


Figure 2: Box plot representing the QLTY metric for ITLD and TDD with the greenfield and brownfield tasks

remaining 74% could be attributable to other causes. This ratio is acceptable in studies dealing with human behaviour.

Kampenes et al. (2007) studied standardized effect sizes in software engineering experiments and suggested that an absolute effect size of 0.17 indicates a small effect, 0.6 indicates a medium effect, and 1.4 indicates a large effect. These values are slightly higher than those suggested in psychological and behavioural sciences (Kampenes et al., 2007). Cohen's d for QLTY between the observations of ITLD and TDD with the greenfield task is -0.65. So, the effect of the difference between the means of those is medium. Cohen's d for QLTY between the observations of TDD with the greenfield and brownfield tasks is 1.57. So, the effect of the difference between the two tasks is large. Cohen's d for QLTY between the observations of ITLD and TDD with the brownfield task is 0.42, which indicates a small to medium effect.

5.2 Productivity (PROD)

5.2.1 Descriptive statistics for PROD

Table 11 presents the central values, dispersion and symmetry for ITLD and TDD treatments respectively. **The mean productivity is highest in the TDD approach with the greenfield task, (47.6%) with a confidence interval between 32% and 63.1%.** The lowest mean productivity is found in TDD with the brownfield task, (15.9%) with a confidence interval between 11.1% and 20.7%. The trimmed mean shows almost the same values as the means in all three tasks.

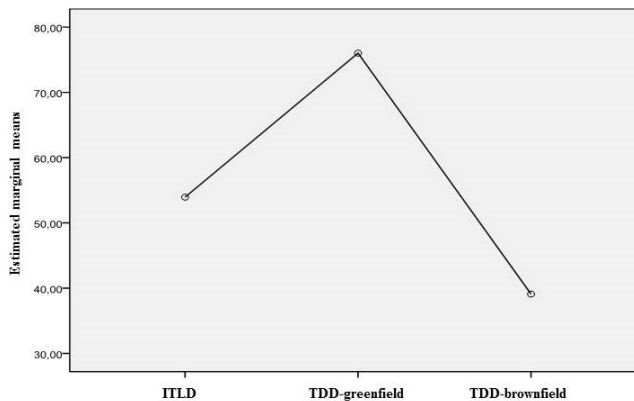


Figure 3: Profile plot for QLTY

The median productivity is in accordance with the mean productivity for the TDD treatments (50% in BSK and 16.7% in MP), but not for the ITLD treatment, where there is a difference of around 10%. This might be due to the fact that a few subjects produced very low productivity values in ITLD.

In an interval of $[0,100]$, the range of variation in ITLD is 87.6% (0% - 87.6%), and 96.6% in TDD with the greenfield task (0% - 96.6%), but it is around the half of this latter range in TDD with the brownfield task (0% - 40.9%). A level of 100% productivity indicates that all subtasks have been successfully delivered, i.e., all assertions associated with all subtasks are passing. Thus, perfect productivity implies perfect quality in our case, but not vice versa.

Normality tests for PROD residuals ($p=0.166$) and z -scores for skewness and kurtosis values ($z_{skewness}=0.831$ and $z_{kurtosis}=-1.03$) show that the residuals may follow a normal distribution.

The box plot in Figure 4 shows no outliers.

5.2.2 Hypothesis testing for PROD

We reject the null hypothesis for Mauchly's sphericity test ($p = 0.03$) for the PROD metric. Hence, we performed both the multivariate test and univariate test with Huynh-Feldt correction. Both tests confirm that there is a significant difference between the mean PROD obtained from the two treatments with tasks of different complexity. Univariate test statistics are ($F(1.66; 36.61) = 12.584$, $p < 0.001$). The observed power (0.99) shows that the analysis has sufficient power. Figure 5 depicts the profile plot for PROD.

Further pairwise comparisons between the marginal means reported in Table 12 indicate that **TDD with the greenfield task is significantly different from both ITLD** ($p = 0.012$) and TDD with the brownfield task ($p = 0.001$). Since there is a significant decrease in the performance of TDD with the brownfield task (from 47.6 to 15.9), it is more likely that the task complexity is con-

	ITLD	TDD-greenfield	TDD-brownfield
Mean	23.5 (5.2)	47.6 (7.5)	15.9 (2.3)
Lower Bound*	12.7	32.1	11.1
Upper Bound*	34.3	63.1	20.7
5% Trimmed Mean	21.5	47.5	15.5
Median	13.5	50.0	16.7
Variance	619.6	1291.5	122.8
Std. Deviation	24.9	35.9	11.1
Minimum	0.0	0.0	0.0
Maximum	87.6	96.6	40.9
Range	87.6	96.6	40.9
Interquartile Range	48.3	77.6	15.9
Skewness	0.8	-0.1	0.1
Kurtosis	0.0	-1.8	-0.4

*Bounds are given for 95% confidence interval of the mean.

Table 11: Descriptive statistics for PROD

founded by the development approach. To check this, we ran the second stage analysis for PROD.

The MIXED model shows that the development approach (TDD versus ITLD) does not significantly affect PROD ($F(1; 23.07) = 3.116, p = 0.09$), whereas the task complexity (greenfield versus brownfield) has a significant effect on PROD ($F(1; 23.06) = 18.771, p < 0.001$). A detailed summary of the marginal model results is presented in Table 13.

Table 14 also shows the mean PROD of development approaches based on the modified population marginal mean. Note that there is a slight improvement (8%) in PROD when subjects apply TDD instead of ITLD due to the aggregated means of the two TDD tasks.

We conclude that **contextual factors -task complexity in particular- significantly matter for PROD.**

	Mean	Std. Error
ITLD	23.5	5.19
TDD-greenfield	47.6	7.49
TDD-brownfield	15.9	2.31

Table 12: The estimated marginal means for PROD

5.2.3 Effect size

The partial-eta squared measure suggests that the development approach explains 36.4% of the total variability in the model for PROD. The remaining 63.6% could be attributable to other causes.

Cohen's d for PROD between the observations of ITLD and TDD with the greenfield task is -0.78. This indicates that the difference between the two

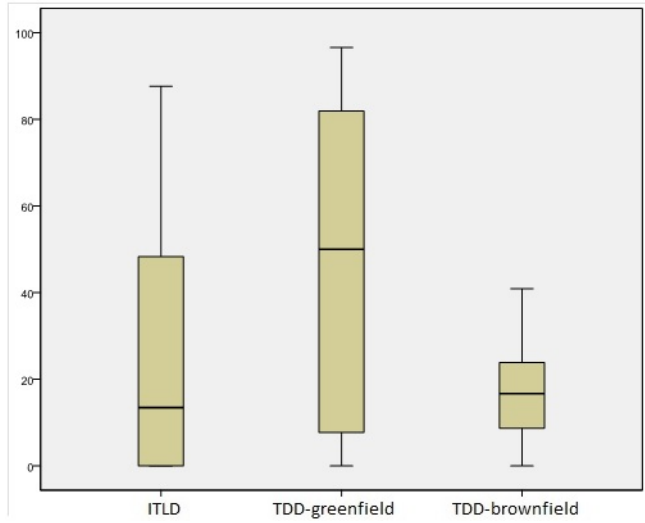


Figure 4: Box plot representing the PROD metric for ITLD and TDD tasks

Source	Numerator df	Denominator df	F	Significance
Intercept	1	22.99	50.92	< 0.001
Development Approach	1	23.07	3.11	0.91
Task	1	23.06	18.77	< 0.001

Table 13: The marginal model results for PROD

treatments is large. Cohen’s d for PROD between the samples of TDD with the greenfield and brownfield tasks is 1.19, which indicates a substantially large effect.

6 Interpretations

On the differences between treatments. In summary, we found that TDD is not statistically different than ITLD in terms of the quality of the work done when both development approaches are applied to a simple task. The quality observed in the second TDD task, the brownfield task, is significantly the lowest value (43% on average) of all treatments.

In terms of productivity, task complexity significantly affects the findings. Applying TDD to a greenfield task yielded the best performance (50%), whereas applying TDD on a brownfield task yielded the worst (16%). One possible explanation for such a difference between the two TDD practices could be the extra steps involved in applying TDD to legacy code. When subjects apply TDD to a greenfield task, the steps of TDD (red-green-refactor) can easily

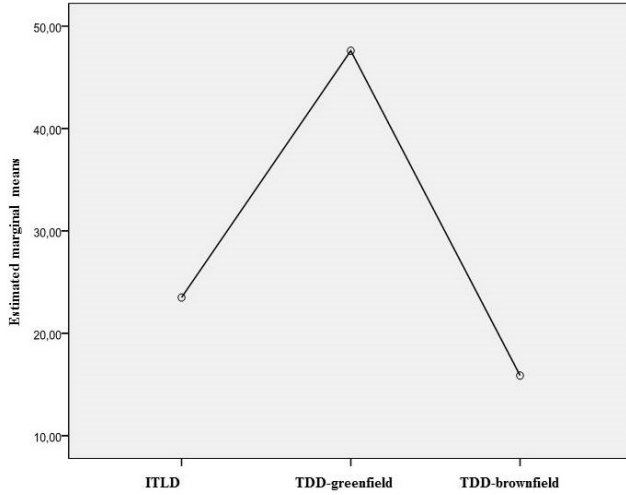


Figure 5: Profile plot for PROD

Development approach	Mean	Std. Error	df	Lower bound*	Upper bound*
ITLD	22.52	5.06	23	12.04	32.99
TDD	31.03 ^b	4.06	23.02	22.63	39.43

*Bounds are given for 95% confidence interval of the mean.

b.Based on the modified population marginal mean.

Table 14: Modified marginal means of the development approaches for PROD

be applied. Applying TDD to a brownfield task requires understanding the legacy code, identifying which parts of the code would be affected by a new piece of functionality, and testing both the new and old functionality at the end of each cycle to confirm everything works properly. The result is a more complex, involved process that requires a lot more attention than is required when implementing a system from scratch.

Notice that there is a link between our productivity and the quality of the work done per task. We consider some subjects to be more productive than others if they deliver a task of higher quality, i.e., a higher number of passed assertions over all assertions per task. Therefore, our productivity measure favours TDD only for a delivery of tasks of higher quality. Based on this fact, we could say that subjects tackle a similar amount of subtasks in both treatments (quality), but they deliver higher-quality work when they apply TDD to a simple task than ITLD (productivity).

On the task complexity. We chose for the second experimental task a brownfield task with legacy code that has more operations to be understood. The reason for this decision was to move TDD outside its normal sweet spot. Based

1
2
3
4
5
6
7
8
9 on the demographics survey, test-last development was well known and used by
10 the subjects, even if they did not master it in the same incremental way that
11 the ITLD approach dictated. So during the control treatment, there was less
12 risk of task complexity affecting the performance of the subjects.

13 When we compare the results of a simple task developed with TDD (e.g.
14 BSK) with those of a simple task developed with ITLD (e.g. MR), a task selec-
15 tion bias is possible. In fact, the decrease in terms of quality and productivity
16 in the second, brownfield TDD task suggests that this bias was present. The
17 MIXED model confirmed this: The goal of the second stage analysis was to see
18 whether TDD with a complex task would still be significantly different than
19 ITLD. But the results did not confirm this hypothesis. Increased task complex-
20 ity obscured the effect of the development approach.

21 We knew that there were differences in the specifications of the greenfield
22 tasks, BSK and MR. The BSK specification appeared more concrete on the
23 surface, and included more examples than the specification of MR did. We
24 wondered if the quality and productivity improvements in the first TDD prac-
25 tice could be due to the perceived difficulty or comprehensiveness of the task
26 specifications themselves and whether the subjects also observed these differ-
27 ences. So we asked the subjects in a post-test survey whether they perceived
28 BSK as easier or more comprehensive than MR. The responses suggest that
29 they perceived the two greenfield tasks to be of similar difficulty and compre-
30 hensiveness. So, when task complexity is constant, TDD may indeed perform
31 better. However, when they were asked about the second TDD task, the brown-
32 field task, the subjects invariably indicated that they found it be much more
33 complex than the other tasks. Therefore, the fact that their performance was
34 lower with this task should not be too surprising.

35 36 37 38 **7 More insights into the experiment artefacts**

39
40 We looked more deeply into the artefacts produced by the subjects during the
41 experimentation in order to get a better understanding on the subjects' perfor-
42 mance, and the extent to which the tasks were implemented. We extracted the
43 number of unit tests written per task, the percentage of tackled subtasks, and
44 its relation with QLTY metric. We analysed the subtasks that were mostly im-
45 plemented correctly, the number of subtasks that were implemented incorrectly
46 and that could not be implemented at all by the subjects. We also checked if
47 there is a difference among the subjects' performance in terms of QLTY and
48 PROD among the three experiment sites.

49 **On the subjects' performance.** The subjects wrote several unit test cases
50 to implement the subtasks of the three tasks, namely MR, BSK and MP. Some
51 subtasks may be tested with a single test case in which there exists a single as-
52 sert statement (e.g. testing whether the MarsRover stays in the initial position
53 (0,0,N) after executing an empty (" ") command), whereas other subtasks may
54 be tested with more than one test case with several assertions. The subjects
55 designed their source code, i.e., classes and methods, and its associated test
56
57
58

1
2
3
4
5
6
7
8
9 cases differently from each other: Some subjects preferred to implement multi-
10 ple assertions inside a test method, whilst the others followed the approach of
11 implementing a single assert statement in a test method. Furthermore, many
12 subjects wrote additional test cases that check methods that are not related to
13 a single functionality (e.g. testing whether a setter function worked properly).
14 Therefore, it was not possible to define a one-to-one relation between the unit
15 test cases written by the subjects and the tackled subtasks.

16 The subjects wrote a median of 8, 12 and 7 unit tests during the imple-
17 mentation of MR, BSK and MP tasks, respectively. It seems that TDD led
18 the subjects practice more on unit testing. However, the number of unit tests
19 written by the subjects does not directly relate to the number of tackled sub-
20 tasks. For example, one subject wrote 22 unit tests corresponding to a total of 6
21 tackled subtasks of MR, while another subject wrote 5 unit tests corresponding
22 to a total of 5 tackled subtasks of MR.

23 The subjects tackled a median of 4, 8 and 5 subtasks of MR, BSK and
24 MP tasks, respectively. Over the total number of subtasks (11, 13 and 11
25 subtasks in MR, BSK and MP, respectively), 32%, 62% and 45% of subtasks
26 were tackled during the implementation of MR, BSK and MP tasks. It seems
27 that the subjects were able to tackle more subtasks during the implementation
28 of TDD tasks. Note that a subtask is considered as tackled if at least one assert
29 statement in the acceptance test suite associated with that subtask passes. The
30 percentage of tackled subtasks is different than our measure, QLTY, which is the
31 ratio of passed assert statements over all assert statements in the acceptance test
32 suite for a particular subtask. Based on the experiment findings, although the
33 subjects tackled more subtasks during TDD on a simple task, they performed
34 similarly in terms of QLTY.

35 **On the tackled subtasks.** Unfortunately, some subtasks could not be tackled
36 by most of the subjects during the experimentation, e.g. only one subject tackled
37 the last five subtasks on obstacle detection in MR. This could be due to the
38 sequential order in which the subjects tackled subtasks, not the complexity of
39 the later subtasks. In general, MR task might have taken more time than it
40 was allocated, or the subjects might have spent more time on the unit testing
41 and incremental development approach than completing the subtasks. The rest
42 of the subtasks in MR were tackled by 10 to 15 subjects. During TDD on
43 a simple task (BSK), subjects seem to perform better: 22 out of 24 subjects
44 tackled the first and third subtasks, whilst the other subtasks were tackled
45 by at least 10 subjects. The only subtask that could not be tackled by the
46 subjects is the last one in BSK, in which the final score of the game should be
47 calculated. The statistics from MP task reside in the middle of MR and BSK:
48 At least 15 subjects tackled the first three subtasks (calculating the distance
49 in miles) and the sixth and seventh subtasks (getting the user destination in
50 erroneous conditions, and recommending the artist). The subjects seemed to
51 have difficulties with the subtasks that are related to error handling for invalid
52 coordinates and artists with no concerts (forth, tenth and eleventh subtasks) and
53 that have more detailed post-conditions such as retrieving the top 20 artists in
54 the order of the number of fans (eighth and ninth subtasks). Those subtasks

1
2
3
4
5
6
7
8
9 were tackled by only five subjects.

10 Regarding the quality, overall findings show no difference between the quality
11 of tackled subtasks during ITLD and TDD on a simple task. However, when
12 we focus on the number of subtasks successfully implemented by the subjects,
13 we observed that the subjects successfully implemented more subtasks during
14 ITLD compared to TDD on a simple task. During ITLD, the subjects achieved
15 100% quality for the most commonly tackled subtasks, more specifically the first
16 six subtasks of MR task. The quality of the last five subtasks in MR is between
17 40% and 100%. Note that these last five subtasks were tackled by a single
18 subject. During TDD on a simple task, a median of 100% quality was achieved
19 for the first four subtasks of BSK. As the number of subjects who tackled the
20 later subtasks in BSK are quite few, the quality achieved in those subtasks also
21 decreases (a median of 20% for the fifth, sixth and seventh subtasks). Still, we
22 observed that some subjects managed to correctly implement all the subtasks
23 that they tackled during TDD on a simple task (100% quality). The worst
24 quality values in BSK belong to the last two subtasks. Finally, during TDD on
25 a complex task (MP), the subjects achieved a quality between 20% to 100% for
26 the first three, sixth and seventh subtasks. For the last three subtasks in MP,
27 unfortunately, none of the subjects managed to achieve a quality higher than
28 20%. These statistics at subtask level indicate that the effect of TDD on quality
29 and productivity could be understood much better at a finer granularity in the
30 future experiments.

31
32 **On the experiment sites.** We conducted our experiment at three different
33 FSecure sites: Oulu (Finland), Helsinki (Finland) and Kuala Lumpur (Malaysia).
34 While executing the experiments at these sites, we did not perform anything dif-
35 ferently regarding the order of treatments, the content of training, the team who
36 attended the training from the researchers' side, the tasks, the time allocated to
37 each task and the amount of information provided to the subjects regarding the
38 tasks. We also did not intervene with the sample selection process in all three
39 sites; all volunteered subjects were welcome. Nevertheless we suspected that the
40 site in which the experiment was conducted might have affected the findings.
41 We additionally built two repeated-measures GLMs (one for QLTY and the other
42 for PROD) in which the experiment site was included as a between-subjects
43 factor.
44

45 Mauchly's sphericity test showed that we cannot reject the null hypothesis for
46 sphericity assumption ($p = 0.26$ for QLTY and $p = 0.09$ for PROD); therefore we
47 could use the univariate test. The univariate tests on QLTY and PROD metrics
48 confirm that there is no significant difference between the metric values obtained
49 from the three sites ($F(2; 20) = 0.014$, $p = 0.986$ for QLTY, $F(2; 20) = 0.796$,
50 $p = 0.465$ for PROD). Hence we conclude that the experiment site did not
51 have a significant effect on the quality of the work done and the productivity of
52 developers.
53
54
55
56
57
58

8 Comparison with the earlier studies

Our findings on external quality complement prior observations summarized by Turhan et al. (2010), by Rafique and Misic (2013) and aggregated by Munir et al. (2014) regarding high-rigour and high-relevance experiments: TDD does not appear to improve external quality, since the difference between TDD on a greenfield task and ITLD is not significant. Furthermore, we cannot generalize this result to tasks of varying complexity. Note that the results of earlier studies could also be dependent on the choice of tasks: Earlier controlled experiments used the same task, which could also be considered a simple greenfield task, in both treatments.

Regarding the industry experiments listed in Table 1, our results in terms of external quality with TDD with the greenfield (BSK) task contrast the findings by George and Williams (2004), who used an adapted version of BSK in the TDD group. George and Williams (2004) reports that TDD significantly improves external quality, whereas we found the opposite on BSK in TDD and MR in ITLD treatments. The other two industry experiments reported in Table 1 did not study the effects of TDD on external quality.

Our results regarding QLTY are in line with the findings of a controlled experiment reported by Erdogmus et al. (2005) with subjects, and a replication of Erdogmus et al. (2005) as a randomized trial with students reported in Fucci and Turhan (2013). The authors in (Fucci and Turhan, 2013) used a between-subject, non-crossover design and compared ITLD and TDD using exactly the same BSK task used in this study with the same amount of allocated time. Both studies ((Erdogmus et al., 2005), (Fucci and Turhan, 2013)) reported no significant differences between the two groups in terms of quality, measured similarly to our experiment. It appears that professional developers might be better and faster at internalizing the TDD process than students, which can explain the more dramatic improvements in quality with simple tasks (89% in both ITLD and TDD reported in Fucci and Turhan (2013), whereas in our study, QLTY is 66% in ITLD versus 80% in TDD). This claim is supported by the evidence collected by Latorre (2014a) when comparing the effects of TDD on professionals and students who have been freshly introduced to the technique.

The comparison of productivity results with previous studies are more interesting. In the literature reviews ((Rafique and Misic, 2013), (Turhan et al., 2010), (Munir et al., 2014)), findings on productivity were inconsistent. This might be due to the differences in productivity measures across different studies. In the three industry experiments, the time required to complete a task was used to quantify productivity. George and Williams (2004) and Canfora et al. (2006) conclude that TDD requires more time, i.e., subjects spend more time applying TDD compared to a test-last approach. However, our metric for productivity does not consider completion time; instead it is based on the amount of work done in a fixed time in terms of percentage of passing assertions. Our findings suggest that subjects are more productive when they implement TDD on a simple task compared to ITLD, but the productivity drops significantly when applying TDD to a complex brownfield task. So, task selection matters

1
2
3
4
5
6
7
8
9 significantly in the interpretation of results.

10 The decrease of productivity from one TDD application to another is a
11 result of the increased complexity of the task. Pancur and Ciglaric (2011) also
12 conjecture that inconsistent findings in the literature could be due to the task-
13 related factors, such as their specifications and their granularity used in the
14 treatments. If we choose a coarser-grained task for the control group vs. the
15 TDD group, the observed benefits of TDD could have been caused by shorter
16 development cycles with finer-grained tasks (Pancur and Ciglaric, 2011).

17 Geras et al. (2004) observe that there is less variability between the estimated
18 and actual effort in a test-first approach, and in turn, the development effort is
19 more predictable in this approach compared to a test-last approach. According
20 to the box plots of PROD, we could argue the opposite, i.e., there is more
21 variability in productivity when TDD is applied. Therefore we could not confirm
22 the observations of Geras et al. (2004) in our context.

23 Fucci and Turhan (2013) report inconclusive results regarding productivity,
24 using a similar measure to ours. Therefore, the effect of TDD on productivity,
25 at least in the short term and in the context of a greenfield development task,
26 is unfortunately still unclear, and deserves further investigation.
27
28

29 9 Threats to Validity

30 We follow the classification by Sjoeborg et al. (2005) and Wohlin et al. (2012)
31 while reporting the threats to the validity of our findings. As suggested in the
32 guidelines by Jedlitschka and Pfahl (2005), we discussed potential design threats
33 separately in Section 3.11.
34
35
36

37 9.1 Conclusion Validity

38 We checked the sphericity assumptions of GLM and applied the statistical tests
39 (univariate or multivariate) whose conditions were required to be met in hy-
40 pothesis testing. The significance levels in both tests were selected as 0.05, but
41 we observe that QLTY significance tests are rejected with $p = 0.001$ and PROD
42 tests are rejected with $p = 0.03$. The observed power of both significance tests
43 was high, indicating our analysis had sufficient power with the existing sample.
44 The effect sizes in both metrics varied from medium to high.
45
46

47 9.2 Internal Validity

48 We have already discussed potential threats to internal validity such as fatigue,
49 carry-over, order, and practice effects in Section 3.11. Additionally, our experi-
50 ment could be subject to attrition threat (loss of participation), selection threat
51 (subject selection) and instrumentation threat (task selection). Two subjects
52 dropped out after the first day (before any treatments were applied). The rest
53 of the subjects stayed for the remainder of the study and during the adminis-
54 tration of all treatments. We think the reason for the two subjects leaving the
55
56
57
58

1
2
3
4
5
6
7
8
9 study might have been due to their workload or due to the training not meet-
10 ing their expectations. We conclude that our experiment did not suffer from a
11 signification attrition threat.

12 The selection threat might occur due to the sampling approach used to select
13 the experiment subjects. Due to the environment in which the experiments were
14 conducted, we did not have the opportunity to randomly select the subjects from
15 a given population, nor internally randomize the treatment groups. We had to
16 rely on a convenience sample instead and a repeated-measures design in which
17 all subjects performed all treatments. Thus, our conclusions are subject to the
18 selection threat.

19 The instrumentation threat might occur due to the differences in specificity
20 and format of BSK and MR specifications. BSK specification has more fine-
21 grained user stories provided to the subjects, whereas MR is written in terms
22 of the operations that the rover must perform. We asked the subjects in a post-
23 treatment survey whether they perceived BSK as easier or more comprehensive
24 than MR. The responses show that they perceived the two greenfield tasks to be
25 of similar difficulty and comprehensiveness. Nevertheless we acknowledge that
26 during BSK, the subjects did not need to spend time for deciding on the user
27 stories, compared to MR in which the subtasks are slightly hidden inside the
28 rover descriptions. We plan to investigate the effect of the task on the findings
29 in detail in our future experiments.

30 Other internal validity issues such as history (applying treatments at dif-
31 ferent times), mortality (leaving the treatments before completion) were not
32 applicable in the context of this experiment.
33
34

35 **9.3 External Validity**

36 External validity deals with the generalizability of results in terms of objects
37 (tasks), subjects, and technologies used. The tasks we chose for all three treat-
38 ments were small in terms of LOC, and their complexities were also lower than
39 typical real-life industrial applications. Our results cannot be generalized to
40 large software systems with different domain characteristics. We addressed re-
41 alism to a certain extent by including a brownfield task as an object. As far as
42 we know, ours is the first experiment to use such a task.

43 Most of the experiments reported in the literature use Java as the program-
44 ming language, Eclipse as the development environment, and JUnit as the unit
45 testing tool (e.g., (Erdogmus et al., 2005; George and Williams, 2004; Fucci
46 and Turhan, 2013)). We chose the same technological setup, and most of the
47 subjects were comfortable with it. Technically, there is a risk that our find-
48 ings are dependent on the technologies used in this study, but this risk is fairly
49 small, since the development approaches applied are independent of the IDE,
50 programming language, and unit testing tool used. Most languages and IDEs
51 have functionality and tools analogous to those used in this experiment.

52 Regarding the subjects, our sample consisted entirely of professionals, who
53 were novice to senior developers. All had experience in some agile software de-
54 velopment practices. Around half of them indicated that they had prior knowl-
55
56
57
58

1
2
3
4
5
6
7
8
9 edge of TDD. We talked with these subjects and learned that those subjects
10 individually attended training/workshops on TDD, but they never applied this
11 practice before participating to our study. We believe our results can be gener-
12 alized to professionals who do not have prior hands-on experience of TDD, but
13 have knowledge and experience with other agile practices.
14

15 9.4 Construct Validity

16
17 We used Java as the programming language during both training and treat-
18 ments, and used the same type of instrument (acceptance test suites) to measure
19 dependent variables for all three tasks. Hence, we addressed possible threats
20 regarding the use of different programming languages or tools to measure de-
21 pendent variables.
22

23 Our experiment did not suffer from mono-operation bias for the experimental
24 group (TDD) since we used two objects with different complexity levels.

25 We defined a single metric to quantify each dependent variable (QLTY and
26 PROD) in this experiment. Hence, our experiment may be subject to mono-
27 method bias. We acknowledge that QLTY and PROD can be measured in
28 several different ways: We used the same external quality metrics used in pre-
29 vious studies (Erdogmus et al., 2005; George and Williams, 2004; Fucci and
30 Turhan, 2013)), and explained our rationale for choosing a different productiv-
31 ity metric in Section 3.2 (it was a variation of the metrics used in (Erdogmus
32 et al., 2005; Fucci and Turhan, 2013)). We measured both of these metrics
33 objectively and reliably using automated techniques. They were relative and
34 fine-grained to make comparison meaningful. These characteristics allowed us
35 to decouple the tasks from the metrics used. The metrics could not be gamed
36 by the subjects since the subjects were not aware of we would evaluate their
37 performance. We believe the metrics captured the underlying constructs fairly
38 accurately and reliably.
39
40

41 10 Conclusion and Future work

42
43 We conclude that the task selection significantly affects the results of TDD
44 experiments and obscures the effects of TDD compared to ITLD. We argue
45 that the current findings reported in the literature on TDD should also be
46 assessed based on similar experimental factors, like the similarity of tasks used in
47 previous experiments, task type (a toy example or a more complex application),
48 and granularity of task specifications (how well-sliced each user story is).
49

50 In the context of our ESEIL projects, we are performing new experiments
51 with industry partners to explain TDD phenomenon in detail. We are searching
52 for other factors that may affect the TDD process, such as conformance to the
53 TDD process (Fucci et al., 2014, 2015), experience level of the subjects (Salman
54 et al., 2015), slicing of task specifications, and use of real applications specific
55 to industry.
56
57
58

1
2
3
4
5
6
7
8
9 From an experimental point of view, we are working on changing our design
10 by random allocation of tasks to subjects, and by adding new dimensions such as
11 unit test quality and internal quality. We are also conducting post-experiment
12 surveys in which we measure subjects' understanding of 'completed subtasks',
13 and compare these with the actual tackled subtasks. Observing such differences
14 between the subjects' perceptions of completeness and the measured quality
15 and productivity would shed light to deeper insights on the TDD experiments.
16 Finally, more detailed analysis on the test cases, and the effort spent on writing
17 unit tests and source codes could be performed in future studies. In this exper-
18 iment, we did not calculate the time spent during coding and testing in detail,
19 as we did not enforce the subjects use a version control system which would
20 keep timestamps for each action (e.g. commits of a source code, commits of test
21 cases). Thus, it would be more interesting to accurately observe such differences
22 regarding coding, testing and refactoring efforts through an automated system
23 in a future study.
24
25

26 Acknowledgment

27
28 This research has been partly funded by Spanish Ministry of Science and Inno-
29 vation projects TIN2011-23216, the Distinguished Professor Program of Tekes,
30 and the Academy of Finland (Grant Decision No. 260871). We would like to
31 thank Dr. Lucas Layman for his contributions in designing one of the tasks used
32 in this experiment. We would also like to sincerely thank FSecure Corporation
33 and the software professionals who attended our training/experiment.
34
35

36 References

37 References

- 38
39
40
41 (2014). Eclipse helios. <https://www.eclipse.org/helios/>.
42
43 (2014). Oracle virtual box 4.3.
44
45 (2015). The bowling game kata.
46
47 Aniche, M. F. and Gerosa, M. A. (2010). Most common mistakes in test-driven
48 development practice: Results from an online survey with developers. In *Third*
49 *International Conference on Software Testing, Verification and Validation*
50 *Workshop*.
51
52 Basili, V. (1992). Software modeling and measurement: The
53 goal/question/metric paradigm. Technical Report CS-TR-2956, UMIACS-
54 TR-92-96, University of Maryland.
55
56 Beck, K. (2003). *Test Driven Development: By Example*. Addison Wesley.
57
58
59
60
61
62
63
64
65

- 1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
- Becker, K., Pimenta, M. S., and Jacobi, R. P. (2014). Besouro: a framework for exploring compliance rules in automatic tdd behavior assessment. *Information and Software Technology*.
- Bergersen, G. R., Sjøberg, D. I. K., and Dybå, T. (2014). Construction and validation of an instrument for measuring programming skill. *IEEE Trans. Software Eng.*, 40(12):1163–1184.
- Canfora, G., Cimitile, A., Garcia, F., Piattini, M., and Visaggio, C. A. (2006). Evaluating advantages of test driven development: a controlled experiment with professionals. In *ISESE*, pages 364–371.
- Causevic, A., Sundmark, D., and Punnekkat, S. (2010). An industrial survey on contemporary aspects of software testing. In *Third IEEE International Conference on Software Testing, Verification and Validation*.
- Causevic, A., Sundmark, D., and Punnekkat, S. (2011). Factors limiting industrial adoption of test driven development: A systematic review. In *Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 337–346.
- C.E. McCulloch, C. and Searle, S. (2000). *Generalized, Linear, and Mixed Models*. John Wiley and Sons.
- Coe, R. (2002). It’s the effect size, stupid: What effect size is and why it is important. In *Annual Conference of the British Educational Research Association*.
- Cohen, J. (1992). A power primer. *Psychological Bulletin*, 112(1):155–159.
- Draper, D. (2006). Dojo, kata or randori?
- Ellis, P. D. (2010). *The Essential Guide to Effect Sizes: Power, meta-analysis and the interpretation of research results*. Cambrigde.
- Emam, K. (2003). Finding success in small software projects, agile project management executive report. Technical report, Cutter Consortium, Arlington, Massachusetts.
- Erdogmus, H., Morisio, M., and Torchiano, M. (2005). On the effectiveness of the test-first approach to programming. *IEEE Trans. Softw. Eng.*, 31:226–237.
- Field, A. (2007). *Discovering Statistics using SPSS*. Sage Publications Inc.
- Fucci, D. and Turhan, B. (2013). A replicated experiment on the effectiveness of test-first development. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 103–112.

- 1
2
3
4
5
6
7
8
9 Fucci, D., Turhan, B., Juristo, N., Dieste, O., Tosun-Misirli, A., and Oivo, M.
10 (2015). Towards an operationalization of test-driven development skills: An
11 industrial empirical study. *Information and Software Technology*, 68:82 – 97.
12
13 Fucci, D., Turhan, B., and Oivo, M. (2014). Impact of process conformance on
14 the effects of test-driven development. In *Proceedings of the 8th ACM/IEEE*
15 *International Symposium on Empirical Software Engineering and Measure-*
16 *ment*, page 10. ACM.
17
18 Gamma, E. and Beck, K. (2014). Junit testing framework.
19 <https://github.com/junit-team/junit/wiki/Download-and-Install>.
20
21 George, B. (2002). Analysis and quantification of test driven development ap-
22 proach. Master’s thesis, NC State University.
23
24 George, B. and Williams, L. (2003). An initial investigation of test driven
25 development in industry. In *ACM Symposium on Applied Computing*.
26
27 George, B. and Williams, L. (2004). A structured experiment of test-driven
28 development. *Information and Software Technology*, 46(5):337 – 342. Special
29 Issue on Software Engineering, Applications, Practices and Tools from the
30 {ACM} Symposium on Applied Computing 2003.
31
32 Geras, A., Smith, M., and Miller, J. (2004). A prototype empirical evaluation
33 of test driven development. In *10th International Symposium on Software*
34 *Metrics (METRICS)*.
35
36 Ivarsson, M. and Gorschek, T. (2011). A method for evaluating rigor and in-
37 dustrial relevance of technology evaluations. *Empirical Software Engineering*,
38 16:365–395.
39
40 Jedlitschka, A. and Pfahl, D. (2005). Reporting guidelines for controlled ex-
41 periments in software engineering. In *International Symposium on Empirical*
42 *Software Engineering*.
43
44 Juristo, N. (2016). Experiences conducting experiments in industry: The es-
45 eil fidipro project. In *4th International Workshop on Conducting Empirical*
46 *Studies in Industry*. ACM.
47
48 Kampenes, V. B., Dyba, T., Hannay, J. E., and Sjoberg, D. I. (2007). A sys-
49 tematic review of effect size in software engineering experiments. *Information*
50 *and Software Technology*, 49(11-12):1073 – 1086.
51
52 Kim, H.-Y. Y. (2013). Statistical notes for clinical researchers: assessing nor-
53 mal distribution (2) using skewness and kurtosis. *Restorative dentistry &*
54 *endodontics*, 38(1):52–54.
55
56 Kollanus, S. (2010). Test driven development - still a promising approach? In *7th*
57 *International Conference on the Quality of Information and Communications*
58 *Technology*, pages 403–408.

- 1
2
3
4
5
6
7
8
9 Latorre, R. (2014a). Effects of developer experience on learning and applying
10 unit test-driven development. *Software Engineering, IEEE Transactions on*,
11 40(4):381–395.
12
13 Latorre, R. (2014b). A successful application of a test-driven development strat-
14 egy in the industrial environment. *Empirical Software Engineering*, 19:753–
15 773.
16
17 Madeyski, L. and Szala, L. (2007). *Lecture Notes in Computer Science*, chapter
18 The Impact of Test-Driven Development on Software Development Productiv-
19 ity An Empirical Study, pages 200–211. Springer.
20
21 Maximilien, E. M. and Williams, L. (2003). Assessing test-driven development
22 at ibm. In *International Conference on Software Engineering (ICSE)*.
23
24 Munir, H., Moayyed, M., and Petersen, K. (2014). Considering rigor and rele-
25 vance when evaluating test driven development: A systematic review. *Informa-
26 tion and Software Technology*, 56:375–394.
27
28 Nagappan, N., Maximilien, E. M., Bhat, T., and Williams, L. (2008). Realizing
29 quality improvement through test driven development: results and experi-
30 ences of four industrial teams. *Empirical Software Engineering*, 13:289–302.
31
32 Pancur, M. and Ciglaric (2011). Impact of test-driven development on produc-
33 tivity, code and tests: A controlled experiment. *Information and Software
34 Technology*.
35
36 Rafique, Y. and Mistic, V. B. (2013). The effects of test-driven development on
37 external quality and productivity: A meta-analysis. *IEEE Transactions on
38 Software Engineering*, 39(6):835–856.
39
40 Rodriguez, P., Markkula, J., Oivo, M., and Turula, K. (2012). Survey on agile
41 and lean usage in finnish software industry. In *Six International Symposium
42 on Empirical Software Engineering and Measurement*.
43
44 Salman, I., Tosun Misirli, A., and Juristo, N. (2015). Are students representa-
45 tives of professionals in software engineering experiments? In *Proceedings of
46 the 37th International Conference on Software Engineering-Volume 1*, pages
47 666–676. IEEE Press.
48
49 Sanchez, J. C., Williams, L., and Maximilien, E. M. (2007). On the sustained
50 use of a test-driven development practice at ibm. In *AGILE conference*, pages
51 5–14.
52
53 Shadish, W. R., Cook, T. D., and Campbell, D. T. (2001). *Experimental and
54 Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mif-
55 flin.
56
57 Siniaalto, M. (2006). Test driven development: Empirical body of evidence.
58 Technical report, Information Technology for European Advancement, Eind-
59 hoven.
60
61
62
63
64
65

- 1
2
3
4
5
6
7
8
9 Sjoeberg, D. I. K., Hannay, J. E., Hansen, O., Kampenes, V. B., Karahasanovic,
10 A., Liborg, N.-K., and Rekdal, A. C. (2005). A survey of controlled experi-
11 ments in software engineering. *IEEE Trans. Softw. Eng.*, 31(9):733–753.
12
13 Still, J. (2007). Experiences in applying agile software development in f-secure.
14 In Munch, J. and Abrahamsson, P., editors, *Product-Focused Software Process*
15 *Improvement*, volume 4589 of *Lecture Notes in Computer Science*, pages 3–3.
16 Springer Berlin Heidelberg.
17
18 Tosun-Misirli, A., Erdogmus, H., Juristo, N., and Dieste, O. (2014). Topic selec-
19 tion in industry experiments. In *3rd International Workshop on Conducting*
20 *Experiments in Software Industry (CESI)*.
21
22 Turhan, B., Layman, L., Diep, M., Shull, F., and Erdogmus, H. (2010). *Making*
23 *Software: What Really Works, and Why We Believe It*, chapter How Effective
24 is Test Driven Development? OReilly Press.
25
26 VersionOne (2013). 8th annual state of agile survey. Technical report.
27
28 Williams, L., Maximilien, E. M., and Vouk, M. (2003). Test-driven development
29 as a defect-reduction practice. In *14th International Symposium on Software*
30 *Reliability Engineering (ISSRE)*.
31
32 Winer, B. (1971). *Statistical Principles in Experimental Design*. McGraw-Hill
33 Series in Psychology, 2nd edition edition.
34
35 Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., and Regnell, B. (2012).
36 *Experimentation in Software Engineering*. Springer.
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65