On the Effectiveness of Unit Tests in Test-driven Development

Ayse Tosun
Istanbul Technical University
Istanbul, Turkey
tosunay@itu.edu.tr

Muzamil Ahmed
University of Oulu
Oulu, Finland
muzamil_kpr@hotmail.com

Burak Turhan
Brunel University London
Middlesex, UK
burak.turhan@brunel.ac.uk

Natalia Juristo
Universidad Politechnica de Madrid
Madrid, Spain
natalia@fi.upm.es

ABSTRACT

Background: Writing unit tests is one of the primary activities in test-driven development. Yet, the existing reviews report few evidence supporting or refuting the effect of this development approach on test case quality. Lack of ability and skills of developers to produce sufficiently good test cases are also reported as limitations of applying test-driven development in industrial practice. **Objective:** We investigate the impact of test-driven development on the effectiveness of unit test cases compared to an incremental test last development in an industrial context. **Method:** We conducted an experiment in an industrial setting with 24 professionals. Professionals followed the two development approaches to implement the tasks. We measure unit test effectiveness in terms of mutation score. We also measure branch and method coverage of test suites to compare our results with the literature. **Results:** In terms of mutation score, we have found that the test cases written for a test-driven development task have a higher defect detection ability than test cases written for an incremental test-last development task. Subjects wrote test cases that cover more branches on a test-driven development task compared to the other task. However, test cases written for an incremental test-last development task cover more methods than those written for the second task. **Conclusion:** Our findings are different from previous studies conducted at academic settings. Professionals were able to perform more effective unit testing with test-driven development. Furthermore, we observe that the coverage measure preferred in academic studies reveal different aspects of a development approach. Our results need to be validated in larger industrial contexts.

CCS CONCEPTS

• **Software and its engineering → Agile software development**; **Empirical software validation**; *Software testing and debugging*;

KEYWORDS

Test-driven development, unit testing, mutation score, code coverage, empirical study

ACM Reference Format:
Ayse Tosun, Muzamil Ahmed, Burak Turhan, and Natalia Juristo. 2018. On the Effectiveness of Unit Tests in Test-driven Development. In *ICSSP '18: International Conference on the Software and Systems Process 2018 (ICSSP '18), May 26–27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3202710.3203153

ICSSP '18, May 26–27, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-6459-1/18/05…$15.00
https://doi.org/10.1145/3202710.3203153

1 INTRODUCTION

Test-driven Development (TDD) was introduced as a software development practice in early 1960s during NASA's Mercury project [8]. It is popularized by Beck [7] as a way of managing fear (e.g. progressing slowly due to unnecessary thinking on up-front design, finishing higher complexity tasks at a single step) during programming. TDD is proposed to be a suitable methodology for agile development, as it encourages thinking on the functionality ahead of writing the code, refactoring the code frequently and continuously maintaining high code quality, and increasing the maintainability and reliability of the code through running automated regression test cases [7, 21]. For more than a decade, many empirical studies have been conducted to examine the effects of TDD on different project attributes. Most of the reported work on TDD studied its effects on internal code quality, external code quality and developer productivity [26, 30, 34, 41]. Literature reviews [30, 41] and meta-analysis [34] report inconsistent results on the effects of TDD on quality and productivity with respect to the study settings. Turhan et al. [41] report that TDD has a mixed effect on internal quality, positive effect on external quality, whereas it has an inconsistent effect on productivity when all studies' findings were aggregated. When studies of high rigor, i.e., controlled experiments and pilot studies designed with good constructs in medium to large scale studies, were considered, the results on quality remains inconsistent, whereas the results on productivity seems to be improving with TDD. Munir et al. [30] similarly classified empirical studies on TDD into four categories in terms of rigor and relevance. High rigor and relevance studies report significant improvements in external quality with a loss of productivity when TDD is applied. Rafique and Misic [34], in their meta-analysis, report that TDD, in general, has a positive effect on external quality, but little or no effect on productivity.

Effects of TDD on other attributes have been studied less often through experiments and case studies. Turhan et al. [41] reports 17 TDD trials that study the effect of this development approach on test quality captured by test productivity, test density, test coverage and test effort. Munir et al. [30] also reports the effects of TDD on the effort/time spent for coding and testing, developer opinion (on simplicity, ease of use, correctness), conformance to the rules of TDD and robustness.

Although TDD is still being investigated with more case studies conducted with professionals (e.g. [27]) and controlled experiments in academic settings (e.g. [18]), the industrial adoption of TDD has not been completed yet [29]. Causevic et al. [10], in their literature review, highlighted 18 factors that could limit the industrial adoption of TDD. These 18 factors influence the way TDD is applied. The limiting factors studied the most are increased development time, insufficient TDD experience, lack of upfront design, domain and tool specific issues, lack of skills in writing test cases, insufficient adherence to TDD protocol and legacy code [10]. The authors suggest investigating those limiting factors in future studies, and proposing guidance to overcome those for successful industrial adoption of TDD.

During TDD practice, testing is expected to be improved with a growing number of automated test cases [41]. Furthermore, quality of these test cases could be higher since they are written at a finer granularity level. Turhan et al. [41] state that the previous findings also support this argument despite insufficient evidence to reach a conclusion from industry. Madeyski [26] also highlights the importance of unit tests' effectiveness and thoroughness during TDD practice in terms of fault detection, since the coding is mainly driven by test cases during TDD, and effective and thorough test cases lead to a higher quality software product. Therefore, the immediate impact of TDD compared to a traditional test-last approach is expected to be on unit test cases. Considering the mentioned relation, we found few empirical studies that investigate the effects of TDD on unit test case effectiveness in terms of catching code defects [11, 12, 26]. These three studies conducted in academic settings report that there is minor or no difference between the effectiveness of unit tests implemented in TDD and test-last development. However, as highlighted in [41], there is a need to investigate the phenomenon in industrial settings to generalize the results.

In this paper, we report our empirical analysis based on an industrial experiment on TDD conducted with 24 professionals at a software organization. We analyse the effects of TDD on unit test case effectiveness with respect to mutation score indicator. We also measured the code coverage with respect to unit test cases, as all the prior academic experiments [11, 12, 26] also assessed the test effectiveness in terms of coverage. Our study further contributes to the existing literature by assessing the same hypotheses with professionals in an industrial setting.

2 RELATED WORK

In this section, we provide information about the objectives of unit testing and report different attributes associated with unit test (case) effectiveness from prior studies. We then present the experiment details and conclusions from three prior academic experiments [11, 12, 26] that investigated the effects of TDD on unit test cases.

2.1 Unit Testing

Hunt and Thomas [22] describe unit tests as a piece of code written by developers with the intention of exercising/testing a specific functionality on a small area of the code. Unit testing is an important process in software development, as it helps to spot problematic areas and reduce bugs in the source code [5]. Today, unit tests are easily automated and the programming language of unit tests is the same as that of production code [15]. These tests are put under revision control and are an explicit part of the code.

Earlier software testing techniques focused on defining and achieving higher level of code coverage (e.g. thorough testing) to verify the program structure. Several metrics for identifying the coverage criteria have been proposed, such as branch and statement coverage [36]. Later, this trend has moved towards fault-based testing techniques, also called mutation testing, which aims to test specifications and functionalities of a program with more data [36]. Mutation score indicator was proposed and extensively explored to analyse the fault detection effectiveness of test cases [31].

2.1.1 Unit test effectiveness. There are different views associated with a set of attributes proposed to measure the quality of unit test cases in the literature [9]. Kaczanowski [25] discusses three characteristics to measure test quality: Test smells, code coverage and mutation score indicator. Causevic et al. [11, 12] in their experiments also refer these as test quality attributes. A recent mapping study summarizes different ways to assess the quality of test-code among 28 primary studies [43]. Garousi et al. [43] highlights that around half of primary studies investigated test smells, whereas other studies used coverage as an indicator of test code/ assertions quality. Some studies on test smells argue that these issues degrade the quality of test code since test smells prevent the maintainability of test codes written by the development teams [20, 35]. Madeyski [26] on the other hand discusses the quality of test suite in terms of their thoroughness and effectiveness: Coverage metrics are indicators of test suite thoroughness, while mutation score indicator is found to be effective at finding faults [26]. Similar views on code coverage point out that high coverage do not necessarily indicate that a test suite is effective at catching faults [23]. Mutation testing, on the other hand, is related to test suite effectiveness in terms of test suite's ability to detect faults [4, 13, 24]. Following the argument by Madeyski [26], we also use mutation score indicator as a measure of unit test effectiveness in this experiment. Below, we summarize the literature on the three attributes (test smells, code coverage metrics and mutation score indicator) and justify our decision to choose mutation score indicator as the measure of unit test effectiveness.

2.1.2 Test smells. Test smell is similar to the term "source code smell" (when something does not look right in production code), but it is not as standardized as source code smells [25]. We can use static analysis tools (e.g. FindBugs [3]) for the test code as well; but test code is much simpler than production code, which makes it difficult to find any smells in the test code. The study by Deursen et al. [15] is the first that describes 11 test smells: mystery guest, resource optimization, test run war, general fixture, eager test, lazy test, assertion roulette, indirect testing, for testers only, sensitive equality and test code duplication. Deursen et al. [15] also state

that test code smells are different than production code smells. The difference in source code and test code smells depends on how test cases are implemented, organized, and how they interact with each other. The authors in [15] further provide specific refactoring activities to avoid these test smells. Kaczanowski [25] also recommends eliminating smells in both test and source codes by following the best programming practices. Studies on test smells often propose new test smell sub-categories such as general fixture and eager, and methods to detect those (e.g. [20], [35]), or they study the impact of test smells on maintenance activities (e.g. [32]). Rompaey et al. [35] proposes a set of metrics to identify general fixture and eager test smells, and suggest automating the detection of these smells through the metrics.

Although the research on test smells have been increasing with new tool and methods for detection and prevention, there is neither a common understanding of test smells in the literature, nor a specialized tool to identify specifically test smells. Furthermore, test smells are often considered as indicators of test code quality than a way to understand test suite effectiveness. For these reasons, we do not use test smells as unit test effectiveness in this study.

2.1.3 Code coverage. Testing is useless if one fails to execute a faulty element in the code [33]. To measure the quality of test cases, code coverage is a commonly used measure. It indicates which part of the production code (e.g. line, statements, and branches) is exercised/ executed during testing. 100 percent line coverage indicates that all lines of the code have been executed with the existing test suite. There are different ways to measure code coverage. A literature review by Shahid, Ibrahim, and Naz [38] proposes 12 different granularities of measuring code coverage, some of which are branch, instruction, line, cyclomatic complexity, instruction and method coverage. According to an empirical study [19], the use of coverage measures to determine the defect detection capability of a test suite could be dependent on the context. In industrial environments, statement coverage is found as the most effective predictor of test suite quality, although, branch coverage or some variants of path coverage may be more useful in research contexts [19]. Another study by Tengeri et al. [39] argues that the correlation between high code coverage and defect detection effectiveness of test suites is not always present or evident. Inozemtseva and Holmes [23] report similar findings in which authors study whether the statement, decision and modified condition coverage of test suites are correlated with test suite effectiveness in terms of mutation score. Results show that there is a low to moderate correlation between test suite effectiveness and code coverage when the size of test suite is controlled. Based on these recent studies, we also argue that code coverage does not necessarily indicate effectiveness of a test suite; rather it measures the thoroughness of a test suite [26]. Therefore, we do not use coverage as our main measure for test suite effectiveness in this experiment; but we refer to coverage analysis for comparing our results with the previous findings.

2.1.4 Mutation score indicator. Mutation testing is a way to measure the ability of a test case to catch defects in the code [31]. To verify how good a test suite is, multiple versions of the program code, in which different types of defects are injected, can be created [25]. According to Pezzè and Young [33], a mutant is a program which is different from the original program at one syntactic item. A mutation testing tool creates a mutant of the program by applying a mutation operator at a single location of the program [16]. After executing a test case, if a mutant can be distinguished from the original program, it is said that test case kills the mutant. If the mutant cannot be distinguished, it is called as an equivalent mutant [16]. Mutation testing is used to check the fault detection effectiveness of a given test suit. Mutation score indicator is calculated as the percentage of faults (mutants) detected by the test suite [28]. This score can be used to measure the effectiveness of a test set in terms of its ability to catch faults [14, 24]. A higher mutation score indicates higher effectiveness of test suite in detecting the faults. In order to compute mutation score of a program, all test cases in that program's test suite should pass before execution of a mutation tool.

According to [4], mutation operators generate mutants that are similar to real faults found in the system although they might be harder to detect than hand-seeded faults. Gopinath et al. [19] also argues that, in terms of defect detection ability, mutation testing could be considered as the most effective way to predict test suite effectiveness. Other coverage measures such as statement and branch coverage could also be correlated with mutation detection capability although these findings are dependent on the selection of a proper population of subjects and test suite. Following Gopinath et al.'s [19] claim that the development approach in which unit tests are generated could likely affect the interpretation of unit tests, we use mutation score indicator to measure the effectiveness of unit tests written in the context of two different development approaches.

2.2 Unit Test Effectiveness in TDD

There is some evidence suggesting that TDD improves test quality [41]. However, most of the evidence comes from pilot studies, while the majority of controlled experiments report no difference between TDD and alternative treatments. There is also insufficient evidence from TDD's industrial use to reach a conclusion. Authors point out that test case development is one of the primary activities of TDD, and hence, they would have expected stronger results regarding this approach's effect on test quality [41]. Causevic et al. [10] also state in their literature review that there is no explicit investigation of the quality of test cases written by developers during TDD apart from two studies. Those two studies that Causevic et al. [10] included in their review, on the other hand, reported difficulties of applying TDD when there is lack of ability and skills of the developer to produce sufficiently good test cases.

A recent systematic literature review conducted by Munir et al. [30] identifies patterns and themes used among the studies on TDD. Munir et al. [30] did not mention any category related to unit test effectiveness or any study on the impact of TDD on unit test effectiveness. However, the authors list branch coverage metric under internal code quality category, as it was previously used to quantify internal code quality in a TDD study. Furthermore, a set of metrics, such as the number of test case passed, total number of assertions passed/failed, and total number of test cases written, were discussed under the size and external quality categories [30].

We found one study (Madeyski [26]) with a focus on unit test effectiveness of TDD. Madeyski [26] performed an experiment with

a group of master students in an academic setting. They used branch coverage to evaluate the effects of test-first (TF) development on unit test thoroughness, and mutation score to evaluate the effects of TF on unit test effectiveness. The control treatment was test-last development. Their experiments did not reveal any significant differences in terms of branch coverage and mutation score of unit test cases between the two treatments. Causevic et al. [11, 12] later extended the experiment of Madeyski [26] in academia with master students. They conducted one experiment with 14 students, and published results of this experiment in two studies [11, 12]. They also found no difference in code coverage and mutation score of unit test cases written during a test first and a test last approach. The details of the experiments conducted by Madeyski [26] and Causevic et al. [11, 12] are summarized in Table 1.

The dependent variables of these three studies, listed in Table 1, are stated differently. Madeyski [26] define the dependent variable as the unit test quality composed of unit test thoroughness and effectiveness. Causevic et al. [11] refer to Madeyski's work and state that their study is in relation to the earlier experiment in terms of its goal. Causevic et al. [12] later rephrased their dependent variable to efficiency and effectiveness of test cases, although their overall goal stays the same in both of their papers. In all three studies, authors quantify unit test quality in terms of mutation score and code coverage. Madeyski [26] used branch coverage to represent unit test thoroughness. Causevic et al. [11, 12], on the other hand, used statement coverage. The interpretation of both metrics was also different by the two authors. Causevic et al. [11, 12] evaluate both mutation score and statement coverage as internal quality attributes of test cases. To evaluate the external quality of test cases, the authors further defined an additional metric called defect detection ability in [6], and total number of failing assertions in [7].

The independent variable in all the three studies, listed in Table 1, was the development approach with two levels: Test-first (TF) and Test-last (TL) programming. Note that TF is used to refer to TDD, whereas TL is used to refer to traditional test-last development. Madeyski [26] defines TL as an incremental development approach as TF, but the only difference between the two is when and how often subjects write unit tests during their practices. The author explicitly mentioned that he taught both techniques to the students prior to the experimentation. Causevic et al. [6, 7] state that participants in the TF group were instructed to use TDD, but they did not mention any training on TL. Experiments were conducted with graduate (MSc level) students. It is important to mention that in [11, 12], authors did not use statistical tests due to insufficient numbers of subjects, while Madeyski [26] used multivariate ANOVA after confirming its assumptions on data hold true.

Our study is an extension of those three studies. We partially share the research goal with Madeyski [26]: To evaluate unit test case effectiveness of TDD compared to ITLD. To further confirm or refute the previous findings on unit test thoroughness (as stated by Madeyski [26]), we also measure code coverage of unit tests in terms of method and branch coverage. We did not use statement coverage as in Causevic et al. [11]; instead, we choose method coverage as a complement to branch coverage. We believe that the change in development approach (from ITLD to TDD) might lead to writing fewer methods with more branches, whereas it may not affect the statements covered by unit tests. During TDD, subjects need to focus on writing tests associated with small pieces of functionality and incrementing their code for each new, failed test case. This approach does not require adding new methods for each failing test case. Hence, it is a different practice than ITLD, in which subjects complete their code for a small piece of functionality before testing, and whenever a new piece of functionality is to be implemented subjects tend to associate it with new methods.

Different than the studies in Table 1, we conduct an industrial experiment with professionals located in three different sites of a software organization. We believe our research fills a gap mentioned in all prior studies that more experiments in different contexts (e.g. industrial) is necessary to strengthen the existing findings, and to establish evidence-based recommendations [26]. We also choose two objects with similar complexity and compare unit test case effectiveness of those objects implemented in a TDD and ITLD fashion. The research in [11, 12] used EclEmma tool [1] for code coverage analysis in their experiments. For mutation score indicator, Judy [28] was used by all three studies. We also employed the same toolset for extracting metrics data in order to keep the measurement process consistent with the prior studies. Finally, we evaluate our hypotheses using non-parametric statistical tests.

3 EXPERIMENTAL DESIGN

This study uses data from an experiment conducted with professionals to observe the effects of TDD on external quality and productivity [40]. The experimental setup is therefore identical to the original experiment: the same group of professionals from a software organization, the same independent variables (TDD and ITLD), and the same programming tasks. Different from [40], we define a new research question to observe the effect of TDD on unit test effectiveness, collect data for our new dependent variable in this study, and extract new measures. In this section, we present the goal, its corresponding research question and hypotheses, the variables and the metrics of this study. The details of all the experimental setup can be found in [40].

Research objectives. The goal of this study is to understand the effects of TDD on unit test effectiveness. Hence, we formulate our research objective following the guidelines provided by [6]:

> *Analyze TDD*
> *For the purpose of evaluating its effects*
> *With respect to the effectiveness of unit tests*
> *Compared with Incremental Test Last Development*
> *From the point of view of professionals in industry.*

Based on our research objective, we define our research question as follows: *"What is the effect of TDD on unit test effectiveness compared to an incremental test last approach (ITLD)?"*

Variables. The **independent** variable of our study is software development technique, with two treatments: Incremental Test Last Development (ITLD) and Test-Driven Development (TDD). We compare TDD with a closely related process, which we call incremental test-last development (ITLD). Both TDD and ITLD follow the same, iterative steps except the order of the activities

Table 1: Summary of Related Studies

Title	[26]	[11]	[12]	**Our study**
Goal	Compare Test-First (TF) vs. Test-Last (TL) programming with regard to thoroughness and fault detection effectiveness of unit tests.	Compare the quality of test cases produced using test-first and test-last approaches	Compare efficiency and effectiveness of the testing effort produced by test-first and test-last developers.	Analyze the effect of TDD on unit test effectiveness compared to an incremental test last approach (ITLD)
Independent Variable	Development approach: TF or TL			TDD and ITLD
Metrics	Branch coverage, Mutation score	Defect detecting ability, Statement coverage, Mutation score	Total number of failing assertions, Statement coverage, Mutation score	Mutation score
Methodology	Controlled experiment			
Design	One Factor Two Treatments			Repeated measures design
Subjects	22 third and fourth-year graduate MSc software engineering students	14 (software engineering master) students		24 professionals
Objects	Web-based paper submission and review system	Bowling Score Keeper		Bowling Score Keeper and Mars Rover API
Results	No significant difference	No difference		Significant difference

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 [40]. 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 since this development technique forces a control that involves testing, making the comparison less biased and fairer [40].

The **dependent** variable of our study is unit test case effectiveness. We prefer to use this term "effectiveness" rather than unit test quality, since there are different attributes, e.g. internal, external as in [11], and test smells, coverage measures as in [25], associated with test suite quality. However, unit test effectiveness is often associated with mutation score in the literature (e.g. [4, 14, 24, 26]). Therefore, we also choose mutation score indicator as our metric. This metric is calculated based on the ratio between total number of killed mutants by your test suite and total number of non-equivalent mutations [28].

Hypotheses. Based on our research question, we define the null and alternative hypotheses.

- $H_0 \mu(MS)_{TDD} = \mu(MS)_{ITLD}$**:** There is no difference in the mutation score (MS) of unit tests implemented by subjects during TDD and ITLD.
- $H_1 \mu(MS)_{TDD} \neq \mu(MS)_{ITLD}$**:** There is a significant difference in the mutation score of unit tests implemented by subjects during TDD and ITLD.

Design. We choose experimentation as our research methodology. Experiments are usually performed in a laboratory environment that provides maximum control [42]. In an experiment, the goal is to manipulate one or more variables, while other variables are controlled at a fixed level. The effects of this manipulation are measured which later can be used to perform statistical analysis [42]. We performed an experiment in an industrial setting with professionals. We choose one factor (software development technique) with two treatments (TDD and ITLD) and follow a repeated measure design in our experiment [37]: The subjects of the experiment were exposed to both treatments sequentially, and hence, each subject is matched with herself cancelling out her inherent variability and increasing the power greatly [37]. We chose this design in the experiment since the volunteers wanted to participate all events (exercises and treatment tasks) and training. They did not prefer us to group the subjects into two groups as in prior experiments in Table 1. We reported the findings based on the two treatments in this paper. More details on the schedule, training, execution of the events and exercises are reported in [40].

Subjects. We conducted the experiment in three different sites of a large-scale organization [40]. The company operates at multinational level and provides security services and products to protect digital life of consumers and business. In total, 24 participants volunteered for the experiment at the chosen sites of company. These participants were novice developers with no hands-on-practice on TDD before. The detailed demographics of the participants were provided in the previous experiment report [40].

Objects. Participants were asked to implement two programming tasks (objects) which are matched with the two treatments of the

experiment: MarsRover API for ITLD treatment and Bowling Score Keeper for TDD treatment. Participants have to work with Java classes, objects and write Junit test cases to check the correctness of the output. The details of the tasks were provided in the previous experiment report [40].

Instrumentation. The participants used virtual machines during the experiment. Supporting software i.e. operating system, java development kit and tools required for development (i.e. eclipse) were pre-installed on these virtual machines [40]. For this study, we use Judy and EclEmma tools to collect mutation score and code coverage results respectively. Judy: Judy is a mutation tool for Java source code [28]. Judy supports 16 predefined mutation operators and works on FAMTA Light algorithm to achieve high mutation performance [28]. It generates mutants of the given software product at project level and reports the percentage of mutants that are killed by the associated test suite. EclEmma: It is a JaCoCo code coverage library [2] based tool to calculate code coverage for Eclipse. It works on Java byte code [1]. It provides code coverage analysis for instruction, line, branch, method, and cyclomatic complexity at class level.

Analysis procedure. Our analysis procedure starts with a comparison of descriptive statistics; we compare mean, median, standard deviation, minimum and maximum of mutation score values regarding the two treatments. Then, we evaluate our results graphically with box plots. Afterwards, we choose non-parametric statistical tests to check whether the null hypotheses can be rejected. In this study, we evaluate our hypotheses using Wilcoxon rank sum statistical test with a significance level of 0.05. We chose this test, since the mutation score values extracted from the subjects' test and source codes do not follow a normal distribution. Wilcoxon rank sum test is a non-parametric alternative to paired t-test [42]. When the design of the experiment involves one factor, two treatments, and a paired comparison between them, a Wilcoxon rank sum test can be used.

4 RESULTS

4.1 Dataset Reduction

We had 24 participants who attended our experiment [40]. Unfortunately, due to compile time errors and runtime errors in three participants ITLD task's source codes, and lack of test code in one participant's ITLD task, we were not able to calculate metrics for those participants. To avoid any false statistics, we removed those participants' data from our analysis. Furthermore, we could not compute the mutation score for some participants' data, since all existing unit test cases did not execute and pass. We fixed minor issues (correction of method and constructor call) in eight participants' test cases to pass those, and executed mutation score calculation tool (Judy) on participants' source codes. The final number of implemented tasks whose mutation score could be measured is presented in the next subsection.

4.2 Descriptive Statistics

Table 2 presents the descriptive statistics of mutation score for the tasks implemented in both treatments. We observe that subjects wrote more effective unit tests in detecting faults (injected into the source code) in TDD, i.e., 84% in median mutation score and 71% in mean mutation score, compared to ITLD, i.e. 62% in median and 55% in mean. Table 3 shows that the number of participants are not equal in both treatments due to the problems encountered on source codes and test code execution. Therefore, the mutation score values cannot be compared per subject. To make a paired comparison between the mutation scores of different treatments, we chose the participants whose mutation score could be measured in both ITLD and TDD tasks and presented the descriptive statistics for those in Table 3. We came up with 10 common participants' data, for which we could measure mutation score for both ITLD and TDD tasks. We observe that, TDD scores better in terms of mutation score, i.e. higher median and mean values, and lower standard deviation. This indicates that the unit test cases written in TDD could be able to detect more faults/defects than the unit test cases written in ITLD. To support our claim, we conducted statistical tests on data and reported our findings in the next subsection. The box plot in Figure 1 more clearly present that TDD has a higher mutation score than ITLD. This box plot also shows a greater variability of mutation scores in the ITLD task.

Table 2: Descriptive Statistics of Mutation Scores

	#Part.	Min.	Median	Mean	Max.	Var.
ITLD	13	2.0	62.0	54.7	83.0	27.9
TDD	18	6.0	84.0	70.6	93.0	28.4

Table 3: Descriptive Statistics of Mutation Scores for Pairwise Comparisons

	#Part.	Min.	Median	Mean	Max.	Var.
ITLD	10	17	67	57.5	83	26.17
TDD	10	56	87.5	83.5	93	10.69

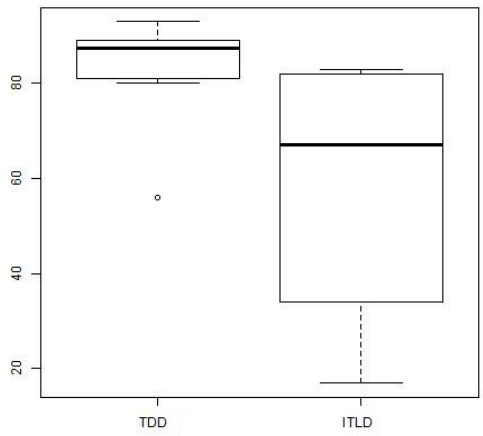
4.3 Hypothesis Testing

Wilcoxon rank sum test results between ITLD and TDD tasks on mutation score show that **there is a significant difference between the effectiveness of unit tests measured in two treatments** ($p - value = 0.04$). Higher mutation score shows higher defect detection ability, and hence in our study, test cases written in TDD have more fault detection abilities than test cases written in ITLD.

5 INTERPRETATION

Our results differ from the prior studies [11, 12, 26]. This difference can be a result of task assignment to the treatments. In the previous three studies, authors divided the participants into equal groups and assigned the same object for both the treatments (TDD and TLD). On the other hand, in our experiment [40], we assigned different objects for the treatments: Mars Rover for ITLD and Bowling Score Keeper for TDD. Moreover, the same group of participants first performed the task in ITLD and then the other task in TDD. In terms of mutation score [11, 12], the previous results show higher mutation scores in both treatments (on average 81,90% for TDD and 83,29% for TLD) compared to our study (on average 83.5% for TDD

Figure 1: Box plots for mutation score indicator in ITLD and TDD



and 57.5% for ITLD). Madeyski [26] was expecting a positive effect of test first approach in mutation score indicator. He explained that his study's results are likely to be affected by the difficulty of test-first technique and pre-existing difference in subjects. In our experiment [40], we avoided the pre-existing difference in subjects by generating a heterogeneous sample who applied both treatments sequentially. We also avoided the difficulty of TDD by adding a two-day training on TDD with hands-on exercises prior to the experimentation [40].

5.1 Coverage Analysis for the Thoroughness of Unit Tests

In addition to mutation score, earlier studies also measured code coverage of unit tests between the two development approaches to evaluate the thoroughness and effectiveness of unit tests [11, 12, 26]. To compare our study with these earlier studies, we also measured code coverage and evaluated the differences in terms of branch and method coverages of test suites implemented during TDD and ITLD using non-parametric statistical tests. Branch coverage refers to the conditional statements and the ratio of covered branches in the conditions over the total number of branches in the source code. Method coverage, on the other hand, refers to the execution of non-abstract methods (contains at least one instruction) during testing.

We define the null hypotheses for the two coverage metrics to evaluate TDD's effect on unit test thoroughness:

$H_0\mu(BRCOV)_{TDD} = (BRCOV)_{ITLD}$**:** There is no difference in the branch coverage (BRCov) between TDD and ITLD.

$H_0\mu(MTCOV)_{TDD} = (MTCOV)_{ITLD}$**:** There is no difference in the method coverage (MTCov) between TDD and ITLD.

Regarding code coverage, we applied Wilcoxon rank sum test at project level data, by aggregating class level metrics into project level. We chose this test because the data sample does not follow a normal distribution. For instance, for branch coverage metric, we calculated the mean, median, variance, minimum and maximum of branch coverage values obtained from classes in each project and created a new dataset. Then, we compared the median branch coverage and method coverage between subjects using Wilcoxon rank sum test.

The descriptive statistics of all coverage metrics calculated at project level are presented in Table 4. In Table 4, we have two rows for development methods (ITLD and TDD). Columns represent the metrics, (BRCov = branches covered, MTCov = methods covered) in percent.

From Table 4, we see that there are differences between ITLD and TDD with respect to branch coverage. On average, 37% of branches are covered by unit tests during ITLD, while 60% of branches are covered during TDD. The differences are even higher in quantity at median level. On the contrary, we can see more methods are covered by unit tests during TDD.

Statistical tests reported in Table 5 show us that the development approach does have an effect on code coverage of unit tests. The results indicate that TDD has a positive effect on branch coverage. The positive effects of TDD on branch coverage could be explained with the inherent properties of TDD. In TDD, developers have to write test cases first, and then implement the code that satisfies the conditions of test cases. This practice would result in an increase in the branch coverage of test cases. On the other hand, in ITLD, developers write and focus on production code first, and according to our observation, they may pay less attention to test case implementation.

Test cases written in ITLD cover significantly more methods than test cases written in TDD. This could possibly be related to developers' testing attitudes. While writing tests in ITLD, developers create unit test cases that check the functionality of functions in the production code (usually implemented as methods in object-oriented programing). Thus, we observe an increase in the method coverage. On the other hand, in TDD developers pay more attention to writing test cases for user stories, rather than addressing functions directly.

Our findings in this industrial setting are different from those reported in the previous experiments. In terms of code coverage Causevic et al. [11, 12] and Madeyski [26] compared only branch coverage between test-last and test-first development in academic experiments. Their findings did not reveal any difference in terms of branch coverage. Causevic et al. [11, 12] also used additional metric to evaluate test case quality by executing test cases of one participant on other participants' source code. They did not find any distinction in test case quality of both TDD and TLD.

5.2 Analysis on the Subjects' Codes

To further understand the coverage results, we examined the source codes of a sample of subjects. We selected four subjects based on the code coverage values: The first subject has higher method coverage-higher branch coverage, whereas the second subject has higher method coverage-lower branch coverage. The third subject has lower method coverage and higher branch coverage, and the fourth has lower scores in both method and branch coverage.

Table 4: Descriptive Statistics for Code Coverage

		BRCov	MTCov
ITLD	Mean	37.5	79.1
	Median	32.5	87.9
	Variance	33.8	23.8
	Min.	0	11.1
	Max.	96.4	100
TDD	Mean	59,8	61,4
	Median	89.2	70.7
	Variance	45.6	30.8
	Min.	0	0
	Max.	100	100

Table 5: Wilcoxon Test Results for Code Coverage

Metric	p-value	Hypothesis
$Median_{BRCov}$	0.033	rejected
$Median_{MTCov}$	0.045	rejected

After examining the source code of these subjects, we confirm that in TDD, subjects seem to pay more attention to test different branches compared to testing the whole functions/methods at once. We found several methods in the subjects' codes without any test cases associated with them. However, those methods that are associated with some unit tests are written to cover most of the branches/paths in the methods. Our examinations on source codes written during ITLD, on the other hand, show us that developers were more focused on implementing a small functionality in a single method, and hence, method coverage is higher in this approach. We also found that most of the test cases were written to verify happy/positive scenarios only. Developers pay less attention to verify negative/false conditions that leads to fewer branch coverage but higher method coverage.

6 THREATS TO VALIDITY

We consider threats to validity of results based on the checklist presented in Wohlin et al. [42], some of those threats are listed below.

Conclusion validity highlights those threats, which can direct to have wrong conclusions about the relationship between treatments and the results of the experiment [42]. Violated assumptions of statistical tests are concerned with assumption for tests. To avoid such threats, we used a non-parametric test, Wilcoxon rank sum test, that does not force any distributional assumption on two samples. Reliability of measure is another threat to validity of results [42]. To avoid this threat, we supported mutation score measure by also examining code coverage to answer our research question. Reliability of treatment implementation indicates the risk of having different implementations by different participant or in different occasions [42]. We avoided this risk by providing standardized implementation environments to all participants.

Internal validity concerns the results observed in the study and the true causes of those results. For example, results might be related to the independent variables or due to the fact that some other factors are involved [26]. Results are highly fragile to maturation validity threat. The knowledge of testing in ITLD, and repeated testing in TDD can cause this threat, as the subjects knew about the test, they could behave differently each time [42]. We avoided this threat by hiding the effectiveness and thoroughness of the tests written on the first day to the participants. The subjects were in a training during both ITLD and TDD [40] and they had to write coding and testing in different orders in the two treatments. Therefore, we believe this threat did not exist in this experiment.

Validity threat of selection might exist in this experiment as the subject volunteered to attend the training and experimentation, and according to Wohlin et al. [42], volunteers are more suited for a new task as they are generally more motivated, compared to a whole population. Unfortunately, in an industrial setting, it was not possible to create a random sample for such an experimentation and we had to prioritize the managers' requests while forming the participants.

The tasks specified in this experiment were considered as similar complexity by the subjects [40], and we also analyzed the size and complexity of both tasks' implementations in terms of code metrics and number of assertions written per task. We are still conducting additional analysis regarding this, and we believe that although the two tasks share similar complexity, there might be other intrinsic factors affecting the way the subjects implement these tasks. Nevertheless, more analysis is needed to finalize our claims by comparing different tasks.

Construct validity is about the relationship of theory or concept and the results of the experiment, as described by Madeyski [26], to which extent the measure reflects the theory or concept correctly. Experiments can be misleading if we use a single type of observation or measure [42]. We could avoid this measurement bias by using multiple measures for unit test effectiveness. The existing studies [4, 13, 23, 24] point out that unit test effectiveness is more associated with mutation score indicator compared to coverage. Therefore, we used mutation score as our construct. To avoid potential threats to the construct validity (measurement bias), we also analyzed code coverage of unit tests implemented by practitioners.

We minimized another risk to the conclusion validity by hiding information about our hypotheses from the subjects of the experiment. Fear of being evaluated can cause threat of "evaluation apprehension", which can motivate participant to forge the results of experiment [42]. We clearly mentioned that during this experiment subjects would not be evaluated on the basis of their performance in TDD and ITLD. We also measured the process conformance of subjects during the experiments through a set of tools, and hence, we made sure that subjects followed TDD and ITLD as much as possible. The detailed findings on process conformance were further discussed in [17].

External validity highlights the concerns which can limit the capacity of generalization of study findings, from a sample population to larger population [42]. Our empirical analyses conceptually replicated earlier studies conducted in academia, and therefore we assessed the generalizability of findings in industry. Furthermore, the sample population attended our training consisted of junior to

senior programmers in a large software organization with no prior hands-on TDD practice [40]. Therefore our study findings could also generalized to such population.

7 CONCLUSIONS

In this study, we analyse the impact of TDD approach on the unit test effectiveness in terms of tests' faulty detection capabilities (mutation score) and thoroughness of tests (code coverage) in the context of an industrial experiment. Using the same context and experimental setup of a prior industry experiment [40], we have defined new research questions and hypotheses, new measures and analysis models to analyse a different phenomenon stated by earlier academic experiments [11, 12, 26]. Our results contradict with these academic experiments in terms of mutation score. We have found that TDD improves the mutation score of unit tests compared to ITLD. We have also found that different coverage values depict different scenarios: TDD helps the subjects write refined test cases that cover more branches, whereas ITLD triggers writing unit tests that cover more methods in the source code. So depending on the coverage metric, the results reveal different conclusions.

As a future work, we would like to validate these findings in a larger context with more participants. We also plan to analyse the type of unit tests (happy versus sad paths) and their relations with the development approaches.

ACKNOWLEDGMENTS

This work has partially been supported by Istanbul Technical University Scientific Research Projects (MGA-2017-40712), and the Academy of Finland (Decision No. 278354).

REFERENCES

[1] 2014. EclEmma - Java Code Coverage for Eclipse. http://www.eclemma.org/
[2] 2014. JaCoCo - Coverage Counter. http://www.eclemma.org/jacoco/trunk/doc/counters.html
[3] 2015. FindBugs Eclipse Plugin. https://marketplace.eclipse.org/content/findbugs-eclipse-plugin
[4] J.H. Andrews, L.C. Briand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *27th International Conference on Software Engineering (ICSE)*. 402âĂŞ411.
[5] M.F. Aniche, G.A. Oliva, and M. A. Gerosa. 2013. What Do the Asserts in a Unit Test Tell Us about Code Quality? A Study on Open Source and Industrial Projects. In *17th European Conference on Software Maintenance and Reengineering*. 111–120.
[6] V.R. Basili. 1992. *Software modeling and measurement: the Goal/ Question/ Metric paradigm*. Technical Report. University of Maryland at College Park.
[7] K. Beck. 2002. *Test Driven Development: By Example*. Addison Wesley, Longman.
[8] T. Bhat and N. Nagappan. 2006. Evaluating the Efficacy of Test-Driven DevelomentâĂŤ Industrial Case Studies. In *2006 ACM/IEEE international symposium on Empirical software engineering (ISESE '06)*. 356–363.
[9] David Bowes, Tracy Hall, Jean Petrić, Thomas Shippey, and Burak Turhan. 2017. How Good Are My Tests?. In *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics (WETSoM '17)*. IEEE Press, Piscataway, NJ, USA, 9–14. https://doi.org/10.1109/WETSoM.2017..2
[10] A. Causevic, D. Sundmark, and S. Punnekkat. 2011. Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review. In *4th IEEE International Conference on Software Testing, Verification and Validation*. 337âĂŞ346.
[11] A. Causevic, D. Sundmark, and S. Punnekkat. 2012. Quality of testing in test driven development. In *Eighth International Conference on the Quality of Information and Communications Technology (QUATIC)*. 266âĂŞ271.
[12] A. Causevic, D. Sundmark, and S. Punnekkat. 2012. Test case quality in test driven development: A study design and a pilot experiment. In *16th International Conference on Evaluation & Assessment in Software Engineering (EASE)*. 223âĂŞ227.
[13] MickaĂńl Delahaye and Lydie du Bousquet. 2013. A Comparison of Mutation Analysis Tools for Java. In *Proceedings of the International Symposium on the Physical and Failure Analysis of Integrated Circuits, IPFA*. 187–195.
[14] M. E. Delamaro and J. Offutt. 2014. Assessing the Influence of Multiple Test Case Selection on Mutation Experiments. In *IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. 171–175.
[15] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok. 2001. Refactoring Test Code. In *2nd International Conference on Extreme Programming and Flexible Processes (XP)*. 92–95.
[16] P.G. Frankl, S.N. Weiss, and C. Hu. 1997. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software* 38, 3 (1997), 235âĂŞ253.
[17] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo. 2017. A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last? *IEEE Transactions on Software Engineering* 43, 7 (July 2017), 597–614. https://doi.org/10.1109/TSE.2016.2616877
[18] D. Fucci and B. Turhan. 2013. On the role of tests in test-driven development: a differentiated and partial replication. *Empirical Software Engineering* 19, 2 (2013), 1âĂŞ26.
[19] R. Gopinath, C. Jensen, and A Groce. 2014. Code coverage for suite evaluation by developers. In *36th International Conference on Software Engineering (ICSE)*. 72âĂŞ82.
[20] M. Greiler, A. van Deursen, and M. A. Storey. 2013. Automated Detection of Test Fixture Strategies and Smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 322–331. https://doi.org/10.1109/ICST.2013.45
[21] L. Huang and M. Holcombe. 2009. Empirical investigation towards the effectiveness of Test First programming. *Information and Software Technology* 51, 1 (2009), 182âĂŞ194.
[22] A. Hunt and D. Thomas. 2003. *Pragmatic Unit Testing in Java with Junit* (1st edition ed.). The Pragmatic Programmers.
[23] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 435–445. https://doi.org/10.1145/2568225.2568271
[24] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sept 2011), 649–678. https://doi.org/10.1109/TSE.2010.62
[25] T. Kaczanowski. 2012. *Practical Unit Testing with TestNG and Mockito* (1st edition ed.). Eigenverl. des Verf.
[26] L. Madeyski. 2010. The impact of Test-First programming on branch coverage and mutation score indicator of unit tests: An experiment. *Information and Software Technology* 52, 2 (2010), 169âĂŞ184.
[27] L. Madeyski and M. Kawalerowicz. 2013. Continuous test-driven development: A novel agile software development practice and supporting tool. In *8th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. 260–267.
[28] L. Madeyski and N. Radyk. 2010. Judy âĂŞ a mutation testing tool for Java. *IET Software* 4, 1 (2010), 32.
[29] Ayse Tosun Misirli, Hakan Erdogmus, Natalia Juristo, and Oscar Dieste. 2014. Topic Selection in Industry Experiments. In *Proceedings of the 2Nd International Workshop on Conducting Empirical Studies in Industry (CESI 2014)*. ACM, 25–30. https://doi.org/10.1145/2593690.2593691
[30] Hussan Munir, Misagh Moayyed, and Kai Petersen. 2014. Considering Rigor and Relevance when Evaluating Test Driven Development: A Systematic Review. *Inf. Softw. Technol.* 56, 4 (April 2014), 375–394. https://doi.org/10.1016/j.infsof.2014.01.002
[31] A. Jefferson Offutt and Ronald H. Untch. 2001. Mutation Testing for the New Century. Kluwer Academic Publishers, Norwell, MA, USA, Chapter Mutation 2000: Uniting the Orthogonal, 34–44. http://dl.acm.org/citation.cfm?id=571305.571314
[32] Rocco Oliveto, Andrea De Lucia, Abdallah Qusef, David Binkley, and Gabriele Bavota. 2012. An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM) (ICSM '12)*. IEEE Computer Society, Washington, DC, USA, 56–65. https://doi.org/10.1109/ICSM.2012.6405253
[33] Mauro Pezzè and Michal Young. 2007. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley.
[34] Yahya Rafique and Vojislav B. Misic. 2013. The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis. *IEEE Transactions on Software Engineering* 39, 6 (2013), 835–856. https://doi.org/10.1109/TSE.2012.28
[35] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. 2007. On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. *IEEE Transactions on Software Engineering* 33, 12 (Dec 2007), 800–817. https://doi.org/10.1109/TSE.2007.70745
[36] M. Roper. 1994. *Software testing*. McGraw-Hill Ryerson, Limited. https://books.google.com.tr/books?id=m6xQAAAAMAAJ
[37] W. R. Shadish, T. D. Cook, and Donald T. Campbell. 2001. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference* (2 ed.). Houghton Mifflin.

[38] Dr Muhammad Shahid, Suhaimi Ibrahim, and Mohd Mahrin. 2011. A Study on Test Coverage in Software Testing, In International Conference on Telecommunication Technology and Applications. *International Conference on Telecommunication Technology and Applications*.

[39] D. Tengeri, L. VidÃącs, ÃĄ. BeszÃIdes, J. JÃąsz, G. Balogh, B. Vancsics, and T. GyimÃşthy. 2016. Relating Code Coverage, Mutation Score and Test Suite Reducibility to Defect Density. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 174–179. https://doi.org/10.1109/ICSTW.2016.25

[40] Ayse Tosun, Oscar Dieste, Davide Fucci, Sira Vegas, Burak Turhan, Hakan Erdogmus, Adrian Santos, Markku Oivo, Kimmo Toro, Janne Jarvinen, and Natalia Juristo. 2016. An industry experiment on the effects of test-driven development on external quality and productivity. *Empirical Software Engineering* (2016). published online.

[41] Burak Turhan, Lucas Layman, Madeline Diep, Forest Shull, and Hakan Erdogmus. 2010. *How Effective is Test Driven Development?* OÃĈâĂŹReilly Press, Chapter Making Software: What Really Works, and Why We Believe It.

[42] Claes Wohlin, Per Runeson, Martin HÃĈÂűst, Magnus C. Ohlsson, and BjÃĈÂűrn Regnell. 2000. *Experimentation in Software Engineering*. Springer.

[43] Vahid Garousi YusifoÃ§lu, Yasaman Amannejad, and Aysu Betin Can. 2015. Software test-code engineering: A systematic mapping. *Information and Software Technology* 58 (2015), 123 – 147. https://doi.org/10.1016/j.infsof.2014.06.009