# Studying Test-driven Development and its Retainment Over a Six-month Time Span

Maria Teresa Baldassarre[a], Danilo Caivano[a], Davide Fucci[b], Natalia Juristo[c], Simone Romano[a,*], Giuseppe Scanniello[d], Burak Turhan[e,f]

[a]*University of Bari, Bari, Italy*
[b]*Blekinge Institute of Technology, Karlskrona, Sweden*
[c]*Universidad Politecnica de Madrid, Madrid, Spain*
[d]*University of Basilicata, Potenza, Italy*
[e]*Monash University, Melbourne, Australia*
[f]*University of Oulu, Oulu, Finland*

**Abstract**

Test-Driven Development (TDD) is an approach to agile software development, which is claimed to boost both external quality of software products and developers' productivity. The results about the claimed effects of TDD are inconclusive, therefore researchers have recommended taking a longitudinal perspective when studying TDD—*i.e.,* studying TDD over a time span. By following such a recommendation, we investigated: *(i)* the retainment of TDD over a time span of about six months, as well as *(ii)* the effects of TDD. To pursue our two-fold objective, we conducted a quantitative longitudinal cohort study with 30 novice developers (*i.e.,* third-year undergraduate students in Computer Science). We found that TDD is retained by developers over a time span of about six months. As for the effects of TDD, we observed that this development approach affects neither the external quality of software products nor developers' productivity. However, we observed that the participants applying TDD produced significantly more tests, with a higher fault-detection capability, than those using a non-TDD approach.

*Keywords:*

## 1. Introduction

Test-Driven Development (TDD) [1, 2] is a cyclic development approach where unit tests drive the incremental development of small pieces of function-

*Corresponding Author
*Email addresses:* `mariateresa.baldassarre@uniba.it` (Maria Teresa Baldassarre), `danilo.caivano@uniba.it` (Danilo Caivano), `davide.fucci@bth.se` (Davide Fucci), `natalia@fi.upm.es` (Natalia Juristo), `simone.romano@uniba.it` (Simone Romano), `giuseppe.scanniello@unibas.it` (Giuseppe Scanniello), `burak.turhan@monash.edu` (Burak Turhan)

Published in: Journal of Systems and Software.
This version of the article has been accepted for publication, after peer review and is subject to Elsevier terms of use.
The Version of Record is available online at: https://doi.org/10.1016/j.jss.2021.110937

ality [3]. Each development cycle starts with the writing of unit tests for an unimplemented piece of functionality. A cycle ends when unit tests pass as well as the existing regression test suite. An important role in the process underlying TDD is played by refactoring. It allows a TDD practitioner to improve the internal structure of the code, as well as its design, while preserving the external behavior of the code thanks to the safety net the existing regression test suite provides [2]. The end of a cycle allows a TDD practitioner to tackle a new piece of functionality, not yet implemented, so starting a new development cycle [1, 2]. Advocates of TDD recommend ending a development cycle in few minutes (five or ten minutes [4]) and keeping the rhythm as uniform as possible over time [1, 3]. The order with which unit tests interpose within the process underlying TDD—*i.e.,* the writing of a test precedes the one of the corresponding production code—is known as *test-first sequencing* (or also *test-first dynamic*) [5]. It is worth noting that test-first sequencing refers to just one central aspect of TDD [6]. That is, it does not capture the full nature of TDD [5]. Other central aspects that characterize the development process underlying TDD are: *granularity*, *uniformity*, and *refactoring effort* [5]. Granularity refers to the duration of the development cycles, while uniformity reflects how constant their duration is over time [5]. Finally, refactoring effort captures how much refactoring a TDD practitioner performs.

It is claimed that TDD leads to higher-quality products in terms of both external (*i.e.,* functional) and internal quality, while increasing developers' productivity [1]. These claimed benefits have encouraged some software companies to adopt TDD, while others are considering its adoption [7]. TDD has been assessed from a quantitative point of view (*e.g.,* [8, 9]) and according to a qualitative perspective (*e.g.,* [10, 11]). A number of primary studies, like experiments or case studies, have been conducted on TDD (*e.g.,* [8, 9, 12, 13, 14]). Their results, gathered and combined in a number of secondary studies (*e.g.,* [6, 15, 16, 17, 18, 19]), do not fully support the claimed benefits of TDD. Therefore, some researchers have recommended taking a longitudinal perspective when investigating such a development approach (*e.g.,* [16, 18, 20, 21])—*i.e.,* studying TDD over a time span. Nevertheless, only a few studies have taken such a perspective (*e.g.,* [22]).

Longitudinal studies[1] employ continuous or repeated measures to follow particular individuals over a time span of weeks, months, or even years [23]. In this paper, we present a study on TDD that takes a longitudinal perspective. In particular, we conducted a longitudinal cohort study in which our *cohort* consisted of 30 novice software developers of homogeneous experience who attended the same training regarding agile software development, including TDD. Thanks to that cohort, we collected separate measurements of the same constructs (*i.e.,* external quality, developers' productivity, number of tests written, fault-detection capability of tests written, test-first sequencing, granularity, uniformity, and

[1] There are three major types of longitudinal studies: *(i)* repeated cross-sectional studies; *(ii)* prospective studies (including cohort studies); and *(ii)* retrospective studies [23].

refactoring effort) about six months apart with the goal of understanding how well TDD can be applied over time, giving an indication of its *retainment*. To have a term of comparison, we contrasted TDD with a non-TDD approach (*e.g.,* iterative test-last, big-bang testing, or no testing at all), namely the approach that developers would normally follow. We refer to the non-TDD approach as *Your Way* (*i.e.,* YW). Our results indicate that novice developers retain TDD over a time span of about six months. Moreover, although we did not find any improvement, due to TDD, in the external quality of software products and developers' productivity, we observed that TDD allows creating larger test suites with a higher fault-detection capability.

This paper extends the one by Fucci *et al.* [24] as follows:

- We investigated the retainment of TDD with respect to four aspects that characterize the process underlying TDD: test-first sequencing, granularity, uniformity, and refactoring effort.
- Since Fucci *et al.* had found that TDD leads developers to write more tests, we studied whether writing more tests implies that the fault-detection capability of those tests is actually better. This was to strengthen the conclusions from Fucci *et al.*'s study. It is worth mentioning that we studied both effect and retainment of TDD with respect to fault-detection capability of written tests.
- We extended the inferential statistics by applying a second statistical model. This allowed us to mitigate as much as possible threats to the conclusion validity of the results shown in the paper of Fucci *et al.*.
- We extended the discussion of results. This extension is directly related to the several extensions and improvements of our data analysis which has allowed strengthening our conclusions.
- We improved the discussion of both related work and threats to validity.

**Paper structure.** In Section 2, we outline work related to ours. We present our study in Section 3. The obtained results are presented and discussed in Section 4 and Section 5, respectively. Final remarks conclude the paper.

## 2. Related Work

In this section, we describe the kinds of longitudinal studies researchers have conducted in the context of software engineering. Moreover, we show the results from secondary studies on the claimed effects of TDD—*i.e.,* better external quality and increased developer productivity—as well as those from long-term investigations on TDD.

### 2.1. Longitudinal studies in Software Engineering

The goal of a longitudinal study is to investigate *"how certain conditions change over time"* [25]. Therefore, the data collection happens over a time span and can require the researchers to be co-located with the case and context in which the phenomenon of interest takes place. In the context of software engineering, longitudinal studies are often associated with the case study methodology. For example, McLeod *et al.* [26] spent several hundred hours, over a time span of two years, at the case company. Here, the researcher attended meetings, observed and interviewed stakeholders to characterize software development as an emergent process.

In other cases, longitudinal studies are employed to observe the impact of a potentially disrupting event, such as the introduction of a new development practice. This scenario is similar to interrupted time series in quasi-experimental designs [27] in which, due to the lack of experimental manipulation, a specific event is used to identify the experimental groups. For example, Li *et al.* [28] studied the differences between Scrum and a waterfall-like approach to software development in a small company. The authors collected data from the development team for three years. In the first 17 months, the team used a waterfall-like approach before switching to Scrum for the following 20 months. The management decision to introduce Scrum represented the event allowing researchers to perform a *before-after* comparison of defects density and productivity. Such an extended time span avoided a biased comparison between an established process and an immature one. Salo and Abrahamson [29] followed the introduction of Software Process Improvement (SPI) techniques in the workflow of five agile projects over a time span of 18 months. They recorded the output of retrospective meetings, interviewed developers, and collected metrics from SPI tools. Vanhanen *et al.* [30] assessed the impact of introducing pair programming over a time span of two years with data collected through a survey with the developers.

A third kind of longitudinal study in software engineering retrospectively covers an extended time span by analyzing archival data. Harter *et al.* [31] analyzed the type of defects identified over time by the progressive introduction of SPI (Software Process Improvement) techniques in a firm and its subsequent CMMI (Capability Maturity Model Integration) improvements over a time span of 20 years. Given the availability of a large amount of versioned and timestamped data, longitudinal archival studies are usually performed in conjunction with software repositories mining studies. For example, Borges *et al.* [32] studied the growth over time in popularity of more than 2,000 GitHub repositories by identifying useful patterns for the maintainers.

### 2.2. Evidence regarding TDD

The effects of TDD on several outcomes, including the ones of interest for this study—*i.e.,* functional quality and productivity—is the topic of several empirical studies, summarized in Systematic Literature Reviews (SLRs) and meta-analyses (*e.g.,* [15, 17, 18, 19]). The SLR by Turhan *et al.* [17] includes 32 primary studies (*e.g.,* controlled experiments and case studies) published from

2000 to 2009. The gathered evidence shows a moderate effect in favor of TDD on functional quality while the evidence about productivity is inconclusive.[2] Bissi *et al.* [15] conducted an SLR that includes 27 primary studies published between 1999 and 2014. The results show an improvement of functional quality due to TDD while, as for productivity, they are inconclusive. Rafique and Misic [19] conducted a meta-analysis of 25 controlled experiments published in between 2000 and 2011. The authors observed a small effect in favor of TDD on functional quality while the results on productivity are inconclusive. Finally, Munir *et al.* [18] in their SLR classifies 41 primary studies published from 2000 to 2011 into four categories based on high/low rigor and high/low relevance. They found that in each category different conclusions could be drawn for both functional quality and productivity. This implies that, when looking at these studies as a unique set, the results are inconclusive. The authors concluded that more long-term studies are needed to better understand the effects of TDD.

An example of long-term investigation is the one by Marchenko *et al.* [33]. The authors conducted a three-year-long case study about the use of TDD at Nokia-Siemens Network. They observed and interviewed eight participants (one Scrum master, one product owner, and six developers) and then ran qualitative data analyses. The participants perceived TDD as important for the improvement of their code from a structural and functional perspective. Moreover, productivity increased due to the team improved confidence with the code base. The results show that TDD was not suitable for bug fixing, especially when bugs are difficult to reproduce (*e.g.,* when a specific environment setup is needed) or for quick experimentation due to the extra effort required for testing. The authors also reported some concerns regarding the lack of a solid architecture when applying TDD.

Beller *et al.* [34] executed a long-term study *in-the-wild* covering 594 open-source projects over the course of 2.5 years. They found that only 16 developers use TDD more than 20% of the time when making changes to their source code. Moreover, TDD was used in only 12% of the projects claiming to do so, and for the majority by experienced developers.

Borle *et al.* [35] conducted a retrospective analysis of (Java) projects, hosted on GitHub, that adopted TDD to some extent. The authors built sets of TDD projects that differed one another based on the extent to which TDD was adopted within these projects. The sets of TDD projects were then compared to control sets so as to determine whether TDD had a significant impact on the following characteristics: average commit velocity, number of bug-fixing commits, number of issues, usage of continuous integration, and number of pull requests. The results did not suggest any significant impact of TDD on the above-mentioned characteristics.

Latorre [22] studied the capability of 30 professional software developers of different seniority levels (junior, intermediate, and expert) to develop a complex software system by using TDD. The study targeted the *learnability* of TDD since

[2] It means that the results do not lead to a firm conclusion.

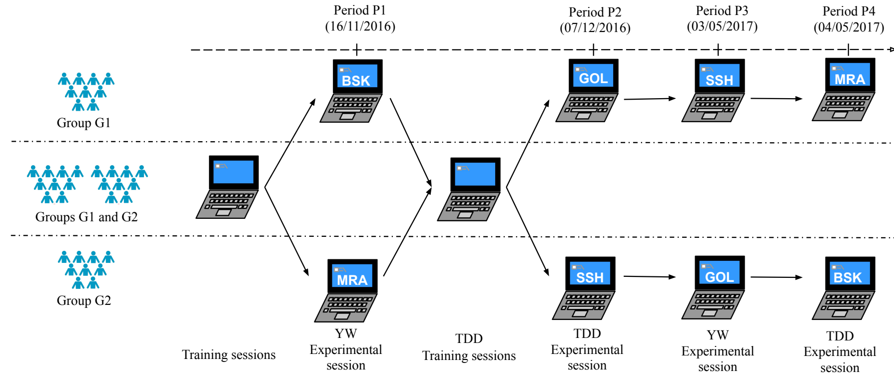



Figure 1: Study summary.

the participants did not know that technique before participating in the study. The longitudinal one-month study started after giving the developers, proficient in Java and unit testing, a tutorial on TDD. After only a short practice session, the participants were able to correctly apply TDD (*e.g.,* following the prescribed steps). They followed the TDD cycle between 80% and 90% of the time, but initially, their performance depended on experience. The seniors needed only few iterations, whereas intermediates and juniors needed more time to reach a high level of conformance to TDD. Experience had an impact on performance—when using TDD, only the experts were able to be as productive as they were when applying a traditional development methodology (measured during the initial development of the system). According to the junior participants, refactoring and design decision hindered their performance. Finally, experience did not have an impact on long-term functional quality. The results show that all participants delivered functionally correct software regardless of their seniority. Latorre [22] also provides initial evidence on the retainment of TDD. Six months after the study investigating the learnability of TDD, three developers, among those who had previously participated in that study, were asked to implement a new functionality. The results from this preliminary investigation suggest that developers retain TDD in terms of developers' performance and conformance to TDD.

Although the above-mentioned studies [22, 33, 34, 35] have taken a longitudinal perspective when studying TDD, none of them has mainly focused, as our longitudinal cohort study, on the retainment of TDD—although Latorre's study [22] provides initial evidence on the retainment of TDD, the main goal of this study was the learnability of TDD.

## 3. The Longitudinal Cohort Study

In Figure 1, we summarize the design of our longitudinal cohort study. The study participants were third-year undergraduate students in Computer Science.

They were divided into two groups G1 and G2 and were asked to take part in four experimental sessions. In any experimental session, each participant had to perform a development task by following either the YW approach or the TDD approach. Before the first experimental session, all the participants had practiced unit testing, iterative test-last development, and big-bang testing thanks to training sessions.

In the first experimental session, which was held in the period P1—a period is the time during which a treatment is applied [36]— the participants had to use the YW approach to perform development tasks. The two groups were asked to accomplish different development tasks—*i.e.,* G1 dealt with an experimental object (BSK, *i.e.,* Bowling Score Keeper) while G2 dealt with another one (MRA, Mars Rover API). We provide further details on the experimental objects in Section 3.3.

The participants in G1 and G2 learned and practiced (together) TDD between the first experimental session and the second one (held in the period P2). That is to say no one in the first experimental session knew TDD. It is easy to grasp that the first experimental session was introduced to have a baseline when the participants were not knowledgeable on TDD yet.

In the second experimental session, all the participants used TDD to perform the development tasks. The tasks assigned to G1 and G2 were different. In particular, G1 worked on GOL (Game Of Life), while G2 on SSH (SpreadSHeet).

After about six months (over two subsequent semesters of the same academic year, 2016-2017), we asked the participants to take part in the third and fourth experimental sessions, which were held in the periods P3 and P4 respectively. In particular, in the third experimental session, all the participants followed the YW approach when performing the development tasks, while in the fourth experimental session, they followed the TDD approach. We introduced the last two periods (*i.e.,* P3 and P4) some months after the first two periods to assess whether TDD was retained. It is worth mentioning that the tasks we asked the groups G1 and G2 to perform were different both in P3 (SSH and GOL, respectively) and P4 (MRA and BSK, respectively).

To plan and execute our cohort study, we followed the recommendations by Juristo and Moreno [37], and Wohlin *et al.* [38]. To report the planning and the execution of that study, we took into account the guidelines by Jedlitschka *et al.* [39].

### *3.1. Research Questions*

In our longitudinal cohort study, we have investigated the following main Research Questions (RQs):

**RQ1.** Do novice software developers retain TDD?

RQ1 was defined to study whether the retainment of TDD affects the application of YW, as well as the application of TDD, over approximately six months. We studied the retainment of TDD with respect to the following

constructs: *(i)* external quality of the implemented solutions, *(ii)* developers' productivity, *(iii)* number of tests written, *(iv)* fault-detection capability of tests written, *(v)* test-first sequencing, *(vi)* granularity, *(vii)* uniformity, and *(viii)* refactoring effort. We considered external quality of the implemented solutions and developers' productivity because TDD is claimed to improve external quality of the implemented solutions and increase developers' productivity [1]. We focused on the number of tests because past research has shown that TDD results in more tests [9]. However, having a test suite with more tests could not imply an increased fault-detection capability of that suite; therefore, we also studied the fault-detection capability of tests written. Finally, we considered test-first sequencing, granularity, uniformity, and refactoring effort because these are the four dimensions that characterize the development process underlying TDD [5]. It is worth recalling that the study of these four dimensions, together with the one on the fault-detection capability of tests written, is a new contribution with respect to our previous paper [8].

**RQ2.** Are there differences between TDD and YW?

We defined RQ2 to study if there are differences, due to TDD, in the *(i)* external quality of the implemented solutions, *(ii)* developers' productivity, *(iii)* number of tests written, and *(iv)* fault-detection capability of tests written. The reasons behind the study of these constructs have been explained above. We did not consider the four dimensions that characterize the development process underlying TDD because these dimensions do not characterize YW. With respect to the paper by Fucci *et al.* [8], we also investigated whether or not TDD leads to a higher fault-detection capability of tests written—*i.e.,* this is another new contribution of this paper.

### *3.2. Experimental Units*

The participants were third-year undergraduate students in Computer Science at the University of Bari (Italy). We sampled them by convenience among the students who attended the *Integration and Testing* course (first semester of the academic year 2016/2017). The program of this course included the following topics: unit testing, integration testing, SOLID principles, refactoring, big-bang testing, iterative test-last development, and TDD. During the course, the students participated in both face-to-face lessons and laboratory sessions. The students practiced unit testing, big-bang testing, iterative test-last development, and TDD through laboratory sessions and some homework was assigned too. Java was the programming language of the course, while JUnit and Eclipse were the testing framework and the Integrated Development Environment (IDE), respectively. Among the 53 students of the Integration and Testing course, 39 decided to take part in the study. The first two experimental sessions of our study were held during the Integration and Testing course.

Some students of the Integration and Testing course then attended the *Software Quality* course (second semester of the academic year 2016/2017). The

program of this course included the following topics: software quality (*i.e.,* internal, external, and in-use); ISO standards for software quality; software quality assessment, monitoring, and improvement; supporting tools for quality management (*e.g.,* SonarQube); and process control. The students enrolled in the Software Quality course were 45, 30 of them took part in the third and fourth experimental sessions. These 30 students had previously attended (and passed) the Integration and Testing course and had participated to the first two experimental sessions. This is to say that the intersection of the students who attended both courses (*i.e.,* Integration and Testing and then Software Quality) and participated in the longitudinal cohort study (*i.e.,* in any of the fourth experimental sessions) were 30—two females and 28 males.

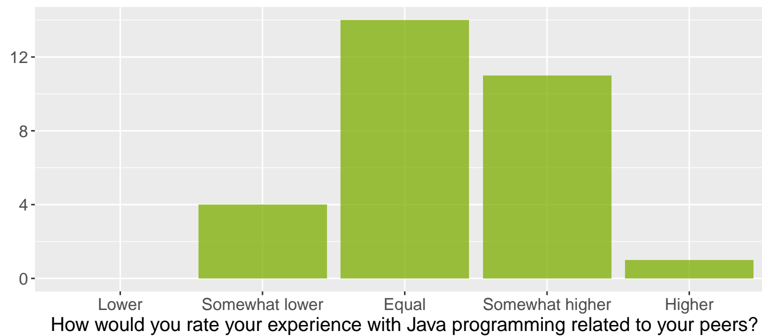
Before participating in the study, the students did not have a notion of TDD since their university curricula did not include courses on TDD. The participants had passed the exams of the following courses: *Procedural Programming*, *Object Oriented Programming*, *Software Engineering*, and *Databases*. Thanks to these courses, the participants had gained experience in C and Java programming. As shown in Figure 2a, most of the participants rated their experience as equal to or somewhat higher than that of their peers. Only four participants believed to be somewhat less expert than their peers. Figure 2b shows how the participants generally rated their experience with Java programming. Most of them stated to be neither experienced nor inexperienced. Only three participants judged themselves as inexperienced with Java programming. Summing up, the participants' characteristics can be considered homogeneous.

To encourage the students to participate in the study, we informed them that they would be rewarded with a bonus in the final mark of the Integration and Testing course. The students also knew that their participation would not affect their final mark (except for the bonus mentioned just before) and that the gathered data would be used only for research purposes. It is worth mentioning that students could not be paid for their participation in the study because this is forbidden in Italy (while rewarding them with a bonus in their final mark is allowed). Participation was voluntary in the sense that the students were not coerced to participate. All these choices were made to have motivated participants even if we were conscious that it could represent a threat to the internal validity of the results (see Section 5.3.1). We also informed the students that the collected data would have been treated confidentially and shared anonymously.

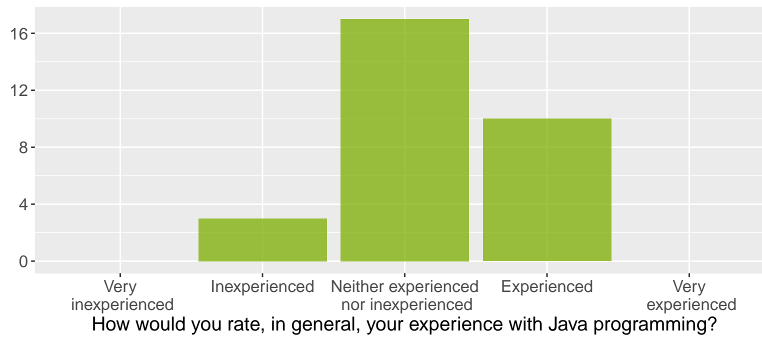
### *3.3. Experimental Materials*

The experimental objects were four code katas (*i.e.,* programming exercises used to practice a programming language or a development approach like TDD). A description of these code katas follows:

- **Bowling Score Keeper (BSK).** The goal of this kata is to develop an API for calculating the score of a bowling game made up of ten frames (plus potential bonus throws). The API allows: adding frames and bonus throws to a bowling game; identifying if a frame is a spare or a strike;




(a)




(b)

Figure 2: Barplots on the participants' experience with Java programming related to (a) their peers and (b) in general.

and computing the score of a single frame as well as the score of a bowling game.

- **Mars Rover API (MRA).** This kata aims at developing an API for moving a rover on a planet. The planet is represented as a grid of cells, which can contain obstacles that the rover cannot go through. The rover moves thanks to a sting made up of basic commands (*i.e.,* moving forward/backward and turning left/right). When the rover encounters an obstacle, it records that obstacle.

- **SpreadSHeet (SSH).** The goal of this kata is to develop an API for a basic spreadsheet. This API allows setting the content of a spreadsheet's cell and evaluating its content. A cell can contain strings, integers, references to other cells, and formulas (*e.g.,* string concatenations or arithmetic operations among integers).

- **Game Of Life (GOL).** This kata aims at developing an API for Conway's game of life. This game takes place on a square grid of cells. Each cell has

two possible states: alive or dead. The state of a cell evolves according to four rules. The API allows: initializing the grid; and determining the next state of a cell as well as the next state of the grid.

For each code kata (or experimental object, from here onwards), the experimental material included a template project for the Eclipse IDE, which contained stubs of the expected API signatures and an example JUnit test class. The code katas could be broken down into several features to implement; however, the description of the code katas was *coarser-grained*. That is, each code kata was presented as a whole without explicitly identifying the features to be implemented—in contrast to a *finer-grained* description of code katas in which the features to be implemented are described separately, thus they are explicitly identified (*e.g.,* each feature is numbered) [40]. To assess the features the participants implemented, the experimental material also included acceptance test suites. In particular, there was an acceptance test suite for each feature being implemented. The participants were not provided with these acceptance test suite because their purpose was the assessment of the implemented features. So, the acceptance test suites were used to quantify the external quality of the solutions implemented by the participants as well as their productivity.

Our decision to adopt code katas is because their use is common in empirical studies on TDD (*e.g.,* [5, 7, 8, 9, 40, 41]). Furthermore, this allowed us to use existing experimental materials (*e.g.,* from the studies by Fucci *et al.* [8] and Dieste *et al.* [41]). BSK and MRA have been also implied as experimental objects in several empirical studies (*e.g.,* [5, 7, 8, 40, 41]). Whereas for SSH and GOL, we created the experimental materials (*i.e.,* description of code katas, template projects, and acceptance test suites).

To gather some information on the participants (*e.g.,* gender, self-reported experience with Java programming, *etc.*), we defined an on-line pre-questionnaire we shared with the participants through Google Forms. We also created on-line post-questionnaires (by using Google Forms) to gather feedback after the participants had dealt with the code katas.

### *3.4. Tasks*

The participants were asked to carry out four implementation tasks, one for each experimental object (see Figure 1). To this end, each participant received the features to be implemented and the template project of a code kata, thus he/she implemented the features by filling the provided template project. No graphical user interface was required to implement the features.

### *3.5. Independent Variables*

To carry our an implementation task, the participants were asked to follow either TDD or YW (*i.e.,* the approach they preferred, excluding TDD). Therefore, **Approach** is the main independent variable (or also main or manipulated factor) of our study. This variable is nominal and assumes two possible values: *TDD* and *YW*. Since our study is longitudinal—*i.e.,* we collected data over time—we had a second main independent variable we named **Period**. It is a

nominal variable that represents the period during which each treatment (*i.e.,* TDD or YW) was applied. Therefore, this variable can assume the following values: *P1*, *P2*, *P3*, and *P4*. It is worth recalling that P1 and P3 correspond to the application of YW, while P2 and P4 correspond to that of TDD (see Figure 1 for details).

### *3.6. Dependent Variables*

To quantify external quality of the implemented solutions, developers productivity, number of tests written, we used the following dependent variables: **QLTY**, **PROD**, and **TEST**. We chose these dependent variables because they have been used in other empirical studies on TDD (*e.g.,* [5, 7, 8, 9]).

The variable QLTY measures the external quality of the solution to a code kata a participant implemented. This variable is defined as follows (*e.g.,* [8]):

$$QLTY = \frac{\sum_{i=1}^{\#TF} QLTY_i}{\#TF} * 100 \quad (1)$$

where #TF is the number of features a participant tackled, while $\mathrm{QLTY_i}$ is the external quality of *i*-th feature. A feature was tackled if at least one assert in the acceptance test suite (for that feature) passed. #TF is formally defined as follows:

$$\#TF = \sum_{i=1}^{n} \begin{cases} 1 & \#ASSERT_i(PASS) > 0 \\ 0 & otherwise \end{cases} \quad (2)$$

As for $\mathrm{QLTY_i}$, it is computed as the number of asserts passed for the i-th feature divided by the total number of asserts for that feature:

$$QLTY_i = \frac{\#ASSERT_i(PASS)}{\#ASSERT_i(ALL)} \quad (3)$$

QLTY assumes values in between 0 and 100. A value close to 0 means that the quality of the implemented solution is low, while a value close to 100 indicates high quality of the implemented solution.

As for the variable PROD, it measures the productivity of a participant when carrying out the implementation task. PROD is defined as follows (*e.g.,* [7]):

$$PROD = \frac{\#ASSERT(PASS)}{\#ASSERT(ALL)} * 100 \quad (4)$$

where #ASSERT(PASS) is the number of asserts passed in the acceptance test suites, while #ASSERT(ALL) is the total number of asserts in the acceptance test suites. PROD assumes values in between 0 and 100, where a value close to 0 means low productivity, while a value close to 100 means high productivity.

The variable TEST quantifies the number of unit tests a participant wrote. It is defined as the number of asserts in the test suite written by a participant when tackling the implementation task (*e.g.,* [8]). TEST assumes (integer) values in between 0 and $\infty$. A high value is desirable.

Table 1: Description of the mutation operators.

| Mutation operator | Description |
|---|---|
| AOR (Arithmetic Operator Replacement) | Replaces an arithmetic operator (*e.g.,* +) with another one (*e.g.,* −) |
| LOR (Logical Operator Replacement) | Replaces a logical operator (*e.g.,* &) with another one (*e.g.,* \|) |
| COR (Conditional Operator Replacement) | Replaces a conditional operator (*e.g.,* &&) with another one (*e.g.,* \|\|) |
| ROR (Relational Operator Replacement) | Replaces a relational operator (*e.g.,* >) with another one (*e.g.,* >=) |
| ORU (Operator Replacement Unary) | Replaces a unary operator (*e.g.,* ++) with another one (*e.g.,* −−) |
| LVR (Literal Value Replacement) | Replaces a literal value (*e.g.,* 0) with a default value (*e.g.,* 1) |
| STD (STatement Deletion) | Deletes a single statement (*e.g.,* a return statement) |

To quantify fault-detection capability of tests written, we leveraged mutation testing [42]. Given a program, mutation testing consists of automatically seeding artificial faults (*i.e.,* mutation faults) to generate mutants, each of which represents a faulty version of that program. Later, the test suite of the program is run against the mutants to determine the extent to which the test suite is capable of killing the generated mutants (*i.e.,* detecting the corresponding mutation faults). For each solution implemented by a participant, we seeded mutation faults into his/her production code (*i.e.,* we did not seed any fault in the test code) so generating mutants. To this end, we used the *Major* mutation framework [43]. We opted for this framework because it is robust [44], publicly available [43], and has been adopted in previous work (*e.g.,* [44, 45]). We applied the following mutation operators[3] to generate mutants: AOR, LOR, COR, ROR, ORU, LVR, and STD. A description of these operators is available in Table 1. This set of mutation operators is the same as that Papadakis *et al.* [44] used in their empirical investigation on the relationship between mutation and real faults. We ran the test suite the participant had written against the generated mutants so computing the MUTation score (**MUT**), namely the dependent variable we used to estimate fault-detection capability of tests written. MUT is computed as follows (*e.g.,* [42]):

$$MUT = \frac{\#MUTANTS(KILLED)}{\#MUTANTS(ALL)} * 100 \quad (5)$$

where #MUTANTS(KILLED) is the number of mutants the test suite killed, while #MUTANTS(ALL) is the total number of generated mutants. MUT assumes values in the interval [0,100]. The greater MUT, the better it is. In particular, it has been proven that MUT values close to 100 imply a higher fault-detection capability as compared with MUT values close to 0 [44]. This is why we leveraged mutation testing to estimate the fault-detection capability of written tests.

Besides the above-mentioned constructs, we investigated four constructs dealing with the development process underlying TDD, namely: test-first sequencing, granularity, uniformity, and refactoring effort. To quantify these constructs, we broke down the development process of participants applying TDD

[3] They alter a program by systematically applying a rule (*e.g.,* they replace the + arithmetic operator with the − one).

Table 2: Heuristics implemented in Besouro [47] to determine the type of cycles (the description of these heuristics is taken from Kou *et al.*' paper [46]).

| Cycle type | Sequence of actions |
|---|---|
| Test-first | Test creation → Test compilation error → Code editing →Test failure → Code editing → Test pass |
| | Test creation → Test compilation error → Code editing → Test pass |
| | Test creation → Code editing → Test failure → Code editing → Test pass |
| | Test creation → Code editing → Test pass |
| Refactoring | Test editing (file size changes ± 100 bytes) → Test pass |
| | Code editing (number of methods, or statements decrease) → Test pass |
| | Test editing AND Code editing → Test pass |
| Test addition | Test creation → Test pass |
| | Test creation → Test failure → Test editing → Test pass |
| Production | Code editing (number of methods unchanged, statements increase) → Test pass |
| | Code editing (number of methods increase, statements increase) → Test pass |
| | Code editing (size increases) → Test pass |
| Test-last | Code editing → Test creation → Test editing → Test pass |
| | Code editing → Test creation → Test editing → Test failure → Code editing → Test pass |
| Unknown | None of the above → Test pass |

into small cycles as done by Fucci *et al.* [5]. A cycle consists of a sequence of elementary actions and ends with a successful regression testing (*i.e.,* the regression test suite does not highlight regressions). Thanks to the heuristics devised by Kou *et al.* [46], it is possible to determine the type of each cycle (*e.g.,* test-first or refactoring). In Table 2, we report the heuristics we exploited to determine the type of the cycles when the participants applied TDD. The considered heuristics are implemented in *Besouro* [47].

The test-first sequencing construct indicates the prevalence of test-first sequencing within development processes underlying TDD. We quantified this construct by means of the **SEQ** dependent variable, which is defined as follows [5]:

$$SEQ = \frac{\#CYCLES(TEST\text{-}FIRST)}{\#CYCLES(ALL)} * 100 \tag{6}$$

where #CYCLES(TEST-FIRST) is the number of cycles classified as test-first by applying the heuristics in Table 2 when a participant followed TDD. #CYCLES(ALL) is, instead, the total number of cycles for that participant. The SEQ variable assumes values in between 0 and 100. The higher the value for this variable, the higher the amount of test-first cycles when a participant applied TDD.

Granularity refers to the extent to which the development process underlying TDD is fine-grained (or coarse-grained). To estimate this construct, we used the **GRA** dependent variable. It is computed as the median duration (expressed in minutes) of the development cycles a participant carried out [5]. This variable ranges between 0 and $\infty$. A low GRA value indicates that a participant mostly carried out short cycles—*i.e.,* his/her development process was fine-grained. On the other hand, a high GRA value indicates that a participant tended to carry out long cycles—*i.e.,* his/her development process was coarse-grained. The use of median to compute GRA, rather than mean, allows reducing the impact of outliers [5].

Uniformity indicates how uniform the development process underlying TDD is. This construct is quantified by means of the **UNI** variable, which is computed as the Median Absolute Deviation (MAD) of the cycle duration. This variable ranges between 0 and $\infty$. The lower the UNI value, the more uniform the cycles carried out by a participant were. A UNI value equal to 0 means that the cycles had mostly the same duration. The use of MAD to compute GRA, rather than standard deviation, allow reducing the sensitivity to outliers [5].

Refactoring effort indicates the prevalence of refactoring within the development process underlying TDD. We used the variable **REF** to estimate the refactoring effort construct. It is computed as follows [5]:

$$REF = \frac{\#CYCLES(REFACTORING)}{\#CYCLES(ALL)} * 100 \tag{7}$$

where #CYCLES(REFACTORING) is the number of cycles classified as refactoring (by using the heuristics in Table 2) when a participant followed TDD. The REF variable assumes values in between 0 and 100. The higher the value for this variable, the higher the refactoring effort of a participant when he/she applied TDD.

Since test-first sequencing, granularity, uniformity, and refactoring effort characterize the development process underlying TDD, we took into account these constructs only when the participants applied the TDD approach (*i.e.,* within periods P2 and P4).

### *3.7. Hypotheses*

We formulated and investigated the following parameterized null hypotheses:

**$\mathbf{HN1_X}$.** There is no statistically significant effect of Period with respect to X (*i.e.,* QLTY, PROD, TEST, MUT, SEQ, GRA, UNI, or REF).

**$\mathbf{HN2_X}$.** There is no statistically significant effect of Approach with respect to X (*i.e.,* QLTY, PROD, TEST, or MUT).

The alternative hypotheses were two-tailed (*i.e.,* whatever the independent variable was, we did not consider the direction of its effect). We defined $\mathrm{HN1_X}$ to study RQ1, while $\mathrm{HN2_X}$ to study RQ2.

### *3.8. Study Design*

In Table 3 (and Figure 1), we summarize the design of our cohort study. We randomly split the participants into G1 and G2, each of which had 15 participants. Whatever the group was, the participants experimented each treatment (*i.e.,* TDD or YW) twice. In particular, both groups experimented: *(i)* YW in the first period (*i.e.,* P1); *(ii)* TDD in the second period (*i.e.,* P2); *(iii)* YW in third period (*i.e.,* P3); and *(iv)* TDD in the last period (*i.e.,* P4). Accordingly, the design of our study is *repeated measures* (or *within-subjects*). In each period, the participants in G1 and G2 dealt with different experimental objects. For example, in P1, the participants in G1 dealt first with BSK, while those in G2 dealt first with MRA. At the end of the study, the participants had tackled each experimental object only once.

Table 3: Summary of the study design.

| | | Period P1 (16/11/2016) | P2 (07/12/2016) | P3 (03/05/2017) | P4 (04/05/2017) |
|---|---|---|---|---|---|
| Group | G1 | YW, BSK | TDD, GOL | YW, SSH | TDD, MRA |
| | G2 | YW, MRA | TDD, SSH | YW, GOL | TDD, BSK |

### *3.9. Procedure*

The Integration and Testing course—*i.e.,* the course in which the experimental sessions corresponding to P1 and P2 took place—started in October 2016. A mentioned before, we gathered some demographic information on the participant through an on-line pre-questionnaire at the beginning of that course.

The first experimental session (corresponding to the period P1) took place on November 16th, 2016. In particular, we administered the participants with the YW treatment. Between the beginning of the course and P1, the participants had never dealt with TDD. On the other hand, they had knowledge of unit testing, iterative test-last development, and big-bang testing. This is because the participants had taken part in two training sessions and carried out some homework (see Figure 1).

The first application of the TDD treatment took place on December 7th, 2016 (*i.e.,* P2). The participant learned TDD between P1 and P2. They had taken part in three training sessions on TDD and had completed some homework by using this development practice. Given the previous considerations, we can exclude that the knowledge of TDD had affected the application of the YW treatment in P1.

The participants applied the YW treatment again on May 3rd, 2017 (*i.e.,* P3), while the second application of the TDD treatment happened on May 4th, 2017 (*i.e.,* P4). Periods P3 and P4 took place in the Software Quality course. From P2 to P3 passed about six months—over this span of time, the participants followed the same university curricula courses. Although we asked the participants to use TDD, they knew TDD in P3. Therefore, we cannot exclude that the knowledge of TDD had affected the YW treatment in P3 somehow—*i.e.,* if the TDD retainment had affected the (second) application of YW or not. On the other hand, we assessed the retainment of TDD by asking the participants to use TDD (once again) in P4.

The execution of the longitudinal cohort study as additional teaching activities of the Integration and Testing and Software Quality courses somewhat imposed the time span we considered in that study. This is to say that a larger time span could not represent an feasible alternative in our case since the Software Quality course represented the only alternative to catch the largest number of students who had previously attended the Integration and Testing course. Moreover, the considered time span allowed us to counteract the following problems that are typical in longitudinal cohort studies: participants sometimes drop out, while others could loose the motivation to participate. It is worth recalling that the considered time span of about six months to study

the retainment of TDD is similar to the one by Latorre [22] (*i.e.,* the only study that has somehow investigated the retainment of TDD, see Section 2.2).

The implementation tasks were executed, under controlled conditions, in a laboratory at the University of Bari. In each period, the participants in G1 and G2 were assigned to the PCs in the laboratory—this laboratory was also used for the training sessions. When assigning the participants to the PCs, we alternated a participant in G1 with a participant in G2 to avoid that participants dealing with the same experimental object were close to one another. Such an arrangement aimed to avoid interactions among the participants. We also monitored them during the execution of the tasks.

The PCs in the laboratory were all equipped with the same hardware and software. On these PCs, we had installed Eclipse with the Besouro plugin [47]. Each participant received the description of code kata to be implemented, while, on the PC, he/she found the template project (for Eclipse) corresponding to that code kata. To carry out the implementation tasks, the participants had to use Java, JUnit, and Eclipse. At the beginning of a task, the participants launched Besouro within Eclipse, which started gathering data on their development process. These data allowed us to determine the type of development process of the participants who applied the TDD approach. At the end of each task, participants uploaded their implemented solutions on GitHub and then filled out a post-questionnaire to gather feedback on the executed task.

### *3.10. Analysis Procedure*

We analyzed the gathered data according to the following procedure:

1. **Descriptive Statistics and Exploratory Analyses.** We computed descriptive statistics (*i.e.,* mean, median and standard deviation), to summarize the distributions of the dependent variable values. To graphically summarize these distributions, we also boxplots.
2. **Inferential Statistics.** We used the Linear Mixed Model (LMM) analysis method to test the defined null hypotheses. Such a method is appropriate for the analysis of data from longitudinal studies [48]. LMMs are an extension of linear models containing both fixed and random effects. As for $\mathrm{HN1_X}$, we built, for the dependent variable X (*e.g.,* QLTY), the following LMM:

$$LMM1_X = X \sim Period + Group + Period : Group + (1|Participant) \quad (8)$$

where **Period** (*i.e.,* the main independent variable), **Group**—it assumes G1 or G2 as value—, and their interaction (*i.e.,* **Period:Group**) are the fixed effects. $\mathrm{LMM1_X}$ also includes a random effect, namely **Participant** (it identifies each participant, *e.g., 01* is the first participant). Modeling the participants with a random effect is customary in software engineering experiments [36]. When building $\mathrm{LMM1_X}$, we took into account Group because, according to the study design, it also represents the sequence (*i.e.,* the order in which the treatments are applied in combination with

the experimental objects). The sequence effect should be analyzed in repeated-measures designs (like ours) [36].

It is worth recalling that the periods P1 and P3 correspond to the application of the YW treatment, while P2 and P4 correspond to that of the TDD treatment. This implies that if $\text{LMM1}_\text{X}$ does not indicate a statistically significant effect of Period then there is not even a statistically significant effect of Approach. Accordingly, $\text{HN2}_\text{X}$ cannot be rejected. On the other hand, if there is a statistically significant effect of Period, the effect of Approach can be statistically significant. Therefore, only when $\text{LMM1}_\text{X}$ allowed us to reject $\text{HN1}_\text{X}$, we built a second LMM that included Approach (instead of Period) to test $\text{HN2}_\text{X}$:

$$LMM2_X = X \sim Approach{+}Group{+}Approach : Group{+}(1|Participant) \tag{9}$$

If $\text{LMM2}_\text{X}$ revealed a statistically significant effect of Approach, we rejected $\text{HN2}_\text{X}$. The use of $\text{LMM2}_\text{X}$ is new with respect to the paper by Fucci *et al.* [24].

LMMs have two assumptions that must be met: *(i)* the model residuals must be normally distributed; and *(ii)* their mean must be zero [36]. In case LMM assumptions are not met, transforming the dependent variable values is an option (*e.g.,* log-transformation) [36]. To check the normality of the residuals, we used the Shapiro-Wilk test (Shapiro test, from here onwards) [49].

Whatever the test of statistical significance was, we set the $\alpha$ value at 0.05—*i.e.,* we accepted a probability of 5% of committing Type-I error (this is customary in software engineering experiments).

## 4. Results

In the following of this section, we first present the results from the descriptive statistics and exploratory analyses and then we provide results from the inferential statistics.

### *4.1. Descriptive Statistics and Exploratory Analyses*

In Table 4, we report, for each dependent variable, mean, median, and Standard Deviation (SD) grouped by Period and Approach.

#### *4.1.1. QLTY—External Quality of Implemented Solutions*

In Figure 3, we report the boxplots for the QLTY variable grouped by Period or Approach. By looking at these boxplots, we can notice that there are not remarkable differences in the QLTY values when passing from one period to another one. In particular, if we compare the boxplots for P1 and P3—*i.e.,* same YW treatment but different experimental objects—, we can notice that they overlap and the median level in P1 is higher than that in P3 (76.76 vs. 71.28 as shown in Table 4). As for the periods P2 and P4—*i.e.,* same TDD treatment

Table 4: Descriptive statistics for each dependent variable grouped by Period and Approach.

| Variable | Statistic | Period (Approach) | | | | Approach | |
|---|---|---|---|---|---|---|---|
| | | P1 (YW) | P2 (TDD) | P3 (YW) | P4 (TDD) | YW | TDD |
| QLTY | Mean | 59.39 | 63.1 | 63.05 | 58.53 | 61.22 | 60.81 |
| | Median | 76.76 | 69.72 | 71.28 | 74.76 | 72.97 | 71.99 |
| | SD | 37.85 | 31.98 | 30.73 | 34.58 | 34.23 | 33.11 |
| PROD | Mean | 34.11 | 32.47 | 30.99 | 37.96 | 32.55 | 35.22 |
| | Median | 27.52 | 29.06 | 27.9 | 42.85 | 27.9 | 34.88 |
| | SD | 32.18 | 29.03 | 28.97 | 29.19 | 30.4 | 29 |
| TEST | Mean | 4.93 | 7.83 | 7.93 | 10.1 | 6.43 | 8.96 |
| | Median | 4 | 6.5 | 5 | 8.5 | 5 | 7 |
| | SD | 4.05 | 5.52 | 7.51 | 7.24 | 6.17 | 6.48 |
| MUT | Mean | 31.98 | 32.07 | 31.99 | 48.52 | 31.99 | 40.29 |
| | Median | 24.1 | 37.32 | 35.43 | 48.5 | 34.25 | 40.91 |
| | SD | 30.97 | 20.78 | 23.65 | 25.18 | 27.32 | 24.34 |
| SEQ | Mean | - | 27.91 | - | 22.3 | - | 25.1 |
| | Median | - | 19.21 | - | 19.09 | - | 19.09 |
| | SD | - | 25.73 | - | 20.27 | - | 23.1 |
| GRA | Mean | - | 10.29 | - | 4.68 | - | 7.49 |
| | Median | - | 4.26 | - | 2.5 | - | 3.03 |
| | SD | - | 16.33 | - | 6.47 | - | 12.62 |
| UNI | Mean | - | 5.76 | - | 2.82 | - | 4.29 |
| | Median | - | 3.22 | - | 1.74 | - | 2.33 |
| | SD | - | 6.5 | - | 4 | - | 5.54 |
| REF | Mean | - | 22.89 | - | 23.69 | - | 23.29 |
| | Median | - | 18.61 | - | 25.36 | - | 21.98 |
| | SD | - | 17.4 | - | 14.22 | - | 15.73 |

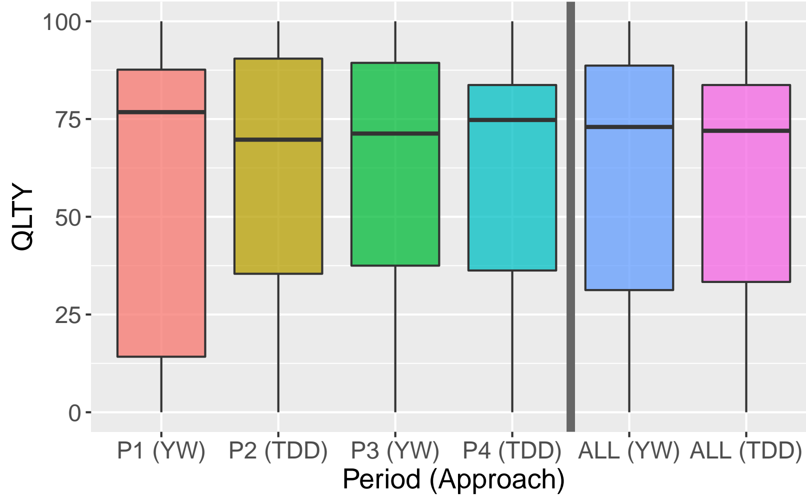



Figure 3: Boxplots for QLTY grouped by Period and Approach.

but different experimental objects—, the boxplots in Figure 3 overlap and the median level is higher in P4 (69.72 vs. 74.76). That is, it seems that, when the experimental objects are BSK and MRA (*i.e.,* in P1 and P4), the medians are higher. Therefore, the observed slight differences between P1 and P3, as well as between P2 and P4, seem to be due to the experimental objects. Summing up, there is no proof that TDD is not retained with respect to QLTY.

When comparing TDD and YW, the results (see Table 4 and Figure 3) do not suggest differences in the QLTY values (*e.g.,* on average, QLTY is equal to 60.81 for TDD, while it is equal to 61.22 for YW). This trend is confirmed when comparing P4 (TDD) with P1 (YW), as well as P2 (TDD) with P3 (YW),—*i.e.,* same experimental objects but different treatment. For instance, in P4 and P1, the mean values for QLTY are similar (58.53 vs. 59.39), although they applied either TDD or YW while tackling the same experimental objects. The comparison between P2 and P3 leads to a similar observation.

#### *4.1.2. PROD—Developers' Productivity*

By observing the boxplots for PROD in Figure 4, we can notice that there is not a huge difference in the PROD values among the periods. Indeed, when comparing P2 with P4—same TDD treatment but different experimental object—, we can observe that the boxplots overlap, although the median level for P4 is higher than for P2 (42.85 vs. 29.06). Such an improvement in the PROD values, when passing from P2 to P4, might be due to the TDD retainment. As for the comparison between P1 and P3—same YW treatment but different experimental object—, the boxplot for P1 is very similar to that for P3. That is, it seems that the TDD knowledge the participants had in P3 did not affect PROD.

Overall, it seems there is a slight difference in the PROD values between

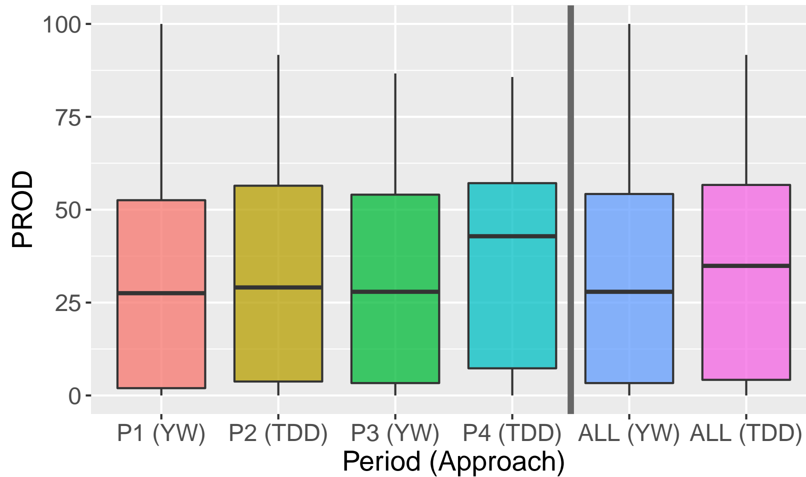



Figure 4: Boxplots for PROD grouped by Period and Approach.

TDD and YW (see Table 4 and Figure 4). This difference in favor of TDD; for instance, the mean PROD value for TDD is equal to 35.22, while that for YW is equal to 32.55. By comparing pairs of periods in which the same experimental objects are used (but different treatments are applied), we can notice that the PROD values in P4 (TDD) are better than those in P1 (YW); *e.g.,*, the mean values are equal to 37.96 and 32.47 in P4 and P1, respectively. Namely, it seems that the participants who applied TDD on BSK and MRA achieved PROD values better than the participants who applied YW on the same experimental objects. When comparing P2 (TDD) and P3 (YW)—the experimental objects were GOL and SSH—, it seems that there is no difference in the PROD values. For instance, the boxplots for P2 and P3 are very similar (see Figure 4).

#### *4.1.3. TEST—Number of Tests Written*

The boxplots for TEST are shown in Figure 5. By observing them, we can notice differences in the TEST values among the periods. In particular, if we compare the YW treatments in P1 and P3, we can notice that the boxplot for P3 is higher than that for P1. The descriptive statistics reported in Table 4 confirm that the TEST values are better in P3 than in P1—*e.g.,* the mean is equal to 7.93 for P3 and 4.93 for P1. This difference might be due to the knowledge the participants had in P3 on TDD. Namely, there might be a positive effect due to the TDD retainment. On the other hand, when comparing the TDD treatments in P2 and P4, the boxplots suggest a less pronounced difference in the TEST values. Indeed, the boxplots for P2 and P4 overlap, even though the median level for P4 is higher than that for P2 (8.5 vs. 6.5). Summing up, the results suggest that TDD can be retained with respect to TEST.

As for the the comparison between TDD and YW, it seems that the participants who followed TDD wrote more tests (see Table 4 and Figure 5). For

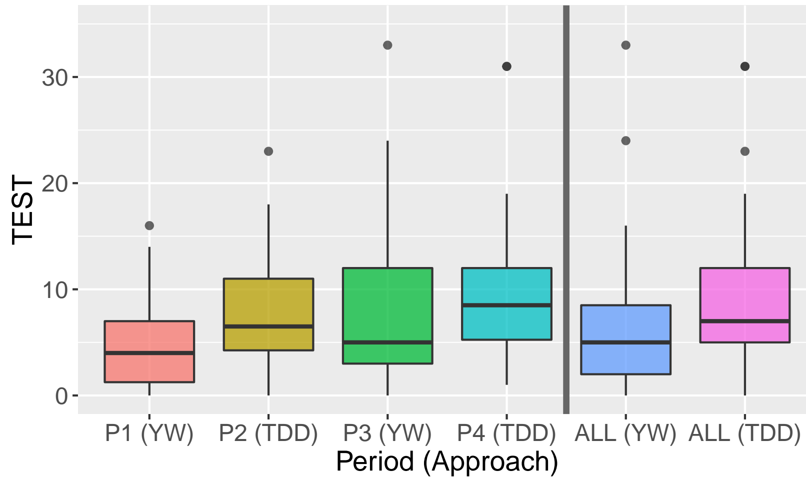



Figure 5: Boxplots for TEST grouped by Period and Approach.

instance, they achieved, on average, TEST values equal to 8.96 and 6.43 when following TDD and YW, respectively. If we consider only P1 and P4—same experimental object but different treatment—, we can observe a clear improvement in the TEST values in P4; *e.g.,* the mean values are 4.93 and 10.1, respectively. Namely, the participants who applied TDD in P4 seem to achieve higher TEST values than those who applied YW in P1 on the same experimental objects. Interestingly, the comparison between P2 (TDD) and P3 (YW) does not highlight a remarkable difference. Namely, it seems that the distributions of the TEST values for P2 (TDD) and P3 (YW) are quite similar (*e.g.,* the mean values are equal to 7.83 and 7.93, respectively), despite the application of either TDD (in P2) or YW (in P3) on the same experimental objects. This could indicate that the TDD retainment had influenced the participants who followed YW in P3.

*4.1.4. MUT—Fault-detection Capability of Tests Written*

By comparing the boxplots in Figure 6, we can observe that there are differences in the MUT values among the periods. If we consider only P1 and P3—the periods in which YW was applied—, the distributions of the MUT values look different. In particular, the boxplots for P1 and P3 overlap, but the latter is shorter and the median level is noticeably higher (24.1 vs. 35.43, see Table 4). However, the mean values for P1 and P3 are almost identical (31.98 vs. 31.99). This is to say that there is a high variation in the MUT values in P1 that reduces in P3—after the participants knew TDD. As for the comparison between the periods P2 and P4—those concerning TDD—, we can observe two different distributions in Figure 6. In particular, the distribution for P4 is noticeably higher than that for P2. That is, we can notice a clear improvement in P4 as far as the MUT values are concerned. The descriptive statistics in Table 4 remark this improvement (*e.g.,* the mean value passes from 32.07 in P2 to 48.52

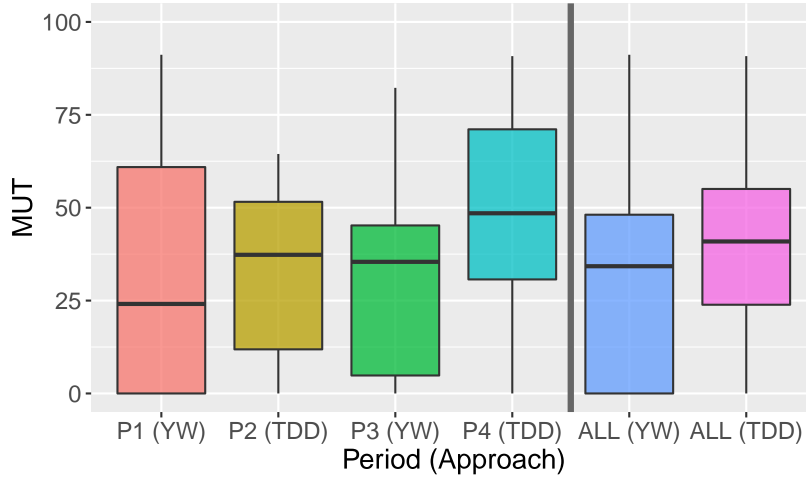



Figure 6: Boxplots for MUT grouped by Period and Approach.

in P4).

As for the overall boxplots of TDD and YW shown in Figure 6, we can notice that the boxplot for TDD is higher and shorter than that for YW. Such a difference can be also observed by looking at Table 4. For example, the mean values of MUT are 31.99 and 40.29 for YW and TDD, respectively. If we compare only P1 and P4—same experimental object but different treatment—, we can observe a clear improvement in the MUT values in P4; *e.g.,* the mean values are 31.98 and 48.52, respectively. As for the comparison between P2 and P3—same experimental object but different treatment—, the two distributions look similar; *e.g.,* the boxplots depicted in Figure 6 overlap. The descriptive statistics in Table 4 do not also highlight large differences in the MUT values between P2 and P3 (*e.g.,* the mean values are 32.07 and 31.99, respectively). Summing up, it seems that the differences in the MUT values reflect those in the TEST values. The results from the inferential statistics could strengthen such a conclusion.

#### *4.1.5. SEQ—Test-first Sequencing*

In Figure 7a, we graphically summarize the distributions of the SEQ values for P2 and P4—*i.e.,* the two periods in which the participants applied TDD. By looking at this figure, we can observe that the boxplots for P2 and P4 overlap but the latter is shorter—*i.e.,* there is less variation in the SEQ values in the second application of TDD (as also confirmed by the SD values, 25.73 vs. 20.27, reported in Table 4). Moreover, the median levels for P2 and P4 are very similar (19.21 vs. 19.09), even though, on average, the SEQ values for P2 are higher than those for P4 (27.91 vs. 22.3). Summing up, it seems that there is no huge difference in the application of TDD between P2 and P4 with respect to the SEQ so suggesting a retainment of TDD.

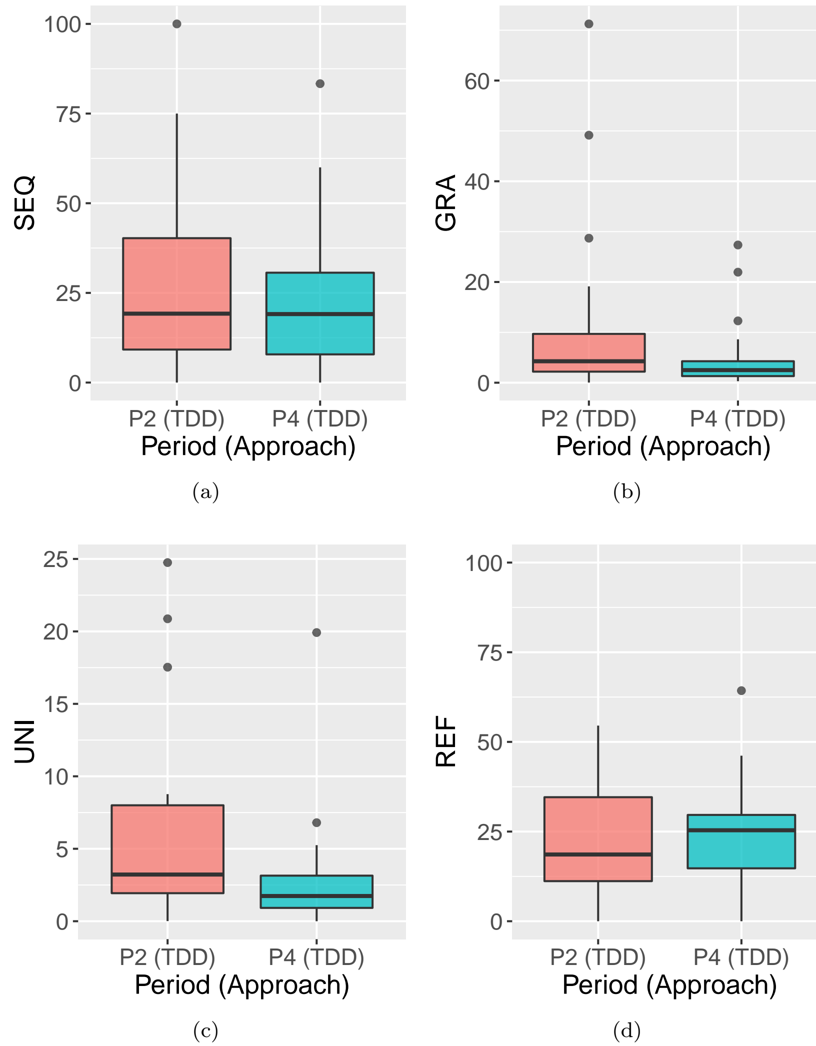



Figure 7: Boxplots for (a) SEQ, (b) GRA, (c) UNI, and (d) REF grouped by Period (only P2 and P4).

Table 5: Results (*i.e.,* p-values) from the inferential statistics. Note that $LMM2_X$ is built only when $LMM1_X$ indicates a statistically significant effect of Period. Moreover, $LMM2_X$ is not built for SEQ, GRA, UNI, and REF.

| Variable | $LMM1_X$ | | | $LMM2_X$ | | |
|---|---|---|---|---|---|---|
| | Period | Group | Period:Group | Approach | Group | Approach:Group |
| QLTY | 0.8837 | 0.6108 | <0.0001* | - | - | - |
| PROD | 0.7973 | 0.8225 | <0.0001* | - | - | - |
| TEST | 0.0002* | 0.0617 | 0.4632 | 0.0005* | 0.0617 | 0.7706 |
| MUT | 0.0017* | 0.734 | <0.0001* | 0.0454* | 0.734 | 0.0025* |
| SEQ | 0.4707 | 0.9864 | 0.5752 | - | - | - |
| GRA | 0.1992 | 0.2123 | 0.0581 | - | - | - |
| UNI | 0.021* | 0.4406 | 0.2211 | - | - | - |
| REF | 0.8581 | 0.5084 | 0.9611 | - | - | - |

* Statistically significant effect.

#### *4.1.6. GRA—Granularity*

The boxplots for GRA are depicted in Figure 7b. They seem to indicate a difference in the GRA values between P2 and P4. In particular, the boxplot for P4 is shorter and lower than that for P2—a lower GRA value is desirable (see Section 3.6). The descriptive statistics confirm this outcome; *e.g.,* the mean GRA values are equal to 4.68 and 10.29 for P4 and P2, respectively. That is, it seems that the participants retained TDD when passing from P2 to P4 and due to the TDD retainment there is an improvement in the GRA values in last period.

#### *4.1.7. UNI—Uniformity*

The distributions for UNI depicted in Figure 7c look different. In particular, the boxplot for P2 is higher and larger than the one for P4. The descriptive statistics in Table 4 seem to also highlight differences in the UNI values; *e.g.,* the mean values are equal to 5.76 and 2.82 for P2 and P4, respectively. In other words, it seems that the participants achieved better UNI values in P4—a lower UNI value is desirable (see Section 3.6)—and such an outcome could be due to a TDD retainment.

#### *4.1.8. REF—Refactoring Effort*

Regarding the distributions for REF, summarized in Figure 7d, we can observe that the distribution for P2 is quite similar to that for P4. Indeed, the boxplots for P2 and P4 overlap and the median level is higher in P4 (18.61 vs. 25.36). However, the REF values was, on average, similar: 22.89 in P2 and 23.69 in P4. Again, it seems the participants retained TDD with respect to REF.

### *4.2. Inferential Statistics*

In Table 5, we report the p-values from the LMM analysis methods. We highlight the effects that are statistically significant with the asterisk symbol.

#### 4.2.1. QLTY—External Quality of Implemented Solutions

The assumptions of $\text{LMM1}_{\text{QLTY}}$ were both met. The residuals of the built LMM were normally distributed (the Shapiro test returned a p-value equal to 0.1166) and their mean was equal to zero.

The LMM analysis method do not allow us to reject $\text{HN1}_{\text{QLTY}}$ because the p-value for Period is 0.8837 (see Table 5), namely the effect of Period is not statistically significant. This means that there is neither a deterioration nor an improvement in the observed time span for the QLTY variable. $\text{LMM1}_{\text{QLTY}}$ also indicates a statistically significant interaction between Group and Period, which is due to the effect of the experimental objects (*e.g.,* regardless of the treatment, the distributions for BSK are higher than those for GOL). As for the effect of Group, it is not statistically significant. We can therefore conclude that TDD can be retained in terms of external quality of software products.

Since $\text{LMM1}_{\text{QLTY}}$ does not indicate a statistically significant effect of Period, this implies that neither the effect of Approach is statistically significant (*i.e.,* there is no need to build $\text{LMM2}_{\text{QLTY}}$). Therefore, we cannot reject $\text{HN2}_{\text{QLTY}}$. This outcome seems to suggest that developing according to the TDD approach does not influence the external quality of software products.

#### 4.2.2. PROD—Developers' Productivity

The assumption of normality was not met for $\text{LMM1}_{\text{PROD}}$ since the Shapiro test returned a p-value equal to 0.0035. To meet this assumption, we needed a data transformation (square-root transformation, in particular). After transforming the data, the LMM assumptions were both satisfied—the residuals were normally distributed according to the Shapiro test (p-value=0.0524) and their mean was equal to zero.

As shown in Table 5, the effect of Period for $\text{LMM1}_{\text{PROD}}$ is not statistically significant (p-value=0.7973). Therefore, we cannot reject $\text{HN1}_{\text{PROD}}$. This outcome indicates that the participants can retain TDD as far as their productivity is concerned. $\text{LMM1}_{\text{PROD}}$ also includes a statistically significant effect, namely the one of Group:Period. Again, this is due to the effect of the experimental objects. Finally, there is no statistically significant effect of Group.

Since we cannot reject $\text{LMM1}_{\text{PROD}}$, neither $\text{HN2}_{\text{PROD}}$ can be rejected. This said, it seems that developers who follow either the TDD or YW approach exhibit a similar productivity.

#### 4.2.3. TEST—Number of Tests Written

We needed a data transformation for the variable TEST because the residuals for $\text{LMM1}_{\text{TEST}}$ were not normally distributed (the p-value the Shapiro test returned was $<0.0001$). Therefore, we applied a log transformation, which allowed us to meet both LMM assumptions. In particular, the Shapiro test returned a p-value equal to 0.0797, suggesting that the residuals followed a normal distribution. The mean of the residuals was equal to zero.

$\text{LMM1}_{\text{TEST}}$ shows a statistically significant effect of Period (p-value=0.0002). Therefore, we can reject $\text{HN1}_{\text{TEST}}$ and conclude that Period significantly affects

the number of tests the participants wrote. By looking at the boxplots in Figure 5, we can notice that this statistically significant effect is not due to a deterioration of TDD over time—the worst distribution of the TEST values can be observed in P1 while the best distribution can be observed in P4. We can therefore conclude that developers following TDD retain the ability to write unit tests. $LMM1_{TEST}$ does not indicate other statistically significant effects.

Since we found a statistically significant effect of Period, there can be a statistically significant effect of Approach. In this respect, the descriptive statistics and boxplots depict a clear difference in favor of TDD when comparing P1 and P4—same experimental objects. This difference is also noticeable when comparing, in general, TDD with YW. To determine whether or not there is a statistically significant effect of Approach, we built $LMM2_{TEST}$. Since the the residuals of $LMM2_{TEST}$ were not normally distributed (the p-value the Shapiro test returned was <0.0001), we performed a log-transformation for TEST. Thanks to this transformation, the assumptions of $LMM2_{TEST}$ were both met: the residuals followed a normal distribution (the p-value returned by the Shapiro test was 0.0562) and their mean was equal to zero.

As reported in Table 5, $LMM2_{TEST}$ only includes a statistically significant effect, namely that of Approach (p-value=0.0005). Therefore, we can reject $HN2_{TEST}$ and conclude that TDD significantly and positively affects the number of tests the participants wrote.

### *4.3. MUT—Fault-detection Capability of Tests Written*

The residuals of $LMM1_{MUT}$ were normally distributed (the Shapiro test returned a p-value equal to 0.5289) and their mean was equal to zero. Therefore, we did not need any data transformation for $LMM1_{MUT}$.

$LMM1_{MUT}$ includes two statistically significant effects: one for Period (p-value=0.0017) and one for Period:Group (p-value<0.0001). The p-value of Period allows rejecting $HN1_{MUT}$, thus recognizing that Period significantly affects the fault-detection capability of the written test suite. Moreover, we can observe an improvement of the MUT values in P4 with respect to any other period (*e.g.,* see Figure 6). Therefore, we can conclude that developers can retain their ability to write tests in terms of both number of written tests and fault-detection capability of these tests. As for the p-value of Period:Group, it suggests that the fault-detection capability of the written tests can depend on the development task at hand.

Since we could reject $HN1_{MUT}$, a statistically significant effect of Approach is possible. In this respect, $LMM2_{MUT}$ allows us to reject $HN2_{MUT}$ because the p-value of Approach is equal to 0.0454. That is, there is a statistically significant effect of Approach, in favor of TDD, on the fault-detection capability of the written test cases. This outcome suggests that TDD practitioners tend to write more tests than non-TDD ones and, moreover, the fault-detection capability of these tests is significantly better. Again, the fault-detection capability of the written tests seems to depend on the development task at hand.

It is worth mentioning that, before applying $\text{LMM2}_{\text{MUT}}$, we checked the LMM assumptions, which were both satisfied (*e.g.,* the Shapiro test returned a p-value equal to 0.247 indicating a normal distribution for the residuals).

#### *4.3.1. SEQ—Test-first Sequencing*

The assumption of normality was not satisfied for $\text{LMM1}_{\text{SEQ}}$ (the Shapiro test returned a p-value equal to 0.0005). To satisfy both LMM assumptions, we square-transformed the SEQ variable. After this data transformation, the residuals were normally distributed according to the Shapiro test (p-value=0.3846) and their mean was equal to zero.

By looking at the p-values in Table 5, we can notice that $\text{LMM1}_{\text{SEQ}}$ do not include a statistically significant effect for Period. This outcome suggests that developers can retain TDD with respect to the test-first sequencing. As for the other p-values, they indicate a statistically significant effect for neither Group nor Period:Group. The p-value for Period:Group seems to indicate that developers' ability of following the test-first dynamic is not affected by the development task (*i.e.,* the experimental object).

#### *4.3.2. GRA—Granularity*

To met the assumptions of $\text{LMM1}_{\text{GRA}}$, we had to apply a log-transformation. This is because the Shapiro test indicated a violation of the normality assumption of the residuals (p-value<0.0001). Thanks to the log-transformation, we satisfied the assumption of normality of the residuals (the p-value returned by the Shapiro test was 0.0568) as well as that concerning their mean.

As shown in Table 5, the effect of Period for $\text{LMM1}_{\text{GRA}}$ is not statistically significant although the boxplots in Figure 7b highlighted an improvement in P4 (with respect to P2). On average, the granularity of the development cycles was 4.26 and 2.5 minutes. These outcomes suggest that developers can retain their ability of following short cycles when applying the TDD approach. The p-values for Group and Period:Group did not highlight any statistically significant difference. Similarly to the test-first sequencing characteristic, it seems that the granularity of development cycles is not affected by the tasks.

#### *4.3.3. UNI—Uniformity*

We applied a log-transformation to meet the assumptions of $\text{LMM1}_{\text{UNI}}$. This is because the residuals were not normally distributed according to the Shapiro test (p-value<0.0001). By applying the log-transformation we met both LMM assumptions (in particular, the Shapiro test returned a p-value equal to 0.065).

The p-values in Table 5 indicate a statistically significant effect of Period (0.021). The distributions of the UNI values (*e.g.,* see Figure 7c) suggests that, in P4, the development cycles of the participants who applied TDD were more uniform as compared to P2. This is to say that developers can retain the TDD characteristic of carrying out uniform development cycles. Finally, there is a statistically significant effect for neither Group nor Period:Group—*i.e.,* it seems that the tasks do not influence the uniformity of development cycles.

#### 4.3.4. REF—Refactoring Effort

We did not need any data transformation for $\text{LMM1}_{\text{REF}}$ because the assumptions were both met. In particular, the residuals followed a normal distribution (the Shapiro test returned a p-value equal to 0.1134) and their mean was equal to zero.

The results, shown in Table 5, do not highlight any statistically significant effect. Concluding, it seems that the refactoring effort is retained when practicing TDD. Again, the tasks seem to not influence the refactoring effort.

## 5. Discussion

In this section, we first answer the RQs to delineate the main findings of our cohort study. We then discuss these findings and present their practical implications. Finally, we discuss the threats that may have affected the validity of these findings.

### 5.1. Answering Research Questions

We observed no deterioration, during the considered time span, in the external quality of the solutions our participants implemented (*i.e.,* QLTY), their productivity (*i.e.,* PROD), and number of tests they wrote (*i.e.,* TEST). Furthermore, the way in which the participants followed the process underlying TDD (*i.e.,* in terms of SEQ, GRA, UNI, and REF) did not deteriorate over time. On the other hand, we observed a significant improvement in the number of tests, which led to a better fault-detection capability of these tests (*i.e.,* MUT), after the participants had known and practiced TDD. The uniformity of the cycles enhanced with time as well. On the basis of these results, we can answer RQ1 as follows.

> **Developers retain TDD over about six months. In particular, while the external quality of software products and developers' productivity are neither deteriorated nor improved over that time span, the amount of written test increases as well as their fault-detection capability. Moreover, the way in which developers follow TDD remains constant in the considered time span, except for the uniformity of the process underlying TDD, which is more uniform over time.**

This outcome is perhaps not overly surprising, but evidence needs to be obtained through empirical studies to move from opinions and common sense to facts (*e.g.,* [50, 51]), as well as to have a first understanding of TDD retainment on several constructs.

As for the comparison between TDD and YW, we observed differences in favor of the former only when considering the amount of written tests (*i.e.,* TEST). Such a difference is also present when considering the fault-detection capability of the tests written (*i.e.,* MUT). Accordingly, we can answer RQ2 as follows.

**While TDD does not increase (or decrease) the external quality of software products and developers' productivity, it leads developers to create larger test suites with a higher fault-detection capability.**

*5.2. Overall Discussion and Future Research*

We show that developers retain TDD over a time span of about six months. This finding is in line with the preliminary empirical evidence gathered by Latorre [22] on the retainment of TDD—where he observed that, in a six-month time span, three developers retained TDD in terms of developers' performance and conformance to TDD. Based on the current empirical evidence on TDD, we can deduce that the investment in training new TDD practitioners is not squandered—it is preserved at least over a time span of six months. Furthermore, previous work has also shown that developers can correctly apply TDD after a short practical session only [22]. Accordingly, we can postulate that the investment in training new TDD practitioners is reasonable. The question that now arises is how long such an investment is preserved, *i.e.,* how long developers retain TDD. To answer this question, further longitudinal cohort studies are needed. Our study has, therefore, the merit to increase the body of knowledge on the retainment of TDD as well as to delineate new possible investigations on how long developers retain TDD.

Among the investigated constructs, we observed that the retainment of TDD is particularly noticeable in the amount of tests written since it increased after the participants had known and practiced TDD. This seems to suggest that TDD raises developers' awareness about the importance of writing unit tests; furthermore, these tests exhibit a higher fault-detection capability. Therefore, we advise instructors to teach TDD when training new unit testers.

As compared to those who follow a non-TDD development process, we also observe that developers practicing TDD, write more unit tests that have a better fault-detection capability as well. This finding is in line with that by Erdogmus *et al.* [9] so bringing further evidence that TDD has a positive effect on the number of written tests. Again, our findings go in the direction of increasing the body of knowledge on the effect of TDD.

Having more unit tests, with a higher fault-detection capability, should encourage software companies that value unit testing (*e.g.,* in order to create regression test suites for continuous integration) to adopt TDD. Possible benefits deriving from having many tests with high fault-detection capability could be early fault detection and facilitated comprehension of unfamiliar source code (*e.g.,* it has been showed that developers dealing with an unfamiliar code base look for examples of input/output values to better understand that code base [52]—unit tests contain such a kind of examples).

Our results do not highlight any improvement due to TDD, with respect to the external quality of software products and developers' productivity, so contributing to the null results in TDD research (*e.g.,* [8, 53]). However, unlike

previous studies, we observe that TDD has no effect even when the same individuals are tested again several months later, under similar conditions. Time did not reduce novice developers' performance when TDD was applied, hinting at the fact that they soon regained familiarity with this technique, similarly to what Latorre reported for the junior developers involved in his study [22]. Although carrying out cohort longitudinal studies—in particular, with several observations over a long time span—is difficult in Software Engineering (*e.g.,* controlling for maturation or keeping motivated the participants), we put forward the idea that we might not be looking long enough (rather than hard enough) for the claimed effects of TDD to become apparent. As a starting point towards this direction, we recommend longitudinal studies in academia capable of following students' careers over several years and thus achieving a good amount of control (*e.g.,* based on grades). We advise this kind of investigation very risky and difficult to conduct. However, our study seem to justify future research on this matter.

We observe neither a deterioration nor an improvement over time in the external quality of software products and developers' productivity. A possible cause for this finding is that, with the only exception of the uniformity of the process underlying TDD, the way in which the participants followed TDD (*i.e.,* test-first sequencing, granularity, and refactoring effort) remained constant in the considered time span. Past work has shown that (external) quality and productivity improvements are primarily positively associated with the granularity and uniformity of the process underlying TDD [5]. Therefore, it is possible that observing a significant difference in the uniformity of the process underlying TDD is not enough to show alone a significant difference in the external quality of software products and developers' productivity.

Finally, to bring further evidence on both retainment of TDD and its effects, we foster replications of our longitudinal cohort study. To this end, we made available on the web our laboratory package:

- `https://doi.org/10.6084/m9.figshare.6850013.v1`.[4]

*5.3. Threats to Validity*

To determine the threats that could affect the validity of our study, as well as its results, we followed the guidelines by Wohlin *et al.* [38]. Despite our effort to lessen or avoid as many threats as possible, some of them are unavoidable. This is because reducing or avoiding a kind of threat (*e.g.,* internal validity) may intensify or introduce another kind of threat (*e.g.,* external validity) [38]. Since we conducted the first cohort study investigating the theory of TDD retainment, we preferred to reduce threats to internal validity (*i.e.,* make sure that the

[4] This is the laboratory package of Fucci *et al.*'s study [24]—awarded as "Open Data Recognition" at ESEM 2018. In case of acceptance of the paper, we are going to update that laboratory package with the new data we gathered (*e.g.,* data on fault-detection capability of written tests), which are temporarily available on: `https://tinyurl.com/Raw-Data-Ext-ESEM`.

cause-effect relationships were correctly identified), rather than being in favor of external validity.

#### *5.3.1. Threats to Internal Validity*

This kind of threat concerns internal factors of our study that could have affected the results.

- **Selection.** The participants in our study were volunteers. This might threaten the validity of the results because volunteers might be more motivated than the overall population [38].
- **Diffusion or treatments imitations.** To prevent that participants exchanged information during the development tasks, at least two researchers monitored them. Moreover, the participants were assigned to each workstation in the laboratory alternating the experimental objects. We also prevented the diffusion of experimental materials by gathering them at the end of each task and asking the participants not to talk with their classmates about the tasks they had implemented. Despite our effort to lessen this threat, we cannot exclude its presence, *e.g.,* some participates might have exchanged information about the tasks outside the laboratory.
- **Resentful demoralization.** Some participants might not perform as good as they generally would since they might have received a less desirable treatment (or tasks). If this threat had existed in our study, it would have equally affected TDD and YW.
- **Maturation.** The control over participants was checked by making sure that the students attended the same courses between the first observation and the last one. This might affect the obtained results. Moreover, it seems that four participants did not launch Besouro at the beginning of the TDD sessions. To have all data paired, we did not take into account these participants in the analyses of SEQ, GRA, UNI, and REF. If the participants had launched Besouro, the results might have changed.

#### *5.3.2. Threats to Construct Validity*

They concern the relationship between theory and observation.

- **Mono-method bias.** We used a single measure for each investigated construct. This might affect the validity of the results if there was a measurement bias. To mitigate this threat, we used well-known measures [5, 7, 8, 9].
- **Hypotheses guessing.** Participants in an empirical study might guess the study goals and then behave according to their guesses. Although we did not disclose our study goals to the participants (*i.e.,* we did not tell them that they were involved in an empirical study, nor how it was planned and how assignments were distributed among participants), someone might have guessed the goals and changed their behavior accordingly.

- **Evaluation apprehension.** Some people are afraid of being evaluated. To mitigate this threat, we informed the participants that they would not be evaluated on the basis of their performance in the study.

- **Restricted generalizability across constructs.** We found that TDD positively affects the number of tests written and that there is retention of TDD on the investigated constructs. However, we cannot exclude that TDD has some side effects that our study was not able to reveal, as well as that TDD is not retained for non-investigated constructs. To deal with this threat, we selected the dependent variables according to industrial needs [54] as well as our previous experiences [5, 8, 16].

*5.3.3. Threats to Conclusion Validity*

This kind of threat concerns the relationship between dependent and independent variables.

- **Reliability of treatment implementation.** Some participants might have followed the TDD approach more strictly than others. This could threaten the validity of the obtained results. Moreover, some participants might have followed the TDD approach when they were asked to use the YW approach (*i.e.,* in P3) or vice-versa (*i.e.,* in P2 and P4). To mitigate this threat, we reminded the participants several times to follow the treatment they were assigned to. Another threat concern the time span considered in our study (*e.g.,* a longer time span could negatively affect the TDD retainment). Given that there is no guideline on the time-span duration and a previous study has considered a six-month time span [22], we believed that a time span of about six months sufficed to (preliminary) study the TDD retainment even though we advise future research on the effect of longer time spans on the TDD retainment. However, a larger time span would introduce some problems to counteract: participants could drop out (shrinking the sample size and decreasing the amount of data collected), while others could simply loose the motivation to participate.

- **Random heterogeneity of participants.** There is always heterogeneity in a study group [38]. To lessen this threat, our study group consisted of students with similar background—*i.e.,* students taking the same courses in the same university with similar development experience. We collected the general information of the sample through a questionnaire before assigning students to the groups.

- **Reliability of measures.** To measure sequencing, granularity, uniformity, and refactoring effort, we exploited the Besouro plugin. Its use might threaten the validity of our results. However, Besouro represents the state-of-the-art tool for capturing the cycles when applying the TDD approach (*e.g.,* [5, 10, 16, 55]).

*5.3.4. Threats to External Validity*

External validity threats concern the ability to generalize the results.

- **Interaction of selection and treatment.** The participants of our cohort study were students and this might affect the generalizability of findings with respect to software professionals. However, the used development tasks did not require a high level of professional programming experience. Therefore, we believe that involving students in our study could be considered appropriate, as suggested in the literature [56, 57]. Moreover, the use of students as participants bring also a number of advantages like having a homogeneous group of participants, having an opportunity of obtaining preliminary empirical evidence, *etc.* [56]. In this respect, thanks to the use of students as participants, we could bring some evidence on the TDD retainment through a longitudinal cohort study.

- **Interaction of setting and treatment.** The used code katas might not be representative of real-world development tasks. However, code katas are widely utilized to assess TDD (*e.g.,* [5, 7, 8, 9]) because they allows having a better control over the participants. For example, such code katas are conceived to be completed in an experimental session of approximately three hours.

## 6. Conclusion

In this paper, we present the results from a quantitative longitudinal cohort study with 30 novice developers to investigate: *(i)* the retainment of TDD over a time span of about six months and *(ii)* the effects of TDD. We found that developers retain TDD over the considered time span. This empirical evidence on the retainment of TDD, together with past preliminary empirical evidence, allows us to conclude that the investment to train TDD developers is guaranteed at least for a time span of six months. We also show that TDD has no effect on external quality of software products and developers' productivity even when novice developers are tested again about six months later, under similar conditions. However, we observed that the participants practicing TDD wrote significantly more unit tests, with a better fault-detection capability, than those practicing a non-TDD approach. These results should foster software companies that value unit testing to have teams of developers that know TDD.

## Acknowledgements

We would like the thank the students for their participation in our study.

## References

[1] K. Beck, Test-driven development: by example, Addison-Wesley, 2003.

[2] D. Astels, Test driven development: A practical guide, Prentice Hall, 2003.

[3] H. Erdogmus, G. Melnik, R. Jeffries, Test-driven development, in: Encyclopedia of Software Engineering, Taylor & Francis, 2010, pp. 1211–1229.

[4] R. Jeffries, G. Melnik, Guest editors' introduction: Tdd–the art of fearless programming, IEEE Software 24 (3) (2007) 24–30. doi:10.1109/MS.2007.75.

[5] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, N. Juristo, 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) (2017) 597–614.

[6] I. Karac, B. Turhan, What do we (really) know about test-driven development?, IEEE Software 35 (4) (2018) 81–85. doi:10.1109/MS.2018.2801554.

[7] A. Tosun, O. Dieste, D. Fucci, S. Vegas, B. Turhan, H. Erdogmus, A. Santos, M. Oivo, K. Toro, J. Jarvinen, N. Juristo, An industry experiment on the effects of test-driven development on external quality and productivity, Empirical Software Engineering 22 (6) (2017) 2763–2805.

[8] D. Fucci, G. Scanniello, S. Romano, M. Shepperd, B. Sigweni, F. Uyaguari, B. Turhan, N. Juristo, M. Oivo, An external replication on the effects of test-driven development using a multi-site blind analysis approach, in: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '16, ACM, 2016, pp. 3:1–3:10. doi:10.1145/2961111.2962592.

[9] H. Erdogmus, M. Morisio, M. Torchiano, On the effectiveness of the test-first approach to programming, IEEE Transactions on Software Engineering 31 (3) (2005) 226–237.

[10] S. Romano, D. Fucci, G. Scanniello, B. Turhan, N. Juristo, Results from an ethnographically-informed study in the context of test driven development, in: Proceedings of International Conference on Evaluation and Assessment in Software Engineering, EASE '16, ACM, 2016, pp. 10:1–10:10. doi:10.1145/2915970.2915996.

[11] G. Scanniello, S. Romano, D. Fucci, B. Turhan, N. Juristo, Students' and professionals' perceptions of test-driven development: A focus group study, in: Proceedings of Annual ACM Symposium on Applied Computing, SAC '16, ACM, New York, NY, USA, 2016, pp. 1422–1427. doi:10.1145/2851613.2851778.

[12] B. George, L. Williams, A structured experiment of test-driven development, Information and Software Technology 46 (5) (2004) 337 – 342. doi:10.1016/j.infsof.2003.09.011.

[13] T. Bhat, N. Nagappan, Evaluating the efficacy of test-driven development: Industrial case studies, in: Proceedings of International Symposium on Empirical Software Engineering, ISESE '06, ACM, 2006, pp. 356–363. doi:10.1145/1159733.1159787.

[14] N. Nagappan, E. M. Maximilien, T. Bhat, L. Williams, Realizing quality improvement through test driven development: Results and experiences of four industrial teams, Empirical Softw. Engg. 13 (3) (2008) 289–302. doi:10.1007/s10664-008-9062-z.

[15] W. Bissi, A. G. S. S. Neto, M. C. F. P. Emer, The effects of test driven development on internal quality, external quality and productivity: A systematic review, Information and Software Technology 74 (2016) 45–54. doi:10.1016/j.infsof.2016.02.004.

[16] D. Fucci, B. Turhan, N. Juristo, O. Dieste, A. Tosun-Misirli, M. Oivo, Towards an operationalization of test-driven development skills: An industrial empirical study, Information and Software Technology 68 (2015) 82–97.

[17] B. Turhan, L. Layman, M. Diep, H. Erdogmus, F. Shull, How effective is test-driven development?, in: O. Media (Ed.), Making Software: What Really Works, and Why We Believe It, 2010, pp. 207–219.

[18] H. Munir, M. Moayyed, K. Petersen, Considering rigor and relevance when evaluating test driven development: A systematic review, Information and Software Technology 56 (4) (2014) 375–394.

[19] Y. Rafique, V. B. Mišić, The effects of test-driven development on external quality and productivity: A meta-analysis, IEEE Transactions on Software Engineering 39 (6) (2013) 835–856.

[20] F. Shull, G. Melnik, B. Turhan, L. Layman, M. Diep, H. Erdogmus, What do we know about test-driven development?, IEEE software 27 (6) (2010) 16–19.

[21] M. M. Müller, A. Höfer, The effect of experience on the test-driven development process, Empirical Software Engineering 12 (6) (2007) 593–615.

[22] R. Latorre, Effects of developer experience on learning and applying unit test-driven development, IEEE Transactions on Software Engineering 40 (4) (2014) 381–395.

[23] E. J. Caruana, M. Roman, J. Hernández-Sánchez, P. Solli, Longitudinal studies, Journal of Thoracic Disease 7 (11) (2015) 537–540. doi:10.3978/j.issn.2072-1439.2015.10.63.

[24] D. Fucci, S. Romano, M. T. Baldassarre, D. Caivano, G. Scanniello, B. Turhan, N. Juristo, A longitudinal cohort study on the retainment of test-driven development, in: Proceedings of International Symposium on Empirical Software Engineering and Measurement, ESEM '18, ACM, 2018,

pp. 18:1–18:10, paper preprint: https://arxiv.org/pdf/1807.02971.pdf. doi:10.1145/3239235.3240502.

[25] R. K. Yin, Case study research: Design and methods (applied social research methods), London and Singapore: Sage.

[26] L. McLeod, S. G. MacDonell, B. Doolin, Qualitative research on software development: a longitudinal case study methodology, Empirical software engineering 16 (4) (2011) 430–459.

[27] T. D. Cook, D. T. Campbell, W. Shadish, Experimental and quasi-experimental designs for generalized causal inference, Houghton Mifflin Boston, 2002.

[28] J. Li, N. B. Moe, T. Dybå, Transition from a plan-driven process to scrum: a longitudinal case study on software quality, in: Proc. ACM-IEEE Empirical software engineering and measurement, ESEM, ACM, 2010, p. 13.

[29] O. Salo, P. Abrahamsson, Integrating agile software development and software process improvement: a longitudinal case study, in: Int.Symposium Empirical Software Engineering, 2005, p. 10.

[30] J. Vanhanen, C. Lassenius, M. V. Mantyla, Issues and tactics when adopting pair programming: A longitudinal case study, in: Software Engineering Advances, 2007. ICSEA 2007. International Conference on, IEEE, 2007, pp. 70–70.

[31] D. E. Harter, C. F. Kemerer, S. A. Slaughter, Does software process improvement reduce the severity of defects? a longitudinal field study, IEEE Transactions on Software Engineering 38 (4) (2012) 810–827.

[32] H. Borges, A. Hora, M. T. Valente, Understanding the factors that impact the popularity of github repositories, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2016, pp. 334–344.

[33] A. Marchenko, P. Abrahamsson, T. Ihme, Long-term effects of test-driven development a case study, in: Int. Conf. on Agile Processes and Extreme Programming in Software Engineering, Springer, 2009, pp. 13–22.

[34] M. Beller, G. Georgios, P. Annibale, P. Sebastian, A. Sven, A. Zaidman, Developer testing in the ide: Patterns, beliefs, and behavior, IEEE Transactions on Software Engineering (1) (2017) 1–12.

[35] N. Borle, M. Feghhi, E. Stroulia, R. Greiner, A. Hindle, Analyzing the effects of test driven development in github, Empirical Software Engineering 23 (2017) 1–28. doi:10.1007/s10664-017-9576-3.

[36] S. Vegas, C. Apa, N. Juristo, Crossover designs in software engineering experiments: Benefits and perils, IEEE Transactions on Software Engineering 42 (2) (2016) 120–135.

[37] N. Juristo, A. M. Moreno, Basics of Software Engineering Experimentation, Kluwer Academic Publishers, 2001.

[38] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, A. Wessln, Experimentation in Software Engineering, Springer, 2012.

[39] A. Jedlitschka, M. Ciolkowski, D. Pfahl, Reporting experiments in software engineering, in: In Guide to Advanced Empirical Software Engineering, Springer, 2008, pp. 201–228.

[40] E. I. Karac, B. Turhan, N. Juristo, A controlled experiment with novice developers on the impact of task description granularity on software quality in test-driven development, IEEE Transactions on Software Engineering (2019) 1–1. doi:10.1109/TSE.2019.2920377.

[41] O. Dieste, A. M. Aranda, F. Uyaguari, B. Turhan, A. Tosun, D. Fucci, M. Oivo, N. Juristo, Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study, Empirical Software Engineering 22 (5) (2017) 2457–2542. doi:10.1007/s10664-016-9471-3.

[42] P. C. Jorgensen, Software Testing: a Craftsman's Approach, 4th Edition, Auerbach Publications, 2013.

[43] R. Just, The major mutation framework: Efficient and scalable mutation analysis for java, in: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2014, ACM, 2014, pp. 433–436. doi:10.1145/2610384.2628053.

[44] M. Papadakis, D. Shin, S. Yoo, D.-H. Bae, Are mutation scores correlated with real fault detection?: A large scale empirical study on the relationship between mutants and real faults, in: Proceedings of International Conference on Software Engineering, ICSE '18, ACM, 2018, pp. 537–548. doi:10.1145/3180155.3180183.

[45] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing?, in: Proceedings of International Symposium on Foundations of Software Engineering, FSE 2014, ACM, 2014, pp. 654–665. doi:10.1145/2635868.2635929.

[46] H. Kou, P. M. Johnson, H. Erdogmus, Operational definition and automated inference of test-driven development with zorro, Automated Software Engineering 17 (1) (2009) 57. doi:10.1007/s10515-009-0058-8.

[47] K. Becker, B. de Souza Costa Pedroso, M. S. Pimenta, R. P. Jacobi, Besouro: A framework for exploring compliance rules in automatic tdd behavior assessment, Information and Software Technology 57 (2015) 494 – 508. doi:10.1016/j.infsof.2014.06.003.

[48] G. Verbeke, G. Molenberghs, D. Rizopoulos, Random Effects Models for Longitudinal Data, Springer, 2010, pp. 37–96.

[49] S. Shapiro, M. Wilk, An analysis of variance test for normality, Biometrika 52 (3-4) (1965) 591–611.

[50] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, J. Rosenberg, Preliminary guidelines for empirical research in software engineering, IEEE Trans. on Soft. Eng. 28 (8) (2002) 721–734.

[51] V. Basili, F. Shull, F. Lanubile, Building knowledge through families of experiments, IEEE Trans. on Soft. Eng. 25 (4) (1999) 456–473.

[52] J. Sillito, G. C. Murphy, K. De Volder, Asking and answering questions during a programming change task, IEEE Transactions on Software Engineering 34 (4) (2008) 434–451. doi:10.1109/TSE.2008.26.

[53] D. Fucci, B. Turhan, A replicated experiment on the effectiveness of test-first development, in: Empirical Software Engineering and Measurement, 2013, IEEE, 2013, pp. 103–112.

[54] A. Causevic, D. Sundmark, S. Punnekkat, Factors limiting industrial adoption of test driven development: A systematic review, in: Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on, IEEE, 2011, pp. 337–346.

[55] S. Romano, D. Fucci, G. Scanniello, B. Turhan, N. Juristo, Findings from a multi-method study on test-driven development, Information and Software Technology 89 (2017) 64–77. doi:10.1016/j.infsof.2017.03.010.

[56] J. Carver, L. Jaccheri, S. Morasca, F. Shull, Issues in using students in empirical studies in software engineering education, in: Proceedings of International Symposium on Software Metrics, IEEE, 2003, pp. 239–249.

[57] M. Höst, B. Regnell, C. Wohlin, Using students as subjects—a comparative study of students and professionals in lead-time impact assessment, Empirical Software Engineering 5 (3) (2000) 201–214.