The Journal of Systems & Software 176 (2021) 110937

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Studying test-driven development and its retainment over a six-month time span ☆

Maria Teresa Baldassarre [a,1], Danilo Caivano [a,1], Davide Fucci [b,1], Natalia Juristo [c,1], Simone Romano [a,*,1], Giuseppe Scanniello [d,1], Burak Turhan [e,f,1]

[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] University of Oulu, Oulu, Finland
[f] Monash University, Melbourne, Australia

ARTICLE INFO

Article history:
Received 27 April 2020
Received in revised form 15 October 2020
Accepted 15 February 2021
Available online 9 March 2021

Keywords:
Test-driven development
TDD
Longitudinal cohort study

ABSTRACT

In this paper, we investigate the effect of TDD, as compared to a non-TDD approach, as well as its *retainment* (or *retention*) over a time span of (about) six months. To pursue these objectives, we conducted a (quantitative) longitudinal cohort study with 30 novice developers (*i.e.,* third-year undergraduate students in Computer Science). We observed that TDD 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. As for the retainment of TDD, we found that TDD is retained by novice developers for at least six months.

1. Introduction

Test-Driven Development (TDD) (Beck, 2003; Astels, 2003) is a cyclic development approach where unit tests drive the incremental development of small pieces of functionality (Erdogmus et al., 2010). 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 (Astels, 2003). 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 (Beck, 2003; Astels, 2003). Advocates of TDD recommend ending a development cycle in few minutes (five or ten minutes Jeffries and Melnik, 2007) and keeping the rhythm as uniform as possible over time (Beck, 2003; Erdogmus et al., 2010). 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 *test-first dynamic*) (Fucci et al., 2017). It is worth noting that test-first sequencing refers to just one central aspect of TDD (Karac and Turhan, 2018). That is, it does not capture the full nature of TDD (Fucci et al., 2017). Other central aspects that characterize the development process underlying TDD are: *granularity*, *uniformity*, and *refactoring effort* (Fucci et al., 2017). Granularity refers to the duration of the development cycles, while uniformity reflects how constant their duration is over time (Fucci et al., 2017). Finally, refactoring effort captures how much refactoring a TDD practitioner performs.

It is claimed that TDD leads to higher-quality software products in terms of both external (*i.e.,* functional) and internal quality, while increasing developers' productivity (Beck, 2003). These claimed benefits have encouraged some software companies to adopt TDD, while others are considering its adoption (Tosun et al., 2017). TDD has been assessed from a quantitative point of view (*e.g.,* Fucci et al. (2016), Erdogmus et al. (2005)) and according to a qualitative perspective (*e.g.,* Romano et al. (2016), Scanniello et al. (2016)). A number of primary studies, like experiments or case studies, have been conducted on TDD (Fucci et al., 2016;

☆ Editor: Sarah Beecham.

* Corresponding author.
E-mail addresses: mariateresa.baldassarre@uniba.it (M.T. Baldassarre), danilo.caivano@uniba.it (D. Caivano), davide.fucci@bth.se (D. Fucci), natalia@fi.upm.es (N. Juristo), simone.romano@uniba.it (S. Romano), giuseppe.scanniello@unibas.it (G. Scanniello), burak.turhan@oulu.fi (B. Turhan).

[1] The authors have equally contributed to the research presented in the paper.

https://doi.org/10.1016/j.jss.2021.110937
0164-1212/

Erdogmus et al., 2005; George and Williams, 2004; Bhat and Nagappan, 2006; Nagappan et al., 2008). Their results, gathered and combined in a number of secondary studies (Karac and Turhan, 2018; Bissi et al., 2016; Fucci et al., 2015; Turhan et al., 2010; Munir et al., 2014; Rafique and Mišić, 2013), do not fully support the claimed benefits of TDD (*i.e.,* while some primary studies have shown that TDD allows improving quality of software products and/or developers' productivity, other primary studies have not). Some researchers have conjectured that long-term observations are needed to see the claimed benefits of TDD and/or to better understand this development approach; therefore, they have recommended taking a longitudinal approach when investigating TDD (Fucci et al., 2015; Munir et al., 2014; Shull et al., 2010; Müller and Höfer, 2007)—*i.e.,* studying TDD over a time span. Nevertheless, only Latorre (2014), Borle et al. (2017), Beller et al. (2017), and Marchenko et al. (2009) have taken a longitudinal approach.

Longitudinal studies[2] employ continuous or repeated measures to follow particular individuals over a time span of weeks, months, or even years (Caruana et al., 2015). In this paper, we present a study on TDD that takes a longitudinal approach. In particular, we conducted a longitudinal cohort study in which our *cohort* consisted of 30 novice developers of homogeneous experience who attended the same training regarding agile software development, including TDD. The design of our study allowed us to have a term of comparison between TDD and a non-TDD approach, defined as the approach that developers would normally follow (*e.g.,* iterative test-last, big-bang testing, or no testing at all—but not TDD), with respect to external quality, developers' productivity, number of tests written, and fault-detection capability of tests written. Moreover, thanks to our 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 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* (or *retention*).[3]

While 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. Moreover, our results indicate that novice developers retain TDD for at least six months (*i.e.,* the time span from when novice developers learned and applied TDD for the first time to when they applied TDD again).

This paper extends the one by Fucci et al. (2018) as follows:

- 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 investigated the retainment of TDD with respect to four aspects that characterize the process underlying TDD: test-first sequencing, granularity, uniformity, and refactoring effort.
- 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 Fucci et al.'s paper.

[2] There are three major types of longitudinal studies: *(i)* repeated cross-sectional studies; *(ii)* prospective studies (including cohort studies); and *(iii)* retrospective studies (Caruana et al., 2015).

[3] TDD retainment concerns the capability of a developer to apply this development approach after she did not apply it for a certain time span.

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

The effect of TDD on several outcomes – including functional quality and productivity, which are of interest for this study – has been the topic of several empirical studies, summarized in Systematic Literature Reviews (SLRs) and meta-analyses (Bissi et al., 2016; Turhan et al., 2010; Munir et al., 2014; Rafique and Mišić, 2013). The SLR by Turhan et al. (2010) 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.[4] Bissi et al. (2016) 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, the results are inconclusive. Rafique and Mišić (2013) conducted a meta-analysis of 25 controlled experiments published 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. (2014) 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. A summary of the outcomes from the above-mentioned studies is reported in Table 1. We can suppose that the inconclusive results about the claimed benefits of TDD depend on the participants in the studies and their set-up (Karac and Turhan, 2018). For example, most studies have focused on only the test-first aspect. In addition, TDD has too many cogs and these cogs highly interact with each other (Karac and Turhan, 2018). There are also so many potential variation points in the context of the studies, as well as in the personality factors of the participants. We deem these points deserved to be studied in more detail, along with the study of theorizing what might be barriers and supporting factors to the application of TDD. To that end, Munir et al. (2014) suggested that more long-term studies are needed to better understand TDD.

An example of long-term investigation is the one by Marchenko et al. (2009). 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's improved confidence with the codebase. 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. (2017) executed a long-term study 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.

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

Table 1
Summary of secondary studies on TDD.

Study	Conclusion for quality	Conclusion for productivity	Inconsistencies in the study categories
Turhan et al. (2010)	Improvement	Inconclusive	Quality: - Among controlled experiments - Among studies with high rigor Productivity: - Among pilot studies - Controlled experiments vs. industrial case studies - Among studies with high rigor
Bissi et al. (2016)	Improvement	Inconclusive	Productivity: Academic vs. industrial setting
Rafique and Mišić (2013)	Improvement	Inconclusive	Quality: Waterfall vs. iterative test-last Productivity: - Waterfall vs. iterative test-last - Academic vs. industrial
Munir et al. (2014)	Improvement or no difference	Degradation or no difference	Quality: - Low vs. high rigor - Low vs. high relevant Productivity: - Low vs. high rigor - Low vs. high relevant

Borle et al. (2017) 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 from one another based on the extent to which TDD was adopted within these projects. The sets of TDD projects were then compared with control sets 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 (2014) studied the capability of 30 professional 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 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 (2014) 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 some 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 (Latorre, 2014; Borle et al., 2017; Beller et al., 2017; Marchenko et al., 2009) have taken a longitudinal approach when studying TDD, none of them has mainly focused, as our longitudinal cohort study, on the retainment of TDD—although Latorre's study (Latorre, 2014) provides initial evidence on the retainment of TDD, the main goal of this study was the learnability of TDD.

3. Empirical study

The goal of a longitudinal study is to investigate *"how certain conditions change over time"* (Yin, 2009). 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. 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 (Cook et al., 2002) in which, due to the lack of experimental manipulation, a specific event is used to identify the experimental groups. A third kind of longitudinal study in software engineering retrospectively covers an extended time span by analyzing archival data. 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.

In medicine, longitudinal studies are sometimes realized in the form of cohort studies. A cohort is a sample of participants (*e.g.,* who undergo a treatment) sharing a specific characteristic of interest (*e.g.,* age). The cohort is tested in some occasions over time to, for example, check for a drug side-effect before releasing it to the market (Cook et al., 2002).

In our longitudinal study, the participants were third-year undergraduate students in Computer Science, who were asked to take part in four experimental sessions over a six-month time span. In any experimental session, each participant had to perform a development task by following either TDD or a non-TDD approach. Before the first experimental session, all participants had practiced unit testing, iterative test-last development, and big-bang testing thanks to training sessions.

In the first experimental session—held in the first period,[5] P1 (*i.e.,* on November 16th, 2016)—, the participants had to use a non-TDD approach (*e.g.,* iterative test-last, big-bang testing, or no

[5] A period is the time during which a treatment is applied (Vegas et al., 2016).

testing at all) to perform the development tasks. The participants learned and practiced TDD between the first experimental session and the second one—held in the second period, P2 (*i.e.,* on December 7th, 2016). That is to say, no one in the first experimental session knew TDD. Later, all participants used TDD to perform the development tasks in the second experimental session.

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—held, respectively, in the third and fourth periods, P3 (*i.e.,* on May 3rd, 2017) and P4 (*i.e.,* on May 4th, 2017). In particular, in the third experimental session, all participants followed a non-TDD approach when performing the development tasks. As for the fourth experimental session, the participants followed TDD to perform the development tasks. We refer to the non-TDD approach we used to have a term of comparison as *Your Way* (*YW*, from here onwards).

In the following of this section, we report the planning and execution of our study, which we briefly summarized above, by taking into account the template by Jedlitschka et al. (2008). To plan and execute our study, we followed the recommendations by Juristo and Moreno (2001), and Wohlin et al. (2012).

3.1. Research questions

We investigated the following Research Questions (RQs):

RQ1. Are there differences between TDD and YW in terms of *(i)* external quality of the implemented solutions, *(ii)* developers' productivity, *(iii)* number of tests written, and *(iv)* fault-detection capability of tests written?

Aim. With RQ1 we wanted to contrast TDD and YW in terms of the constructs mentioned above. 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 (Beck, 2003). We focused on the number of tests because past research has shown that TDD results in more tests (Erdogmus et al., 2005). 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. We did not consider the four dimensions that characterize the development process underlying TDD (*i.e.,* test-first sequencing, granularity, uniformity, and refactoring effort) because these dimensions do not characterize YW. It is worth recalling that the study on the fault-detection capability of tests written is a new contribution with respect to Fucci et al.'s study (Fucci et al., 2016).

RQ2. Is the capability of novice developers to apply TDD affected when they did not use this development approach for a certain time span with respect to *(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?

Aim. RQ2 was defined to study the retainment of TDD, with respect to different constructs, by taking advantage of the longitudinal approach behind our study. The results from this RQ can be used to draw conclusions about the TDD retainment over a six-month time span, not to evaluate the evolution of the TDD technique over time. The reasons behind the study of external quality of the implemented solutions, developers' productivity, number of tests written, and fault-detection capability of tests written are the same as RQ1. We also considered test-first sequencing, granularity, uniformity, and refactoring effort because these are the four dimensions that characterize the development process underlying TDD (Fucci et al., 2017). 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 Fucci et al.'s study (Fucci et al., 2016).

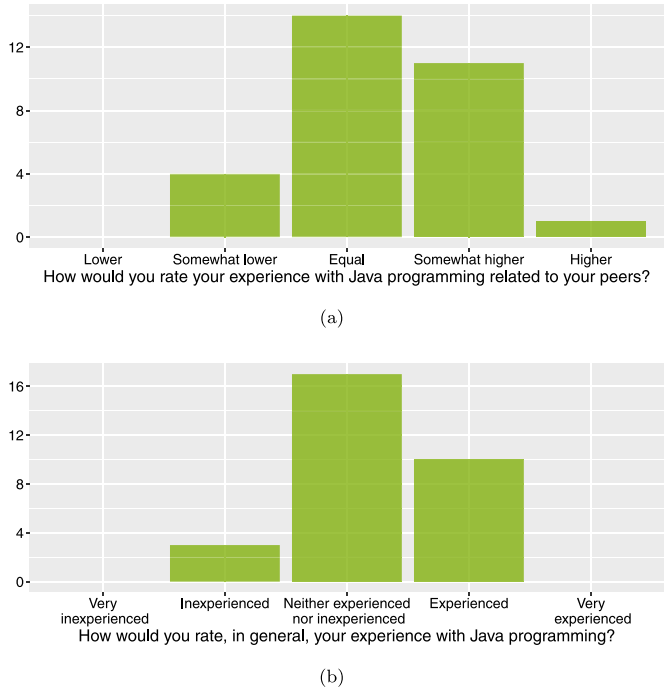

Fig. 1. Barplots on the participants' experience with Java programming related to (a) their peers and (b) in general. The data about the participants' experience were gathered at the beginning of the study—*i.e.,* before the training sessions.

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 in 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 study (*i.e.,* in any of the fourth experimental sessions) was equal to 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 Fig. 1a, 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. Fig. 1b 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 the 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; and computing the score of a single frame as well as the score of a bowling game.
- **Mars Rover API (MRA).** This kata aims to develop 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 string 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 to develop 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) (Karac et al., 2019). 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 suites because their purpose was the assessment of the implemented features. That is, the acceptance test suites were used to quantify the external quality of the solutions implemented by the participants as well as their productivity (see Section 3.6).

Our decision to adopt code katas is because their use is common in empirical studies on TDD (Fucci et al., 2017; Tosun et al., 2017; Fucci et al., 2016; Erdogmus et al., 2005; Karac et al., 2019; Dieste et al., 2017). Furthermore, this allowed us to use existing experimental materials (*e.g.,* from the studies by Fucci et al. (2016) and Dieste et al. (2017)). BSK and MRA have been also used as experimental objects in several empirical studies (Fucci et al., 2017; Tosun et al., 2017; Fucci et al., 2016; Karac et al., 2019; Dieste et al., 2017). As 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 development tasks (in four experimental sessions), one for each experimental object. 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 out a development 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 we collected data over time, we had a second main independent variable 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.

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 (Fucci et al., 2017; Tosun et al., 2017; Fucci et al., 2016; Erdogmus et al., 2005).

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.,* Fucci et al. (2016)):

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

where #TF is the number of features a participant tackled, while $QLTY_i$ is the external quality of the *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 $QLTY_i$, 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 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 development task. PROD is defined as follows (*e.g.,* Tosun et al. (2017)):

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

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 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 development task (*e.g.,* (Fucci et al., 2016)). TEST assumes (integer) values between 0 and ∞. A high value is desirable.

To quantify fault-detection capability of tests written, we leveraged mutation testing (Jorgensen, 2013). 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 (Just, 2014). We opted for this framework because it is robust (Papadakis et al., 2018), publicly available (Just, 2014), and has been adopted in previous work (*e.g.,* Papadakis et al. (2018), Just et al. (2014)). We applied the following mutation operators[6] to generate mutants: AOR, LOR, COR, ROR, ORU, LVR, and STD. A description of these operators is available in Table 2. This set of mutation operators is the same as Papadakis et al. (2018) 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.,* Jorgensen (2013)):

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

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 the MUT value, 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 (Papadakis et al., 2018). 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 into small cycles as done by Fucci et al. (2017). 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. (2009), it is possible to determine the type of each cycle (*e.g.,* test-first or refactoring). In Table 3, we report the heuristics we exploited to determine the type of the cycles when the participants applied TDD. The considered heuristics are implemented in the *Besouro* tool (Becker et al., 2015).

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 (Fucci et al., 2017):

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

where #CYCLES(TEST-FIRST) is the number of cycles classified as test-first by applying the heuristics in Table 3 when a participant followed TDD. #CYCLES(ALL) is, instead, the total number of cycles for that participant. The SEQ variable assumes values between 0 and 100. The higher the value for this variable, the higher the amount of test-first cycles, when a participant applied TDD, is.

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 (Fucci et al., 2017). This variable ranges between 0 and ∞. 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 (Fucci et al., 2017).

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 ∞. The lower the UNI value, the more uniform the cycles carried out by a participant are. A UNI value equal to 0 means

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

Table 2
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)

Table 3
Heuristics implemented in Besouro (Becker et al., 2015) to determine the type of cycles (the description of these heuristics is taken from Kou et al.' paper Kou et al., 2009).

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

that the cycles had mostly the same duration. The use of MAD to compute GRA, rather than standard deviation, allows reducing the sensitivity to outliers (Fucci et al., 2017).

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 (Fucci et al., 2017):

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

where #CYCLES(REFACTORING) is the number of cycles classified as refactoring (by using the heuristics in Table 3) when a participant followed TDD. The REF variable assumes values between 0 and 100. The higher the value for this variable, the higher the refactoring effort of a participant, when he/she applied TDD, is.

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:

$HN1_X$. There is no statistically significant effect of Approach with respect to X (*i.e.*, QLTY, PROD, TEST, or MUT).

$HN2_X$. There is no statistically significant effect of Period with respect to X (*i.e.*, QLTY, PROD, TEST, MUT, SEQ, GRA, UNI, or REF).

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

3.8. Study design

In Table 4, we summarize the design of our cohort study. We randomly split the participants into two groups, 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. We used two experimental groups, rather than only one, to control for the effect of the experimental objects.

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. 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.

The first application of the TDD treatment took place on December 7th, 2016 (*i.e.*, P2). The participant learned TDD between

Table 4
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

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 some 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. In P3, we informed the students not to use TDD, in order to avoid affecting the results in an undesirable way. Although we asked the participants not to use TDD, they knew TDD. Therefore, we cannot exclude that some knowledge of TDD had affected the YW treatment in P3. On the other hand, we assessed the retainment of TDD by asking the participants to use TDD (once again) in P4.

The execution of our 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 a 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 lose 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 (2014) (*i.e.,* the only study that has somehow investigated the retainment of TDD, see Section 2).

The development 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. Each experimental session lasted three hour and a half.

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 (Becker et al., 2015). 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 development 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, the 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 used 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 (Verbeke et al., 2010). LMMs are an extension of linear models containing both fixed and random effects. As for $HN1_X$, we built, for the dependent variable X (*e.g.,* QLTY), the following LMM:

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

where **Approach** (*i.e.,* the main independent variable), **Group**—it assumes G1 or G2 as a value—, and their interaction (*i.e.,* **Approach:Group**) are the fixed effects. $LMM1_X$ 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 (Vegas et al., 2016). When building $LMM1_X$, 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) (Vegas et al., 2016). If $LMM1_X$ revealed a statistically significant effect of Approach, we could reject $HN1_X$. The use of $LMM1_X$ is new with respect to the paper by Fucci et al. (2018).
To test $HN2_X$, we built a second LMM that included Period (instead of Approach):

$$LMM2_X = X \sim Period + Group + Period : Group + (1|Participant) \quad (9)$$

LMMs have two assumptions that must be met: *(i)* the model residuals must be normally distributed; and *(ii)* their mean must be zero (Vegas et al., 2016). In case LMM assumptions are not met, transforming the dependent variable values is an option (*e.g.,* log-transformation) (Vegas et al., 2016). To check the normality of the residuals, we used the Shapiro–Wilk test (Shapiro test, from here onwards) (Shapiro and Wilk, 1965).
Whatever the test of statistical significance was, we set the α value at 0.05—*i.e.,* we accepted a probability of 5% of committing a 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.

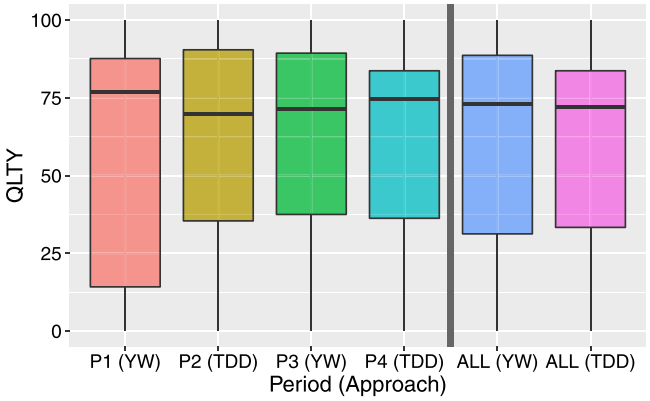

Fig. 2. Boxplots for QLTY (*i.e.,* external quality of implemented solutions) grouped by Period and Approach.

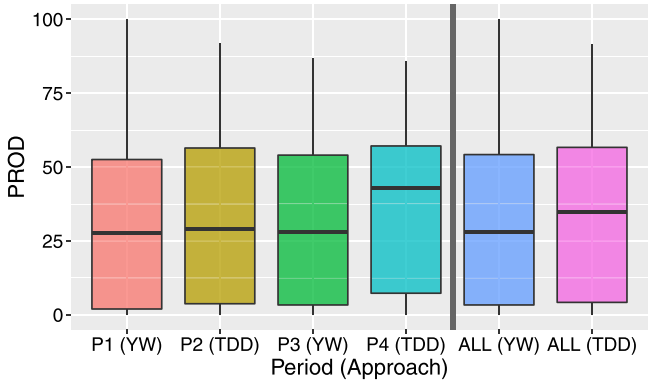

Fig. 3. Boxplots for PROD (*i.e.,* developers' productivity) grouped by Period and Approach.

4.1. Descriptive statistics and exploratory analyses

In Table 5, 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 Fig. 2, we report the boxplots for the QLTY variable grouped by Period or Approach. The (overall) results summarized in Fig. 2, along with those summarized in Table 5, do not suggest differences in the QLTY values between TDD and YW (*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 the participants applied either TDD or YW while tackling the same experimental objects. The comparison between P2 and P3 leads to a similar observation.

By looking at the boxplots in Fig. 2, we can notice that there are no 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 5). As for the periods P2 and P4—*i.e.,* same TDD treatment but different experimental objects—, the boxplots in Fig. 2 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 evidence that TDD is not retained with respect to QLTY.

4.1.2. PROD—Developers' productivity

In Fig. 2, we report the boxplots for the PROD variable grouped by Period or Approach. Overall, it seems there is a slight difference in the PROD values between TDD and YW (see Table 5 and Fig. 3). 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 median values are equal to 42.85 and 27.52 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 Fig. 3).

By observing the boxplots for PROD in Fig. 3, 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 that for P2 (42.85 vs. 29.06). Such an improvement in the PROD values, when passing from P2 to P4, might indicate a retainment of TDD. 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.

4.1.3. TEST—Number of tests written

The boxplots for TEST are shown in Fig. 4. By observing the boxplots grouped by Approach, it seems that the participants who followed TDD wrote more tests. For instance, the participants achieved, on average, TEST values equal to 8.96 and 6.43 when following TDD and YW, respectively (see Table 5). 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.

The boxplots in Fig. 4 seem to suggest 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 5 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. 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.

Table 5
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

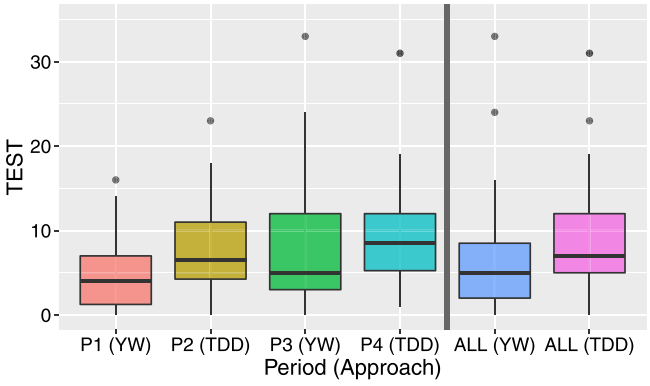

Fig. 4. Boxplots for TEST (*i.e.*, number of tests written) grouped by Period and Approach.

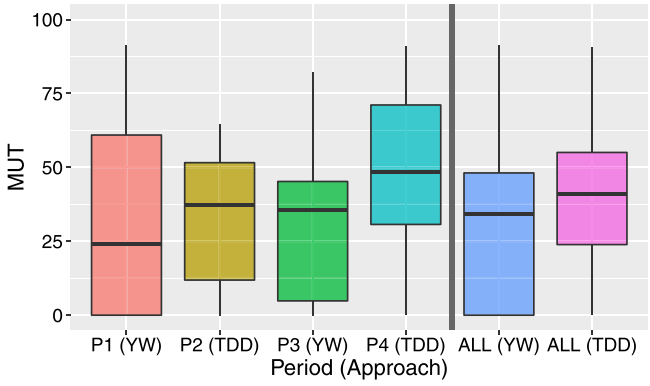

Fig. 5. Boxplots for MUT (*i.e.*, fault-detection capability of tests written) grouped by Period and Approach.

4.1.4. MUT—Fault-detection capability of tests written

The boxplots for MUT are shown in Fig. 5. By looking at the boxplots arranged by approach, 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 5. 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 Fig. 5 overlap. The descriptive statistics in Table 5 neither 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.

As Fig. 5 shows, 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 5). However, the mean values for P1 and P3 are almost identical (31.98 vs. 31.99). That 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 Fig. 5. 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 5 remark this improvement (*e.g.*, the mean value passes from 32.07 in P2 to 48.52 in P4). Summing up, it seems that TDD is retained with respect to MUT.

4.1.5. SEQ—Test-first sequencing

In Fig. 6a, 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.*,

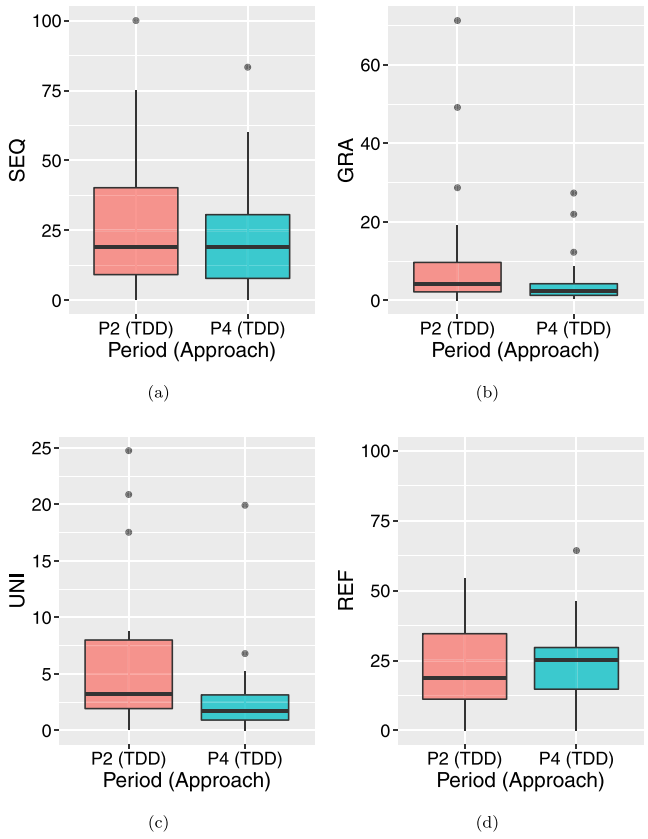

Fig. 6. Boxplots for (a) SEQ (*i.e.,* test-first sequencing), (b) GRA (*i.e.,* granularity), (c) UNI (*i.e.,* uniformity), and (d) REF (*i.e.,* refactoring effort) grouped by period (only P2 and P4).

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 5). 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.

4.1.6. GRA—Granularity

The boxplots for GRA are depicted in Fig. 6b. 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 the last period.

4.1.7. UNI—Uniformity

The distributions for UNI depicted in Fig. 6c look different. In particular, the boxplot for P2 is higher and larger than the one for P4. The descriptive statistics in Table 5 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 Fig. 6d, 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 are, 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 6, 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 assumption of normality was not met for $LMM1_{QLTY}$ since the Shapiro test returned a *p*-value equal to 0.0003. To meet this assumption, we had to use a (second) power transformation. After the data transformation, both assumptions were met—the residuals were normally distributed according to the Shapiro test (p-value = 0.285) and their mean was equal to zero. As shown in Table 6, the effect of Approach is not statistically significant (p-value = 0.8097). Therefore, we cannot reject $HN1_{QLTY}$. This outcome seems to suggest that developing according to the TDD approach does not influence the external quality of software products. $LMM1_{QLTY}$ also indicates a statistically significant interaction between Group and Approach, 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.

As for $LMM2_{QLTY}$, the assumptions 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 does not allow us to reject $HN2_{QLTY}$ because the *p*-value for Period is 0.8837 (see Table 6), 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. Again, there is a statistically significant interaction (*i.e.,* Period:Group), which is due to the effect of the experimental objects. We can therefore conclude that TDD can be retained in terms of external quality of software products.

4.2.2. PROD—Developers' productivity

The assumption of normality was not met for $LMM1_{PROD}$ 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 = 0.2069) and their mean was equal to zero. As shown in Table 6, we cannot reject $HN1_{PROD}$ since the effect of Approach for $LMM1_{PROD}$ is not statistically significant (p-value = 0.4967). This seems to suggest that developers who follow either the TDD or YW approach exhibit a similar productivity. $LMM1_{PROD}$ also includes a statistically significant effect, namely the one of Group:Approach. Again, this is due to the effect of the experimental objects. Finally, there is no statistically significant effect of Group.

Also for $LMM2_{PROD}$, the assumption of normality was not met—the Shapiro test returned a *p*-value equal to 0.0143. Again, we needed a square-root transformation to meet this assumption. After transforming the data, the LMM assumptions were both met—the residuals were normally distributed according to the Shapiro test (p-value = 0.0524) and their mean was equal to zero. As Table 6 suggests, we cannot reject $HN2_{PROD}$ because the effect of Period for $LMM2_{PROD}$ is not statistically significant (p-value =

Table 6
Results (*i.e.*, p-values) from the inferential statistics. Note that $LMM2_X$ is built only for SEQ, GRA, UNI, and REF.

Variable	$LMM1_X$			$LMM2_X$		
	Approach	Group	Approach:Group	Period	Group	Period:Group
QLTY	0.8097	0.4432	<0.0001*	0.8837	0.6108	<0.0001*
PROD	0.4967	0.8225	<0.0001*	0.7973	0.8225	<0.0001*
TEST	0.0005*	0.0617	0.7706	0.0002*	0.0617	0.4632
MUT	0.0454*	0.734	0.0025*	0.0017*	0.734	<0.0001*
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.

0.7973). This outcome indicates that the participants can retain TDD as far as their productivity is concerned. Finally, there is a statistically significant interaction (*i.e.*, Period:Group) due to the effect of the experimental objects.

4.2.3. TEST—Number of tests written

Since the residuals of $LMM1_{TEST}$ 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 $LMM1_{TEST}$ 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 6, $LMM1_{TEST}$ only includes a statistically significant effect, namely that of Approach (p-value = 0.0005). Therefore, we can reject $HN1_{TEST}$ and conclude that TDD significantly and positively affects the number of tests the participants wrote.

Also for $LMM2_{TEST}$, we needed a data transformation—the residuals for $LMM2_{TEST}$ 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. Table 6 shows a statistically significant effect of Period (p-value = 0.0002). Therefore, we can reject $HN2_{TEST}$ and conclude that Period significantly affects the number of tests the participants wrote. By looking at the boxplots in Fig. 4, 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.

4.3. MUT—Fault-detection capability of tests written

The LMM assumptions were both satisfied for $LMM1_{MUT}$—the Shapiro test returned a *p*-value equal to 0.247 and the mean of the residuals was equal to zero. $LMM1_{MUT}$ allows us to reject $HN1_{MUT}$ 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 tests. We can therefore conclude that TDD practitioners tend to write more tests than non-TDD ones and, moreover, the fault-detection capability of these tests is significantly better. As for the *p*-value (0.0025) of Approach:Group, it suggests that the fault-detection capability of the written tests can depend on the development task at hand.

The residuals of $LMM2_{MUT}$ were normally distributed (the Shapiro test returned a *p*-value equal to 0.5289) and their mean was equal to zero. $LMM2_{MUT}$ 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 $HN2_{MUT}$, so recognizing that Period significantly affects the fault-detection capability of the written tests. Moreover, we can observe an improvement of the MUT values in P4 with respect to any other period (*e.g.*, see Fig. 5). 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. Again, the fault-detection capability of the written tests seems to depend on the development task at hand.

4.3.1. SEQ—Test-first sequencing

The assumption of normality was not satisfied for $LMM2_{SEQ}$ (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 6, we can notice that $LMM2_{SEQ}$ does not include a statistically significant effect for Period (p-value = 0.4707). 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 to follow 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 $LMM2_{GRA}$, 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 6, the effect of Period for $LMM2_{GRA}$ is not statistically significant (p-value = 0.1992) although the boxplots in Fig. 6b highlighted an improvement in P4 (with respect to P2). On average, the granularity of the development cycles was 4.26 and 2.5 min. These outcomes suggest that developers can retain their ability to follow 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 $LMM2_{UNI}$. 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 6 indicate a statistically significant effect of Period (p-value = 0.021). The distributions of the UNI values (*e.g.*, see Fig. 6c) suggest 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 $LMM2_{REF}$ 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 6, do not highlight any statistically significant effect. Concluding, it seems that the refactoring effort is retained when practicing TDD. Again, the tasks seem not to 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 might have affected the validity of these findings.

5.1. Answering research questions

As for the comparison between TDD and YW, we observed differences in favor of the former 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). No other differences emerged. Accordingly, we can answer RQ1 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.

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). The uniformity of the cycles enhanced with time as well. On the basis of these results, we can answer RQ2 as follows.

Developers retain TDD at least for 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 (Kitchenham et al., 2002; Basili et al., 1999), as well as to have a first understanding of TDD retainment on several constructs.

5.2. Overall discussion and future research

As compared to developers who follow a non-TDD approach, we 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. (2005) so bringing further evidence that TDD has a positive effect on the number of written tests. Therefore, this finding goes 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.,* 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 shown that developers dealing with an unfamiliar codebase look for examples of input/output values to better understand that codebase Sillito et al., 2008—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 the TDD research (*e.g.,* Fucci et al. (2016) and Fucci and Turhan (2013)). 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 (Latorre, 2014). Although carrying out longitudinal studies 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 benefits of TDD to become apparent. As a starting point towards this direction, we recommend empirical 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 seems to justify future research on this matter.

We show that developers retain TDD, for at least, a time span of six months. This finding is in line with the preliminary empirical evidence gathered by Latorre (2014) 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 for a time span of six months. Furthermore, previous work has also shown that developers can correctly apply TDD after a short practical session only (Latorre, 2014). 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.

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 (Fucci et al., 2017). 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 effect and retainment of TDD, we foster replications of our study. To this end, we made available on the web our laboratory package:

- https://doi.org/10.6084/m9.figshare.14102063.v1.

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. (2012). 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) (Wohlin et al., 2012). 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 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 (Wohlin et al., 2012).
- **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 well 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.
- **History.** The time interval between the first and second applications of YW was slightly greater than the time interval between the first and second applications of TDD. This might affect the obtained results.

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 (Fucci et al., 2017; Tosun et al., 2017; Fucci et al., 2016; Erdogmus et al., 2005).
- **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 a retainment 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 (Causevic et al., 2011) as well as our previous experiences (Fucci et al., 2017, 2016, 2015).

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 (Latorre, 2014), 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 lose the motivation to participate.
- **Random heterogeneity of participants.** There is always heterogeneity in a study group (Wohlin et al., 2012). To lessen this threat, our study group consisted of students with similar backgrounds—*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 (Fucci et al., 2017; Romano et al., 2016; Fucci et al., 2015; Romano et al., 2017).

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 (Carver et al., 2003; Höst et al., 2000). 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.* (Carver et al., 2003). 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. The gathered evidence seems to justify field studies with professionals. This might represent a possible direction for future work.
- **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 (Fucci et al., 2017; Tosun et al., 2017; Fucci et al., 2016; Erdogmus et al., 2005) because they allow 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 the effect of TDD, as compared to a non-TDD approach, as well as the retainment of TDD over a time span of (about) six months.

As for the comparison of TDD with a non-TDD approach, we show that TDD has no effect on external quality of software products and developers' productivity. 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.

The results from our study also suggest that developers retain TDD at least for a six-month 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 six-month time span.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

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

References

Astels, D., 2003. Test Driven Development: A Practical Guide. Prentice Hall.
Basili, V., Shull, F., Lanubile, F., 1999. Building knowledge through families of experiments. IEEE Trans. Softw. Eng. 25 (4), 456–473.
Beck, K., 2003. Test-Driven Development: By Example. Addison-Wesley.
Becker, K., de Souza Costa Pedroso, B., Pimenta, M.S., Jacobi, R.P., 2015. Besouro: A framework for exploring compliance rules in automatic tdd behavior assessment. Inf. Softw. Technol. 57, 494–508. http://dx.doi.org/10.1016/j.infsof.2014.06.003.
Beller, M., Gousios, G., Panichella, A., Proksch, S., Amann, S., Zaidman, A., 2017. Developer testing in the ide: Patterns, beliefs, and behavior. IEEE Trans. Softw. Eng. (1), 1–12.
Bhat, T., Nagappan, N., 2006. Evaluating the efficacy of test-driven development: Industrial case studies. In: Proceedings of International Symposium on Empirical Software Engineering. In: ISESE '06, ACM, pp. 356–363. http://dx.doi.org/10.1145/1159733.1159787.
Bissi, W., Neto, A.G.S.S., Emer, M.C.F.P., 2016. The effects of test driven development on internal quality, external quality and productivity: A systematic review. Inf. Softw. Technol. 74, 45–54. http://dx.doi.org/10.1016/j.infsof.2016.02.004.
Borle, N., Feghhi, M., Stroulia, E., Greiner, R., Hindle, A., 2017. Analyzing the effects of test driven development in github. Empir. Softw. Eng. 23, 1–28. http://dx.doi.org/10.1007/s10664-017-9576-3.
Caruana, E.J., Roman, M., Hernández-Sánchez, J., Solli, P., 2015. Longitudinal studies. J. Thorac. Dis. 7 (11), 537–540. http://dx.doi.org/10.3978/j.issn.2072-1439.2015.10.63.
Carver, J., Jaccheri, L., Morasca, S., Shull, F., 2003. Issues in using students in empirical studies in software engineering education. In: Proceedings of International Symposium on Software Metrics. IEEE, pp. 239–249.
Causevic, A., Sundmark, D., Punnekkat, S., 2011. 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, pp. 337–346.
Cook, T.D., Campbell, D.T., Shadish, W., 2002. Experimental and Quasi-Experimental Designs for Generalized Causal Inference. Houghton Mifflin Boston.
Dieste, O., Aranda, A.M., Uyaguari, F., Turhan, B., Tosun, A., Fucci, D., Oivo, M., Juristo, N., 2017. Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study. Empir. Softw. Eng. 22 (5), 2457–2542. http://dx.doi.org/10.1007/s10664-016-9471-3.
Erdogmus, H., Melnik, G., Jeffries, R., 2010. Test-driven development. In: Encyclopedia of Software Engineering. Taylor & Francis, pp. 1211–1229.
Erdogmus, H., Morisio, M., Torchiano, M., 2005. On the effectiveness of the test-first approach to programming. IEEE Trans. Softw. Eng. 31 (3), 226–237.
Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., Juristo, N., 2017. A dissection of the test-driven development process: Does it really matter to test-first or to test-last?. IEEE Trans. Softw. Eng. 43 (7), 597–614.
Fucci, D., Romano, S., Baldassarre, M.T., Caivano, D., Scanniello, G., Turhan, B., Juristo, N., 2018. A longitudinal cohort study on the retainment of test-driven development. In: Proceedings of International Symposium on Empirical Software Engineering and Measurement. In: ESEM '18, ACM, pp. 18:1–18:10. http://dx.doi.org/10.1145/3239235.3240502, Paper preprint: https://arxiv.org/pdf/1807.02971.pdf.
Fucci, D., Scanniello, G., Romano, S., Shepperd, M., Sigweni, B., Uyaguari, F., Turhan, B., Juristo, N., Oivo, M., 2016. 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. In: ESEM '16, ACM, pp. 3:1–3:10. http://dx.doi.org/10.1145/2961111.2962592.
Fucci, D., Turhan, B., 2013. A replicated experiment on the effectiveness of test-first development. In: Empirical Software Engineering and Measurement, 2013. IEEE, pp. 103–112.
Fucci, D., Turhan, B., Juristo, N., Dieste, O., Tosun-Misirli, A., Oivo, M., 2015. Towards an operationalization of test-driven development skills: An industrial empirical study. Inf. Softw. Technol. 68, 82–97.
George, B., Williams, L., 2004. A structured experiment of test-driven development. Inf. Softw. Technol. 46 (5), 337–342. http://dx.doi.org/10.1016/j.infsof.2003.09.011.
Höst, M., Regnell, B., Wohlin, C., 2000. Using students as subjects—A comparative study of students and professionals in lead-time impact assessment. Empir. Softw. Eng. 5 (3), 201–214.
Jedlitschka, A., Ciolkowski, M., Pfahl, D., 2008. Reporting experiments in software engineering. In: In Guide to Advanced Empirical Software Engineering. Springer, pp. 201–228.
Jeffries, R., Melnik, G., 2007. Guest editors' introduction: Tdd–the art of fearless programming. IEEE Softw. 24 (3), 24–30. http://dx.doi.org/10.1109/MS.2007.75.
Jorgensen, P.C., 2013. Software Testing: A Craftsman's Approach, fourth ed. Auerbach Publications.

Juristo, N., Moreno, A.M., 2001. Basics of Software Engineering Experimentation. Kluwer Academic Publishers.
Just, R., 2014. The major mutation framework: Efficient and scalable mutation analysis for java. In: Proceedings of the International Symposium on Software Testing and Analysis. In: ISSTA 2014, ACM, pp. 433–436. http://dx.doi.org/10.1145/2610384.2628053.
Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G., 2014. Are mutants a valid substitute for real faults in software testing?. In: Proceedings of International Symposium on Foundations of Software Engineering. In: FSE 2014, ACM, pp. 654–665. http://dx.doi.org/10.1145/2635868.2635929.
Karac, I., Turhan, B., 2018. What do we (really) know about test-driven development?. IEEE Softw. 35 (4), 81–85. http://dx.doi.org/10.1109/MS.2018.2801554.
Karac, E.I., Turhan, B., Juristo, N., 2019. A controlled experiment with novice developers on the impact of task description granularity on software quality in test-driven development. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2019.2920377.
Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., El Emam, K., Rosenberg, J., 2002. Preliminary guidelines for empirical research in software engineering. IEEE Trans. Softw. Eng. 28 (8), 721–734.
Kou, H., Johnson, P.M., Erdogmus, H., 2009. Operational definition and automated inference of test-driven development with zorro. Autom. Softw. Eng. 17 (1), 57. http://dx.doi.org/10.1007/s10515-009-0058-8.
Latorre, R., 2014. Effects of developer experience on learning and applying unit test-driven development. IEEE Trans. Softw. Eng. 40 (4), 381–395.
Marchenko, A., Abrahamsson, P., Ihme, T., 2009. Long-term effects of test-driven development a case study. In: Int. Conf. on Agile Processes and Extreme Programming in Software Engineering. Springer, pp. 13–22.
Müller, M.M., Höfer, A., 2007. The effect of experience on the test-driven development process. Empir. Softw. Eng. 12 (6), 593–615.
Munir, H., Moayyed, M., Petersen, K., 2014. Considering rigor and relevance when evaluating test driven development: A systematic review. Inf. Softw. Technol. 56 (4), 375–394.
Nagappan, N., Maximilien, E.M., Bhat, T., Williams, L., 2008. Realizing quality improvement through test driven development: Results and experiences of four industrial teams. Empir. Softw. Eng. 13 (3), 289–302. http://dx.doi.org/10.1007/s10664-008-9062-z.
Papadakis, M., Shin, D., Yoo, S., Bae, D.-H., 2018. 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. In: ICSE '18, ACM, pp. 537–548. http://dx.doi.org/10.1145/3180155.3180183.
Rafique, Y., Mišić, V.B., 2013. The effects of test-driven development on external quality and productivity: A meta-analysis. IEEE Trans. Softw. Eng. 39 (6), 835–856.
Romano, S., Fucci, D., Scanniello, G., Turhan, B., Juristo, N., 2016. 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. In: EASE '16, ACM, pp. 10:1–10:10. http://dx.doi.org/10.1145/2915970.2915996.
Romano, S., Fucci, D., Scanniello, G., Turhan, B., Juristo, N., 2017. Findings from a multi-method study on test-driven development. Inf. Softw. Technol. 89, 64–77. http://dx.doi.org/10.1016/j.infsof.2017.03.010.
Scanniello, G., Romano, S., Fucci, D., Turhan, B., Juristo, N., 2016. Students' and professionals' perceptions of test-driven development: A focus group study. In: Proceedings of Annual ACM Symposium on Applied Computing. In: SAC '16, ACM, New York, NY, USA, pp. 1422–1427. http://dx.doi.org/10.1145/2851613.2851778.
Shapiro, S., Wilk, M., 1965. An analysis of variance test for normality. Biometrika 52 (3–4), 591–611.
Shull, F., Melnik, G., Turhan, B., Layman, L., Diep, M., Erdogmus, H., 2010. What do we know about test-driven development?. IEEE Softw. 27 (6), 16–19.
Sillito, J., Murphy, G.C., De Volder, K., 2008. Asking and answering questions during a programming change task. IEEE Trans. Softw. Eng. 34 (4), 434–451. http://dx.doi.org/10.1109/TSE.2008.26.
Tosun, A., Dieste, O., Fucci, D., Vegas, S., Turhan, B., Erdogmus, H., Santos, A., Oivo, M., Toro, K., Jarvinen, J., Juristo, N., 2017. An industry experiment on the effects of test-driven development on external quality and productivity. Empir. Softw. Eng. 22 (6), 2763–2805.
Turhan, B., Layman, L., Diep, M., Erdogmus, H., Shull, F., 2010. How effective is test-driven development?. In: Media, O. (Ed.), Making Software: What Really Works, and Why We Believe It. pp. 207–219.
Vegas, S., Apa, C., Juristo, N., 2016. Crossover designs in software engineering experiments: Benefits and perils. IEEE Trans. Softw. Eng. 42 (2), 120–135.
Verbeke, G., Molenberghs, G., Rizopoulos, D., 2010. Random effects models for longitudinal data. In: Longitudinal Research with Latent Variables. Springer, pp. 37–96.
Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A., 2012. Experimentation in Software Engineering. Springer.
Yin, R.K., 2009. Case Study Research: Design and Methods (Applied Social Research Methods). Sage, London and Singapore.

Maria Teresa Baldassarre received her Laurea degree with honors in informatics at the University of Bari, Italy, where she has also received her PhD. She is currently assistant professor. Her research interests focus on: empirical software engineering, software quality assurance and human factors in software engineering. She is responsible for several international research collaborations. Partner of the SER&Practices spin-off company. Currently, she is representative of the University of Bari in the International Software Engineering Research Network (ISERN). She is also involved in various program committees related to software engineering and empirical software engineering.

Danilo Caivano is currently an Associate Professor of software engineering and project management with the Department of Computer Science, University of Bari Aldo Moro, and a Consultant for companies and organizations especially in the field of research and development projects. He is also the Head of the SERLAB Research Laboratory and the Director of the short master in cyber security. He contributed to the creation of The Hack Space, Cyber Security Laboratory, University of Bari. He is also a member of the Board of Director of the Southern Italy Chapter Project Management Institute, the Co-Ordinator of the PMISIC Academy, and a member of the Technical Scientific Committee of the Apulian Information Technology District, and the IT Strategic Steering Committee.

Davide Fucci is an Assistant Professor at Blekinge Institute of Technology (Sweden). He received his Ph.D. from the University of Oulu (Finland) in 2016. He has a strong background on empirical studies in software engineering, publishing and serving as committee member and reviewer for several venues in the field (e.g., ESEM, EMSE, IEEE TSE). Currently, his research interests lie in data-drive requirements engineering, test automation, and human aspects of software development. He started the AffectRE workshop series on emotional awareness in requirements engineering at RE and co-organizer of SEmotion'19 workshop at ICSE. He is involved with the Software Engineering ReThought project at BTH and with the H2020 OpenReq project at the University of Hamburg. More on: orcid.org/0000-0002-0679-4361.

Natalia Juristo (grise.upm.es/miembros/natalia) is full professor of software engineering with the Computing School at the Technical University of Madrid (UPM) since 1997. Natalia held a FiDiPro (Finland Distinguish Professor) research grant at University of Oulu from January 2013 to June 2018. She was the Director of the UPM M.Sc. in Software Engineering from 1992 to 2002 and the coordinator of the Erasmus Mundus European Master on SE (with the participation of the University of Bolzano, the University of Kaiserslautern and the University of Blekinge) from 2007 to 2012. Natalia will be General Chair for ICSE 2021 to be held in Madrid. She has served in several Program Committees ICSE, RE, REFSQ, ESEM, ISESE, and others. She has been Program Chair for EASE 2013, ISESE 2004 and SEKE 1997 and General Chair for ESEM 2007, SNPD 2002, and SEKE 2001. She has been member of several editorial boards, including TSE (Jan 2013–Dec 2017), EMSE (since 2002) and Software magazine (1997 to 2001) among others. Natalia has been guest editor of special issues in several journals, including EMSE, IEEE Software, JSS, DKE, and Int J Softw Eng Knowl Eng. Natalia has been ranked number 10 (among the experienced SE researchers) in a paper published in January 2019 at JSS that evaluates the 2010–2017 period.

Simone Romano received his master's degree in Computer Engineering from the University of Basilicata, Italy, in 2014 and then the Ph.D. in Computer Science from the University of Salento, Italy (in collaboration with the University of Basilicata) in 2018. He then joined the Department of Informatics at the University of Bari, where he is currently a postdoctoral research fellow. He has served in the organization and has been a program committee member of different conferences such as ESEM, ICPC, SEAA, and PROFES. His research interests include: software refactoring, software testing, test-driven development, empirical software engineering, and human factors in software engineering.

Giuseppe Scanniello: received his Laurea and Ph.D. degrees, both in Computer Science, from the University of Salerno, Italy, in 2001 and 2003, respectively. In 2006, he joined, as an Assistant Professor, the Department of Mathematics and Computer Science at the University of Basilicata, Potenza, Italy. In 2015, he became an Associate Professor at the same university. His research interests include requirements engineering, empirical software engineering, reverse engineering, reengineering, software visualization, workflow automation, migration, wrapping, integration, testing, green software engineering, global software engineering, cooperative supports for software engineering, visual languages and e-learning. He has published more than 160 referred papers in journals, books, and conference proceedings. He serves on the organizing of major international conferences (as general chair, program co-chair, proceedings chair, and member of the program committee) and workshops in the field of software engineering (e.g., ICSE, ASE, ICSME, ICPC, SANER, and many others). Giuseppe Scanniello leads both the group and the laboratory of software engineering at the University of Basilicata (BASELab). He recently obtained the Italian National Scientific Qualification as Full Professor in Computer Science. He is a member of

IEEE and IEEE Computer Society. More on: sites.google.com/view/prof-giuseppe-scanniello/home.

Burak Turhan is an Associate Professor in Cyber Security & Software Systems at Monash University. His research focuses on empirical software engineering, software analytics, quality assurance and testing, human factors, (agile) development processes, and digital health. Dr. Turhan has published over 100 articles in international journals and conferences, received several best paper awards, and secured several large-scale external research grants. He has served on the program committees of over 30 academic conferences, on the editorial or review boards of several top-tier software engineering journals, and as (co-)chair for PROMISE'13, ESEM'17, and PROFES'17 conferences. He is a member of ACM, ACM SIGSOFT, IEEE and IEEE Computer Society. For more information please visit: turhanb.net.