Empir Software Eng (2017) 22:2763–2805
DOI 10.1007/s10664-016-9490-0

An industry experiment on the effects of test-driven development on external quality and productivity

Ayse Tosun[1] · Oscar Dieste[2] · Davide Fucci[3] · Sira Vegas[2] · Burak Turhan[3] · Hakan Erdogmus[4] · Adrian Santos[3] · Markku Oivo[3] · Kimmo Toro[5] · Janne Jarvinen[5] · Natalia Juristo[2,3]

Published online: 28 December 2016

Abstract 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. We aim to analyse the effects of

Communicated by: Nachiappan Nagappan

✉ Ayse Tosun
tosunay@itu.edu.tr

Oscar Dieste
odieste@fi.upm.es

Davide Fucci
davide.fucci@oulu.fi

Sira Vegas
svegas@fi.upm.es

Burak Turhan
burak.turhan@oulu.fi

Hakan Erdogmus
hakan.erdogmus@west.cmu.edu

Adrian Santos
adrian.santos1987@gmail.com

Markku Oivo
markku.oivo@oulu.fi

Kimmo Toro
kimmo.toro@f-secure.com

Janne Jarvinen
janne.jarvinen@f-secure.com

Natalia Juristo
natalia@fi.upm.es

1 Faculty of Computer Engineering and Informatics, Istanbul Technical University, Istanbul, Turkey

TDD on the external quality of the work done and the productivity of developers in an industrial setting. 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. 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. 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 Introduction

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

The key practices of TDD can be summarized as follows (Beck 2003; Erdogmus et al. 2005):

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

TDD's industrial relevance is evidenced by its consistently high ranking from software development managers (Emam 2003). TDD also gained a reputation as a challenging process to apply (VersionOne 2013). Therefore, TDD has not been adopted much in the industry, despite the fact that companies frequently use other agile practices. A recent survey of agile methods usage at the organizational level reports that TDD is not a very popular development methodology (reported as being used by only 1.7 % of respondents) among more than 400 professionals from 200 software companies (Rodriguez et al. 2012).

[2] Escuela Tecnica Superior de Ingenieros Informatics, UPM, Madrid, Spain

[3] Department of Information Processing Science, University of Oulu, Oulu, Finland

[4] Carnegie Mellon University, Moffett Field, CA, USA

[5] FSecure Corporation, Helsinki, Finland

Often, professionals select and employ a subset of agile practices (e.g. sprint planning, daily standup, and continuous integration) based on their interests and the effort required to implement a practice. Of those professionals who use agile practices, more than 80 % reported that they employed Scrum practices (daily standup, iteration planning, unit testing, self-organizing teams), whereas eXtreme Programming practices were used by 18 % of professionals (Rodriguez et al. 2012).

Yet, there is industry interest in the study of TDD and its effects on cost reduction, productivity, process, and product quality. A longitudinal case study at IBM between the years 2001 and 2006 over 10 releases of a software application revealed that TDD took considerable amount of time at the beginning, but it paid back in terms of more robust code with better quality (Sanchez et al. 2007). Latorre (2014b) reported a successful application of TDD for a period of five and a half months in the industrial environment. The adoption of the developers to this new development approach in Latorre (2014b) led to timely product delivery, reductions in the number of residual errors, and easier-to-maintain source code. In terms of productivity, the time spent at the beginning for coordination and requirements analysis were higher in TDD, but the functional testing took much less time due to increased unit testing (Latorre 2014b).

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

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

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

In this research, we report an experiment conducted at three different sites of a software company in order to analyse the effects of TDD on the external quality of the work done and the productivity of the developers who performed the work. 24 professionals from the company attended a three-day training and experimentation session on TDD. We found that TDD did not improve external quality compared to incremental test-last development. Regarding productivity, the results were confounded by the tasks selected to implement the specified development approach.

2 Related Work

In this section we summarize the results of three literature reviews (Rafique and Misic 2013; Turhan et al. 2010; Munir et al. 2014) on TDD, and in particular discuss the discrepancies among them with respect to how they classify the studies covered. We highlight inconsistent findings with respect to quality and productivity, and lack of industrial experiments in the literature. We then report in detail the only three TDD experiments conducted with professionals in industrial settings.

2.1 Literature Reviews on TDD

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

Munir et al. (2014) report on the results of a systematic review that identifies 41 empirical studies of TDD and analyse them with respect to rigour and relevance scores using an assessment rubric. Based on a prior definition by Ivarsson and Gorschek (2011), rigour concerns adhering to practices of applying and reporting a research methodology, whereas relevance relates to the practical impact and realism of the research setup. Munir et al. (2014) classify 41 empirical studies with respect to the research methodology reported in these studies. The majority of the empirical studies covered by the authors (among which 19 are classified as experiments and one is classified as a case study) with high rigour and low relevance scores indicate that there is no difference between TDD and test-last development for a number of variables: productivity, external quality, internal code quality, number of tests, effort expended for TDD, size of the project and developer opinion. Among the 29 high-rigour studies, only four are identifiable as studies with professionals, three of which are experiments with professionals (Canfora et al. 2006; George and Williams 2004; Geras et al. 2004), with the remaining one being an empirical study of TDD with a single developer (Madeyski and Szala 2007). The analysis of the studies classified as case studies and surveys was more conclusive. Based on seven case studies and two surveys with high rigour and high relevance scores, the authors conclude that the developers perceived an improvement in quality. This finding is consistent with that of Turhan et al. (2010) regarding external quality.

Rafique and Misic (2013) conducted a systematic meta-analysis of 27 empirical studies that investigate the impact of TDD on external quality and productivity. Eight out of 27 empirical studies were classified as industry experiments. Three out of these eight studies classified themselves as controlled experiments (Canfora et al. 2006; George and Williams 2004; Geras et al. 2004), while the rest of the studies classified themselves as "industrial case studies" that report on quantitative effects of TDD. Incidentally, Munir et al. (2014) classified the same three controlled experiments (Canfora et al. 2006; George and Williams 2004; Geras et al. 2004) as high-rigour experiments with professionals. In this meta-analysis, Rafique and Misic (2013) report that both the quality improvement and productivity drop are much larger in industrial studies than in academic studies. They also observed that improvements in quality are significantly larger when the differences in test effort using TDD and task size were substantial.

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

2.2 TDD Experiments with Professionals

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

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

George and Williams (2004) investigated TDD versus a waterfall-like methodology with respect to code quality, external quality and productivity. Three experiments were conducted in three different companies. 24 participants with different 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 tasks. The study findings show that there is little or no difference in productivity, although there are differences in the frequency of unplanned failures. The process that the subjects were supposed to follow in Geras et al. (2004) was dictated by "scripts" or simple instructions based on PSP. However the authors state that the subjects were not sufficiently familiar with PSP or how to follow these scripts, raising a validity threat related to process conformance and study findings.

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

	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	External code quality: Number of test cases passed	Productivity: Time required to develop the task compared to the estimation	External quality: Percentage of acceptance tests passed over tackled sub-tasks

Table 1 (continued)

	Canfora et al. (2006)	George and Williams (2004)	Geras et al. (2004)	Our experiment
	Unit test quality: Mean number of assertions per method, Total number of assertions	Productivity:Total time spent for implementing the tasks	Testing quality: Test case density(test cases/KLOC), test cases per hour, Test case evaluations per hour	Productivity: Percentage of passed assert statements
		Internal code quality: OO metrics, code coverage	Failure: Unplanned failure rate per test case	
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)

In this research, we conduct an *industrial experiment with software professionals*. We summarize our motivation behind running a TDD experiment with professionals as follows:

- **Inconsistent findings:** As briefly summarized above, the findings on the effects of TDD on developers' productivity and external code quality differ in academia and industry depending on the subjects, tasks, and treatments used in the experiments (Siniaalto 2006; Rafique and Misic 2013). The findings in TDD studies with different rigour and relevance scores also come to different conclusions (Munir et al. 2014). The findings of the three TDD industry experiments are also inconclusive, because they all suffer from several validity threats, such as small sample size, lack of detailed information about the sample population, subjective measures of productivity (based on what the participants recorded), and mixed factors confounded with TDD (PSP or pair programming). They did not report the effects of TDD on the external quality of the work, nor did they use the same measures regarding productivity.
- **Few industry studies:** In a recent meta-analysis of the effects of TDD on external quality and productivity (Rafique and Misic 2013) classified only eight out of 27 studies as industrial studies. Many case studies and studies with students claim external quality improvements with TDD, but more evidence is needed in industrial contexts (Munir et al. 2014).
- **Very few industry experiments:** We investigated the eight industrial studies listed in Rafique and Misic (2013) and found that only three of them were truly controlled experiments, while the rest would be more appropriately considered as case studies or surveys with quantitative analyses.

Based on these observations, we believe that further evidence is necessary through experimentation on the effects of TDD on quality and productivity. Our study complements and extends previous TDD experiments in industry in several ways. We aimed to increase participation and were successful in recruiting more professionals compared to George and Williams (2004) and Geras et al. (2004). We collected detailed demographics data on the participants' experience in programming, unit testing, and other relevant knowledge prior to experimentation so that we could better characterize the population. We organized a three-day workshop in the company in which two days were dedicated to training in unit testing and TDD with hands-on individual and group exercises to make sure the participants' level of knowledge is appropriate for TDD experimentation. We made sure that all the participants attended the training and completed the control and experimental tasks individually. Compared to the studies by Canfora et al. (2006) and Geras et al. (2004), we set out to investigate more variables, namely the external quality of the work done. These aspects reduced certain internal validity threats related to process conformance (Fucci et al. 2014), mortality, and learning. Our experimental design also differs from the prior studies to fit the context and purpose of the study: We explain our reasoning behind the selection of a repeated-measures design in Section 3.4. Finally, our metrics and their calculations differ from Canfora et al. (2006) and George and Williams (2004) regarding productivity, and from George and Williams (2004) regarding external quality. We select more granular measures to increase their reliability. The metric choices are explained in Section 3.2. In the next subsection, we give contextual information about the software company in which the experiment was conducted.

2.3 Context

The software company in which we conducted our experiment is FSecure. It has been operating at a multinational level for over 25 years and has over 1000 employees in 20 sites around the world. It provides online security services and products for protecting digital devices and the digital data of consumers and businesses. The company strives to understand and adopt new trends and technologies in software development as well as in security and digital privacy. FSecure has both technical and business-related challenges, such as coping with frequently changing requirements, pressure to decrease time-to-market, and pressure for increasing quality (Still 2007). It is one of the leading companies in applying agile development methods (Still 2007).

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

3 Experimental Design

3.1 Research Objectives

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

Analyse *TDD and incremental test-last approach*
for the purpose of *comparing them*
with respect to the *external quality* of the resulting work and *productivity* of developers
from the point of view of the *researchers*
in the context of *a software company*.

Our research questions are listed below:

- Does TDD affect external code quality compared to an incremental test-last approach?
- Does TDD affect developers' productivity compared to an incremental test-last approach?

With these research questions in mind, in the remainder of this section, we discuss the study design and the process followed based on the proposed reporting structure for experiments by Jedlitschka and Pfahl (2005). We expand the structure in Sections 3.8, 4.2 and 3.10 (Instrumentation, Preparation and Interpretation) to clarify our approach regarding the metrics, training, discussion on the results and limitations.

3.2 Variables

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

The effectiveness of the programming approach can be examined from different perspectives. Previous related research has used several dependent variables and metrics, such as productivity, external quality, internal code quality, effort/time and conformance (Munir et al. 2014).

In this experiment, we adopt the same variables and metrics used in Erdogmus et al. (2005) and Fucci and Turhan (2013). We study *external quality* (QLTY) and *productivity* (PROD) as dependent variables. Using the same 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 (1). In the equation, n is the total number of subtasks compromising the measured task.

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

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

$$QLTY = \frac{\sum_{i=1}^{\#tst} QLTY_i}{\#tst} \tag{2}$$

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

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

$\#Assert_i(Pass)$ 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:

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

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 could be considered more productive (Madeyski and Szala 2007); this is not desirable. Similarly, if we choose number/percentage of user stories or subtasks implemented, subjects who claim to have implemented more subtasks could be considered more productive without supporting objective evidence, for example even when most of the acceptance test cases associated with the subtasks do not pass.

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

3.3 Hypotheses

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

H_{Q0}: $\mu(QLTY)_{ITLD} = \mu(QLTY)_{TDD}$ (Null Hypothesis)
H_{Q1}: $\mu(QLTY)_{ITLD} \neq \mu(QLTY)_{TDD}$ (Alternative Hypothesis)
H_{P0}: $\mu(PROD)_{ITLD} = \mu(PROD)_{TDD}$ (Null Hypothesis)
H_{P1}: $\mu(PROD)_{ITLD} \neq \mu(PROD)_{TDD}$ (Alternative Hypothesis)

3.4 Design

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

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

1. All of the subjects are familiar with ITLD, and some of the subjects are already experienced with TDD. We assign a sample of these subjects to the treatment group, G2, and a sample of the remainder (the ones without any experience with TDD) to the control group, G1.

Table 2 Two-level, basic between-subjects design

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

2. All of the subjects are familiar with ITLD, and none of the subjects are experienced with TDD. In this case, we train a sample of the subjects in TDD and assign them to G2, with the remaining subjects assigned to G1.

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.

In a repeated-treatment design, subjects are all matched with themselves, cancelling out any inherent variability and increasing power greatly (Shadish et al. 2001). A power analysis using a generalized linear model for the design in Table 3 shows that 128 subjects (64 per group) are necessary to generate a medium-sized effect (Cohen's d=0.5) with a resulting chance of Type-I and Type-II errors of $\alpha = 0.05$ and $\beta = 0.2$, respectively (Cohen 1992). If the assumptions of normality and homoscedasticity are not met, the required sample size would be even larger. Using the same parameters in a repeated-treatment design, on the other hand, the sample size drops to 34 subjects. We decided that this sample size was more reasonable in the context of our industry partner, and in turn, we adopted the design

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

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

Table 4 Repeated-treatment design with the training course

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

provided in Table 4. We present our final design, with training-treatment sequences and the assigned tasks for treatments, in Section 3.6 (Table 5).

3.5 Experimental Objects/Tasks

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

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

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

- Matrix initialization and assignment of obstacles to cells
- Command parsing
- Forward and backward moves
- Left and right turns

The forward and backward moves are the most complex operations. Command parsing and left/right turns are straightforward operations. The assignment of obstacles to cells upon initialization requires some parsing and type casting. MR is not a particularly difficult task, although it may require a few cycles of debugging.

We provided the MR specification document and a Java project template to the subjects in order to get them started easily and to have a common package structure that would

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

Training	1st Exp. Session	Training	2nd Exp. Session	3rd Exp. Session
ITLD	ITLD *greenfield task* MarsRover	TDD	TDD *greenfield task* Bowling Scorekeeper	TDD *brownfield task* MusicPhone

make data collection easier to automate. The project template consisted of 17 lines of non-commented code (LOC), a Java class that exposes the signature of the public API required by the task (8 LOC), and a test class containing the stub of a JUnit test case (9 LOC).

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

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

- Add a frame or bonus throws
- Detect when a frame is a spare or strike
- Calculate a frame score
- Calculate the game score

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

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

Both MR and BSK are relatively straightforward greenfield tasks, with some intricate logic that requires attention. In BSK, the complexity was represented by the tricky logic of bonus throws, the handling of frames, and interactions between the scores of subsequent frames. Comparing structural complexities between MR and BSK is, in fact, open to discussion; we believe they are comparable in many aspects. The process of managing, maintaining and acting upon several possible system states is similar in MR and BSK. Rover movements in MR and score calculation in BSK both depend on previous states of the system.

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

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

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

- Compute the distance between two geographical coordinates (given the formula)
- Recommend artists
- Find concerts for artist (given the recommendations)
- Compute an itinerary for the concerts

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

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

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

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 Misic 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.

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

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

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

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

3.9 Data Collection & Measurement Procedures

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

As shown in Fig. 1, the training sessions included several exercises. These exercises are called "katas" (Draper 2006) and were implemented in a solo or "randori" (a sequence of pair programming sessions rotated every 5-10 minutes 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

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

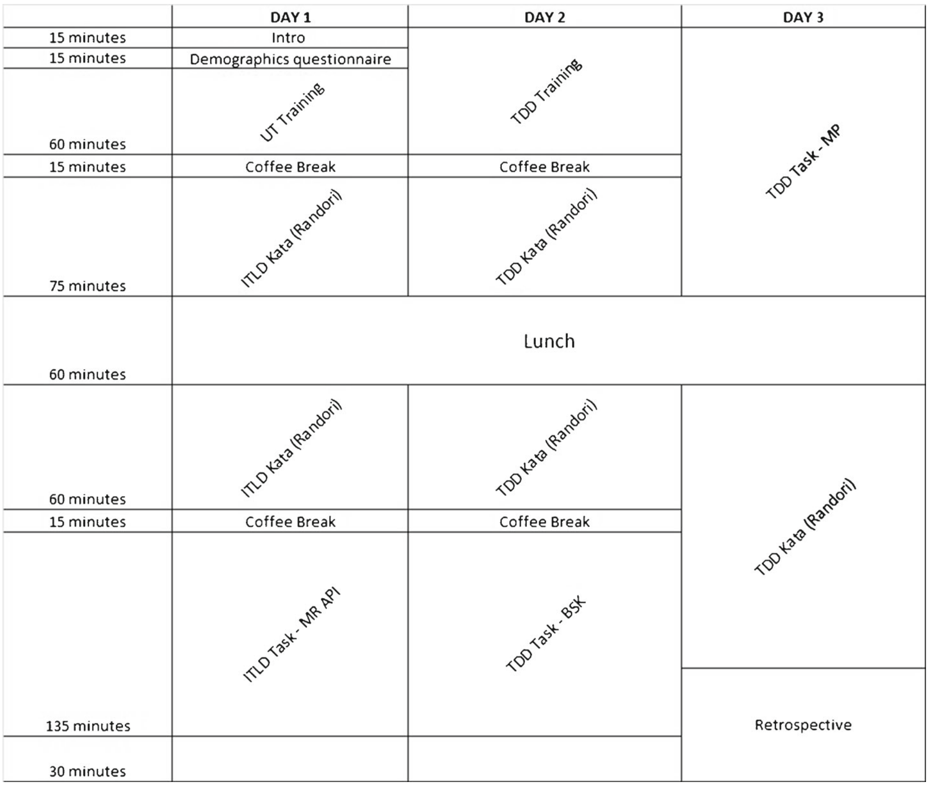
	MarsRover API		MusicPhone		Bowling Scorekeeper	
Subtask	Tests	Assert	Tests	Assert	Tests	Assert
ST1	1	1	4	4	3	3
ST2	7	11	3	12	3	3
ST3	4	8	3	12	2	2
ST4	4	7	4	4	3	10
ST5	4	8	10	26	5	5
ST6	8	8	8	12	6	6
ST7	6	15	7	17	8	8
ST8	4	8	1	1	5	5
ST9	3	8	2	11	5	5
ST10	3	7	2	13	4	4
ST11	8	8	4	20	2	2
ST12	NA	NA	NA	NA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

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 (1) and the PROD metric according to (4).

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., thath 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.



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

Fig. 1 The experimental schedule (planned)

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 differ when applied to comparable tasks and the simpler of the tasks.

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

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

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

3.11 Evaluation of Design Validity

Jedlitschka and Pfahl (2005) recommend that potential threats to the experiment validity due to the choice of the design are discussed and reported separately. We cover them in this section.

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

- Fatigue: We conducted experiments on each site over three days during regular working hours. During this period, the developers could not perform normal work activities; their only responsibility was participation in the experiment and the related training. Therefore, the experiment did not induce any extra effort on the subjects. In fact, their schedule during the experiment was more relaxed than a regular workday. The participants may have felt more energetic closer to the beginning of the training, and as the training progressed, could get more fatigued and less motivated affecting 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-las t

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 demographics 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.

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

# 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- ≤ 5	18.2	18.2	27.3	36.4	18.2
6- ≤ 10	45.5	40.9	36.4	13.6	0
> 10	27.3	40.9	18.2	4.5	0

Table 8 Detailed demographics for the subjects

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

4.2 Preparation

Schedule At the beginning of the training, we presented the planned schedule (Fig. 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 Fig. 1, the training sessions alternated with the treatments.

- **Lectures**: We delivered two lectures, each one hour long. The first lecture 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 Web[1] 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 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 is followed by the boxplots for all the metrics to visualize variations in the data. Then we apply the two-stage

[1]e.g. http://craftsmanship.sv.cmu.edu

analysis described in Section 3.10 to test the hypotheses related to the research questions. We report statistical significance, observed power, and effect size for all tests. We also check for normality test assumptions.

5.1 Quality (QLTY)

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

5.1.1 Descriptive Statistics for QLTY

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

TDD with the Greenfield Task has the Highest Mean in QLTY (76 %) with a Confidence Interval Between 66.4 % and 85.6 % The mean for TDD with the brownfield task has the lowest mean (39.1 %) with a confidence interval between 28.4 % and 49. %. The mean for ITL is higher than TDD with the brownfield task (53.9 %) with a confidence interval between 35.4 % and 72.5 %. The trimmed means are almost the same as the means in the three tasks (54.4 % for MR, 39.4 % for MP and 78.5 % for BSK). In terms of dispersion, ITL has the greatest variance, followed by TDD with the brownfield task. TDD with the greenfield task has the smallest variance.

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

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

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

5.1.2 Hypothesis Testing for QLTY

We performed the repeated-measures GLM on the QLTY metric. Mauchly's sphericity test showed that we cannot reject the null hypothesis for sphericity assumption ($p = 0.23$); therefore we could use the univariate test. The test confirms that there is a significant difference between the mean QLTY obtained 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 9 Descriptive statistics for QLTY

	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 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.

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 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

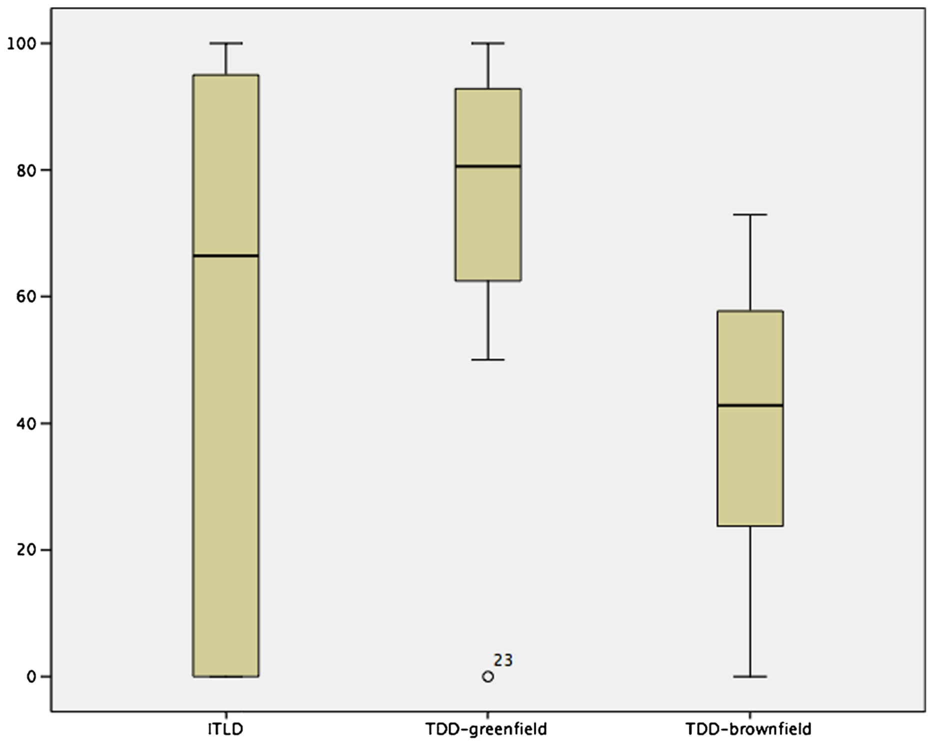


Fig. 2 Box plot representing the QLTY metric for ITLD and TDD with the greenfield and brownfield tasks

between 11.1 % and 20.7 %. The trimmed mean shows almost the same values as the means in all three tasks.

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 Fig. 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

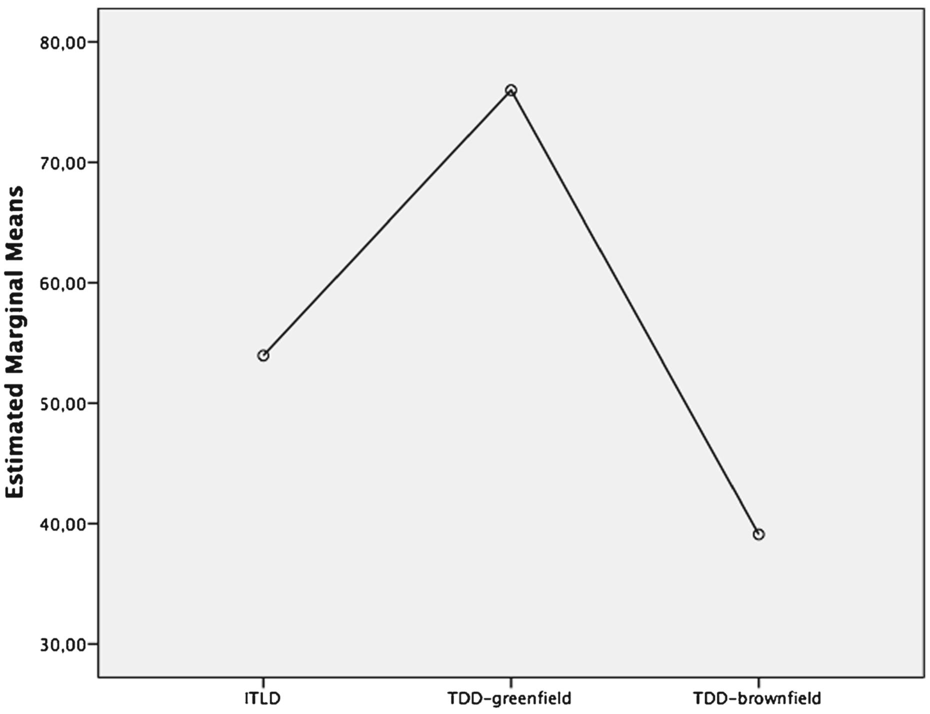


Fig. 3 Profile plot for QLTY

Table 10 The estimated marginal means for QLTY

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

Table 11 Descriptive statistics for PROD

	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

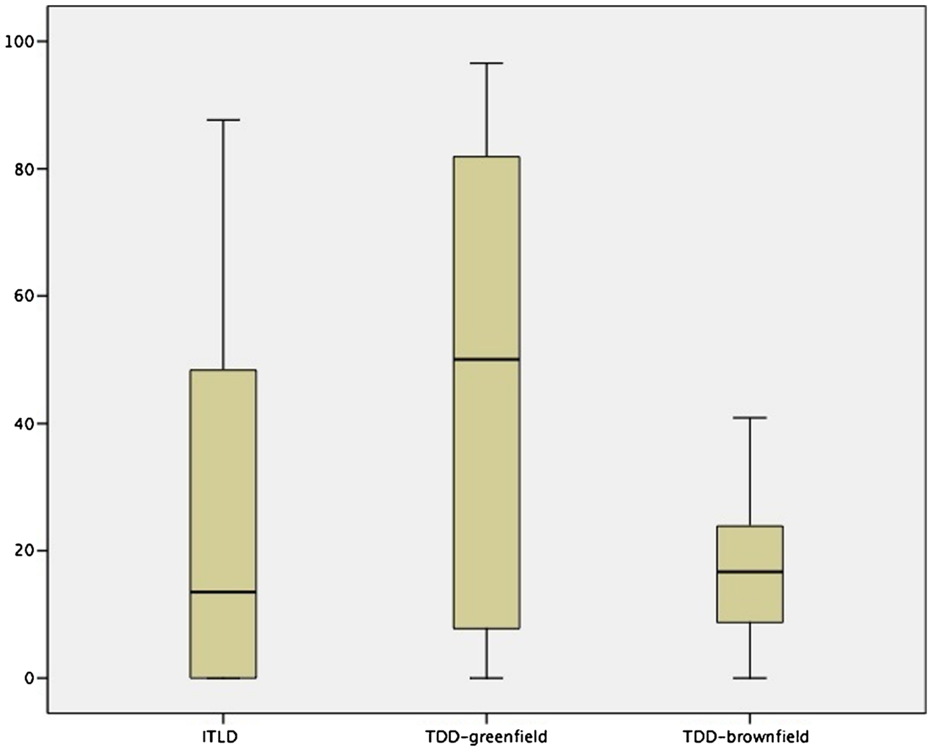


Fig. 4 Box plot representing the PROD metric for ITLD and TDD tasks

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 confounded 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.

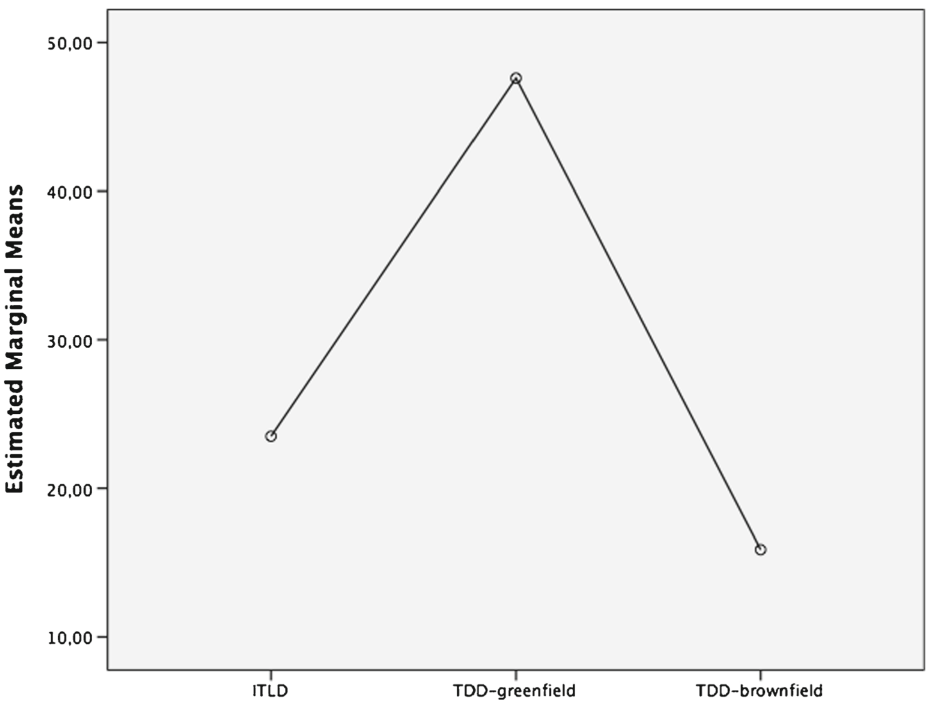


Fig. 5 Profile plot for PROD

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 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

Table 12 The estimated marginal means for PROD

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

Table 13 The marginal model results for PROD

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

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 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 on the demographics survey, test-last development was well known and used by the subjects, even if they did not master it in the same incremental way that the ITLD approach dictated. So during the control treatment, there was less risk of task complexity affecting the performance of the subjects.

When we compare the results of a simple task developed with TDD (e.g. BSK) with those of a simple task developed with ITLD (e.g. MR), a task selection bias is possible. In fact, the decrease in terms of quality and productivity in the second, brownfield TDD task suggests that this bias was present. The MIXED model confirmed this: The goal of the second

Table 14 Modified marginal means of the development approaches 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

stage analysis was to see whether TDD with a complex task would still be significantly different than ITLD. But the results did not confirm this hypothesis. Increased task complexity obscured the effect of the development approach.

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

7 More Insights into the Experiment Artefacts

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

On the Subjects' Performance The subjects wrote several unit test cases to implement the subtasks of the three tasks, namely MR, BSK and MP. Some subtasks may be tested with a single test case in which there exists a single assert statement (e.g. testing whether the MarsRover stays in the initial position (0,0,N) after executing an empty (“ ”) command), whereas other subtasks may be tested with more than one test case with several assertions. The subjects designed their source code, i.e., classes and methods, and its associated test cases differently from each other: Some subjects preferred to implement multiple assertions inside a test method, whilst the others followed the approach of implementing a single assert statement in a test method. Furthermore, many subjects wrote additional test cases that check methods that are not related to a single functionality (e.g. testing whether a setter function worked properly). Therefore, it was not possible to define a one-to-one relation between the unit test cases written by the subjects and the tackled subtasks.

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

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

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

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

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

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

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)s, 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 significantly in the interpretation of results.

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

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

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

9 Threats to Validity

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

9.1 Conclusion Validity

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

9.2 Internal Validity

We have already discussed potential threats to internal validity such as fatigue, carry-over, order, and practice effects in Section 3.11. Additionally, our experiment could be subject to attrition threat (loss of participation), selection threat (subject selection) and instrumentation threat (task selection). Two subjects dropped out after the first day (before any treatments were applied). The rest of the subjects stayed for the remainder of the study and during the administration of all treatments. We think the reason for the two subjects leaving the study might have been due to their workload or due to the training not meeting their expectations. We conclude that our experiment did not suffer from a signification attrition threat.

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

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

Other internal validity issues such as history (applying treatments at different times), mortality (leaving the treatments before completion) were not applicable in the context of this experiment.

9.3 External Validity

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

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

Regarding the subjects, our sample consisted entirely of professionals, who were novice to senior developers. All had experience in some agile software development practices. Around half of them indicated that they had prior knowledge of TDD. We talked with these subjects and learned that those subjects individually attended training/workshops on TDD, but they never applied this practice before participating to our study. We believe our results can be generalized to professionals who do not have prior hands-on experience of TDD, but have knowledge and experience with other agile practices.

9.4 Construct Validity

We used Java as the programming language during both training and treatments, and used the same type of instrument (acceptance test suites) to measure dependent variables for all three tasks. Hence, we addressed possible threats regarding the use of different programming languages or tools to measure dependent variables.

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

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

10 Conclusion and Future Work

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

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

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

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

References

Eclipse helios (2014). https://www.eclipse.org/helios/

Oracle virtual box 4.3 (2014)

The bowling game kata (2015)

Aniche MF, Gerosa MA (2010) Most common mistakes in test-driven development practice: results from an online survey with developers. In: Third international conference on software testing, verification and validation workshop

Basili V (1992) Software modeling and measurement: the goal/question/metric paradigm. Technical Report CS-TR-2956, UMIACS-TR-92-96, University of Maryland

Beck K (2003) Test driven development: by example. Addison Wesley

Becker K, Pimenta MS, Jacobi RP (2014) Besouro: a framework for exploring compliance rules in automatic tdd behavior assessment. Information and Software Technology

Bergersen GR, Sjøberg DIK, Dybå T (2014) Construction and validation of an instrument for measuring programming skill. IEEE Trans Softw Eng 40(12):1163–1184

Canfora G, Cimitile A, Garcia F, Piattini M, Visaggio CA (2006) Evaluating advantages of test driven development: a controlled experiment with professionals. In: ISESE, pp 364–371

Causevic A, Sundmark D, 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, 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, pp 337–346

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. Psychol Bull 112(1):155–159

Draper D (2006) Dojo, kata or randori?

Ellis PD (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, 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, Turhan B (2013) A replicated experiment on the effectiveness of test-first development. In: 2013 ACM / IEEE International symposium on empirical software engineering and measurement, pp 103–112

Fucci D, Turhan B, Juristo N, Dieste O, Tosun-Misirli A, Oivo M (2015) Towards an operationalization of test-driven development skills: an industrial empirical study. Inf Softw Technol 68:82–97

Fucci D, Turhan B, Oivo M (2014) Impact of process conformance on the effects of test-driven development. In: Proceedings of the 8th ACM/IEEE International symposium on empirical software engineering and measurement. ACM, p 10

Gamma E, Beck K (2014) Junit testing framework. https://github.com/junit-team/junit/wiki/Download-and-Install

George B (2002) Analysis and quantification of test driven development approach. Master's thesis, NC State University

George B, Williams L (2003) An initial investigation of test driven development in industry. In: ACM Symposium on applied computing

George B, Williams L (2004) A structured experiment of test-driven development. Inf Softw Technol 46(5):337–342. Special issue on software engineering, applications, practices and tools from the {ACM} symposium on applied computing 2003

Geras A, Smith M, Miller J (2004) A prototype empirical evaluation of test driven development. In: 10th International symposium on software metrics (METRICS)

Ivarsson M, Gorschek T (2011) A method for evaluating rigor and industrial relevance of technology evaluations. Emp Softw Eng 16:365–395

Jedlitschka A, Pfahl D (2005) Reporting guidelines for controlled experiments in software engineering. In: International symposium on empirical software engineering

Juristo N (2016) Experiences conducting experiments in industry: the eseil fidipro project. In: 4th International workshop on conducting empirical studies in industry. ACM

Kampenes VB, Dyba T, Hannay JE, Sjoberg DI (2007) A systematic review of effect size in software engineering experiments. Inf Softw Technol 49(11–12):1073–1086

Kim H-YY (2013) Statistical notes for clinical researchers: assessing normal distribution (2) using skewness and kurtosis. Restor Dent Endod 38(1):52–54

Kollanus S (2010) Test driven development - still a promising approach? In: 7th International conference on the quality of information and communications technology, pp 403–408

Latorre R (2014a) Effects of developer experience on learning and applying unit test-driven development. IEEE Trans Softw Eng 40(4):381–395

Latorre R (2014b) A successful application of a test-driven development strategy in the industrial environment. Emp Softw Eng 19:753–773

Madeyski L, Szala L (2007) Lecture notes in computer science, chapter the impact of test-driven development on software development productivity: an empirical study. Springer, pp 200–211

Maximilien EM, Williams L (2003) Assessing test-driven development at ibm. In: International conference on software engineering (ICSE)

McCulloch CEC, Searle S (2000) Generalized, linear, and mixed models. Wiley

Munir H, Moayyed M, Petersen K (2014) Considering rigor and relevance when evaluating test driven development: a systematic review. Inf Softw Technol 56:375–394

Nagappan N, Maximilien EM, Bhat T, Williams L (2008) Realizing quality improvement through test driven development: results and experiences of four industrial teams. Emp Softw Eng 13:289–302

Pancur M, Ciglaric (2011) Impact of test-driven development on productivity, code and tests: a controlled experiment. Inf Softw Technol

Rafique Y, Misic VB (2013) The effects of test-driven development on external quality and productivity: a meta-analysis. IEEE Trans Softw Eng 39(6):835–856

Rodriguez P, Markkula J, Oivo M, Turula K (2012) Survey on agile and lean usage in finnish software industry. In: Six international symposium on empirical software engineering and measurement

Salman I, Tosun Misirli A, Juristo N (2015) Are students representatives of professionals in software engineering experiments? In: Proceedings of the 37th international conference on software engineering, vol 1. IEEE Press, pp 666–676

Sanchez JC, Williams L, Maximilien EM (2007) On the sustained use of a test-driven development practice at ibm. In: AGILE conference, pp 5–14

Shadish WR, Cook TD, Campbell DT (2001) Experimental and quasi-experimental designs for generalized causal inference. Houghton Mifflin

Siniaalto M (2006) Test driven development: empirical body of evidence. Technical report, Information Technology for European Advancement, Eindhoven

Sjoeberg DIK, Hannay JE, Hansen O, Kampenes VB, Karahasanovic A, Liborg N-K, Rekdal AC (2005) A survey of controlled experiments in software engineering. IEEE Trans Softw Eng 31(9):733–753

Still J (2007) Experiences in applying agile software development in f-secure. In: Munch J, Abrahamsson P (eds) Product-focused software process improvement, volume 4589 of lecture notes in computer science. Springer Berlin Heidelberg, pp 3–3

Tosun-Misirli A, Erdogmus H, Juristo N, Dieste O (2014) Topic selection in industry experiments. In: 3rd International workshop on conducting experiments in software industry (CESI)

Turhan B, Layman L, Diep M, Shull F, Erdogmus H (2010) Making software: what really works, and why we believe it, chapter how effective is test driven development? O'Reilly Press

VersionOne (2013) 8th annual state of agile survey. Technical report

Williams L, Maximilien EM, Vouk M (2003) Test-driven development as a defect-reduction practice. In: 14th International symposium on software reliability engineering (ISSRE)

Winer B (1971) Statistical principles in experimental design, 2nd edn. McGraw-Hill Series in Psychology

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B (2012) Experimentation in software engineering. Springer

Ayse Tosun is an assistant professor at Faculty of Computer and Informatics Engineering, Istanbul Technical University (ITU), Istanbul, Turkey. Prior to joining ITU, she worked as a post-doctoral research fellow at Department of Information Processing Science, University of Oulu, Finland. She received her PhD in 2012, and MSc degree in 2008 from Department of Computer Engineering, Bogazici University, Turkey. Her research interests are empirical software engineering, more specifically mining software data repositories, software measurement, software process improvement, software quality prediction models, and applications of AI on building recommendation systems for software engineering.

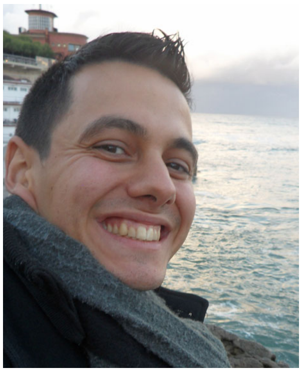
Oscar Dieste received his BS and MS in Computing from the University of La Coruña and his PhD from the University of Castilla-La Mancha. He is a researcher with the UPM's School of Computer Engineering. He was previously with the University of Colorado at Colorado Springs (as a Fulbright scholar), the Complutense University of Madrid, and the Alfonso X el Sabio University. His research interests include empirical software engineering and requirements engineering.

Davide Fucci is a post-doctoral researcher at the Empirical Software Engineering in Software, Systems and Services (M3S) research unit of the University of Oulu, Finland, where he also completed his Ph.D. His research interests include empirical software engineering, software quality and agile development methodologies. Dr. Fucci is a member of ACM and IEEE Computer Society.

Sira Vegas received her PhD degree from the Universidad Politcnica de Madrid in 1991. She is currently associate professor of software engineering at Universidad Politcnica de Madrid. Her main research interests are experimental software engineering and software testing. She is a reviewer of highly ranked journals such as IEEE Transactions on Software Engineering, Empirical Software Engineering Journal, ACM Transactions on Software Engineering and Methodology and Information and Software Technology. Dr. Vegas was program chair for the International Symposium on Empirical Software Engineering and Measurement (ESEM) in 2007. She began her career as a summer student at the European Centre for Nuclear Research (CERN, Geneva) in 1995. She was a regular visiting scholar of the Experimental Software Engineering Group at the University of Maryland from 1998 to 2000, and visiting scientist at the Fraunhofer Institute for Experimental Software Engineering in Germany in 2002.

Burak Turhan is a full Professor of Software Engineering at the Faculty of Information Technology and Electrical Engineering at the University of Oulu, Finland. His research interests are focused on: empirical studies of software quality and programmer productivity, software analytics through the application of machine learning and data mining methods for defect and cost modeling, studying human factors in software engineering - particularly cognitive biases -, and mining software repositories for grounded decision making, as well as agile/ lean software development with a special focus on test-driven development. He is a member of the editorial board of Empirical Software Engineering Journal, a steering group member and General Chair for PROMISE conference, and PC co-chair for PROMISE'13, ESEM'17 and PROFES'17 conferences. He is a member of IEEE, IEEE Computer Society, ACM, and ACM SIGSOFT. For more information and details please visit https://turhanb.net.

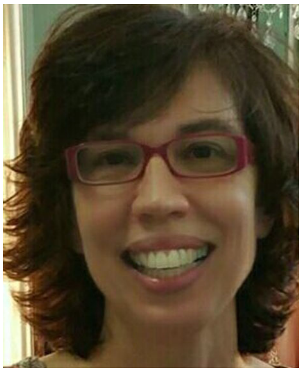
Hakan Erdogmus is an Associate Teaching Professor with the Department of Electrical and Computer Engineering at Carnegie Mellon University. His research interests are in empirical software engineering, software engineering methods, software quality, software engineering economics, and test-driven development techniques. He is a senior member IEEE and a member of ACM.



Adrian Santos received his MSc. in Software and Systems and MSc. in Software Project Management at Technical University of Madrid, Spain, and his MSc. in IT Auditing, Security and Government at the Autonomous University of Madrid, Spain. He is a PhD. candidate in the University of Oulu, Finland. His research interests include empirical software engineering, agile methodologies, statistical analysis and data-mining techniques.

Markku Oivo (PhD, eMBA) is full professor of software engineering at University of Oulu since 2002. He is head of the M3S and M-Group. He has 30+ years of research, R&D and management experience in academy and industry in Finland, US, France, Germany, Italy and Spain. During 2000-2002 he was Vice President and director of R&D at Solid Information Technologies. He held several positions at VTT in 1986-2000 including Professor and Head of Embedded SW R&D. He has held visiting positions at the University of Maryland (1990-91), Schlumberger Ltd. (Paris 1994-95), Fraunhofer IESE (1999-2000), University of Bolzano (2014-2015) and Technical University of Madrid (2015). He has worked at Kone Co. (1982-86) and at the University of Oulu (1981-82). He received the honor of First Class Knight of the White Rose of Finland in 2013. He is a founding member of ISERN. He has initiated and managed more than 100 national and international research projects and programs with tens of millions of euros at national and international scales. He has over 100 publications in international conferences and journals. He has served as chair and committee member in organizing numerous international conferences, and has been a reviewer for top journals. His research interests include empirical research, software engineering methods and processes, agile, lean, cloud, startups, and management of technology.

Dr. Janne Jarvinen is Director, External R&D Collaboration at F-Secure Corporation. Janne has over 25 years of experience in software business in different positions ranging from programmer to VP Engineering, both in small and large software companies. He has also been active in various industry-driven research programmes in national and international level such as Need4Speed (www.n4s.fi). Janne also has recently served as the Future Cloud Action Line Leader of EIT Digital. He is an IEEE member and holds a PhD in Information processing science from University of Oulu (2000).



Dr. Natalia Juristo (http://grise.upm.es/miembros/natalia/) is full professor of software engineering with the Computing School at the Technical University of Madrid (UPM) since 1997. She was awarded, starting in January 2013, a FiDiPro (Finland Distinguished Professor Program) professorship at University of Oulu. She was the Director of the MSc in Software Engineering from 1992 to 2002 and coordinator of the Erasmus Mundus European Master on SE (with the participation of UPM, University of Bolzano, University of Kaiserslautern and Blekinge Institute of Technology) from 2006 to 2012. Her main research interests are experimental software engineering, requirements and testing. Back in 2001 she co-authored the book Basics of Software Engineering Experimentation (Kluwer). Natalia is a member of the editorial boards of IEEE Transactions of SE, Empirical SE Journal and Software: Testing, Verification and Reliability journal. She has served in several conferences Program Committees (ICSE, RE, REFSQ, ESEM, ISESE, etc.), has been Program Chair (EASE13, ISESE04 and SEKE97) as well as General Chair (ESEM07, SNPD02 and SEKE01). Natalia was co-chair of ICSE Technical Briefings 2015 and co-chair the Software Engineering in Practice (SEIP) track at ICSE 2017. She began her career as a developer at the European Space Agency (Rome) and the European Center for Nuclear Research (Geneva). She was a resident affiliate at the Software Engineering Institute in Pittsburgh in 1992. In 2009 Natalia was awarded with a Honorary Doctor from the Blekinge Institute of Technology in Sweden.