Findings from a multi-method study on test-driven development

Simone Romano [a,*], Davide Fucci [b], Giuseppe Scanniello [a], Burak Turhan [b], Natalia Juristo [c]

[a] University of Basilicata, Viale Dell'Ateneo 10, Macchia Romana, Potenza, Italy
[b] M3S, University of Oulu, Pentti Kaiteran katu 1, Oulu, Finland
[c] Facultad de Informatica, Universidad Politecnica de Madrid, Campus de Montegancedo, 28660 Boadilla del Monte, Madrid, Spain

A B S T R A C T

Context: Test-driven development (TDD) is an iterative software development practice where unit tests are defined before production code. A number of quantitative empirical investigations have been con- ducted about this practice. The results are contrasting and inconclusive. In addition, previous studies fail to analyze the values, beliefs, and assumptions that inform and shape TDD.
Objective: We present a study designed, and conducted to understand the values, beliefs, and assump- tions about TDD. Participants were novice and professional software developers.
Method: We conducted an ethnographically-informed study with 14 novice software developers, i.e., graduate students in Computer Science at the University of Basilicata, and six professional software de- velopers (with one to 10 years work experience). The participants worked on the implementation of a new feature for an existing software written in Java. We immersed ourselves in the context of our study. We collected qualitative information by means of audio recordings, contemporaneous field notes, and other kinds of artifacts. We collected quantitative data from the integrated development environment to support or refute the ethnography results.
Results: The main insights of our study can be summarized as follows: *(i)* refactoring (one of the phases of TDD) is not performed as often as the process requires and it is considered less important than other phases, *(ii)* the most important phase is implementation, *(iii)* unit tests are almost never up-to-date, and *(iv)* participants first build in their mind a sort of model of the source code to be implemented and only then write test cases. The analysis of the quantitative data supported the following qualitative findings: *(i), (iii),* and *(iv).*
Conclusions: Developers write quick-and-dirty production code to pass the tests, do not update their tests often, and ignore refactoring.

1. Introduction

Test-driven development (TDD) is an iterative software development practice within agile methodologies [1]. It requires developers to follow three phases: The *red phase* causes a shift in mindset from the test-last approach to the test-first approach, while the *green phase* only develops enough code to pass the tests and the *refactoring phase* focuses on design quality through refactoring operations and uses a set of regression test cases as a safety net. It is claimed that TDD leads to better code quality due to its focus on testing, and improves developers' confidence in their source code [2]. Based on this claim, some software organizations have been quick to adopt TDD, while others are still evaluating its benefits in terms of cost, quality, and productivity [3,4].

To assess TDD, a number of primary (e.g., controlled- and quasi-experiments) and secondary (e.g., systematic literature reviews) empirical studies have been conducted. Primary studies (e.g., [5,6]) have been quantitative in nature and have produced contrasting or inconclusive results [7]. The secondary studies summarize the empirical research results regarding TDD by aggregating, to a varying extent, the evidence from controlled experiments, quasi-experiments, and case studies [3,4,7,8].

TDD has been marginally investigated from a qualitative point of view and from the perspective of the developer [9,10]. Qualitative studies, unlike quantitative ones, inquire into the underlying reasons and motivations behind a given phenomenon [11]. Among the kinds of qualitative methodological approaches, an

* Corresponding Author.
E-mail addresses: simone.romano@unibas.it (S. Romano), davide.fucci@oulu.fi (D. Fucci), giuseppe.scanniello@unibas.it (G. Scanniello), Burak.Turhan@oulu.fi (B. Turhan), natalia@fi.upm.es (N. Juristo).

ethnographically-informed study forces researchers to attend to the taken-for-granted, accepted, and un-remarked aspects of a practice, considering all activities as "strange" so as to prevent the researchers' own backgrounds from affecting their observations [12]. In this type of study, researchers immerse themselves in the study context, participate in the study (e.g., by joining in conversations, attending meetings, reading documents), and observe participants without prejudice or prior assumptions [13].

In this paper, we present the results of an empirical study involving students and professional software developers. We involved 14 graduate students in Computer Science at the University of Basilicata, and six professional developers with one to 10 years' work experience. We asked participants to work in pairs, each pair is composed of a driver and a pointer developer. The goal of our study is to gain insights into how developers apply TDD and deal with each of its phases. In particular, we sought to explore the values, beliefs, and assumptions that inform and shape the application of TDD and its phases. Given this motivation, our methodological approach can be characterized as ethnographic [14–16]. We asked the participants to perform an implementation task. In particular, they had to add new functionality to an existing software implemented in Java using Eclipse. This software is a complex, industrial-like case of which participants had some knowledge. The first author immersed himself in the study environment, participated in conversations, and asked the participants how they were applying TDD to perform the assigned implementation task. We collected information by means of contemporaneous field notes, audio recordings of discussions, and copies of the artifacts produced by the participants during the implementation. Fine-grained data about the participants' application of TDD was also gathered through an automated tool installed in the participants' integrated development environment (IDE). This tool is intended as an Eclipse plug-in and runs in the background without interfering with the IDE use. We analyzed TDD conformance data to support or reject the qualitative findings. This is why we consider this as a multi-method study.

This study builds on a previous study [17] in the following ways:

- We described our ethnographically-informed study with more details.
- We extended the related work section by including a review of quantitative studies relevant for the approach we used to triangulate the results.
- We triangulated the initial qualitative results with analyses of quantitative data. We cross-referenced the results of both analyses and improved the discussion of the attained outcomes and conclusions.

In summary, we make the following contributions:

- We present the implications of the results from an ethnographically-informed study with nine pairs of developers; three of these pairs were professionals. To the best of our knowledge this is the first of such studies explicitly tackling TDD.
- We provide a triangulation of the results based on quantitative data extracted from the developers' IDE.
- We delineate future research to investigate the specificities behind the application of TDD.

The remainder of the paper is organized as follows. In Section 2, we discuss related work. In Section 3, we explain our method, while in Section 4, we present our findings. We show and discuss our findings in Section 5. In Section 6, we highlight limitations of these findings. Final remarks and future work conclude the paper.

2. Related work

In this section, we first discuss ethnographically-informed studies in software engineering and papers reporting quantitative and qualitative investigations on TDD.

2.1. Ethnographically-informed studies in software engineering

Ethnography is a qualitative research method for studying people and cultures. This method is largely adopted in disciplines outside software engineering [14]. However, the importance of ethnography, and the challenges associated with adoption from social sciences have been tackled in other areas of computer science, like computer-supported cooperative work [18]. In other subfields—e.g., software system design and interaction—ethnography is becoming progressively relevant [19]. In the context of software engineering, ethnography could provide an in-depth understanding of the socio-technological realities surrounding everyday software development practices [16]. That is, ethnography could help to uncover not only what practitioners do, but also why they do it. Despite its potential, little ethnographic research exists in the field of software engineering [16]. For example, Beynon-Davies [20] identified a number of uses of uses for ethnographic research in information systems development. In particular, the author noted that for researchers in the software engineering field, ethnographic research may provide value in the area of software development, specifically in the process of capturing tacit knowledge during the software life cycle. Later, Beynon-Davies *et al.* [21] used ethnographic research on rapid application development to uncover the negotiated order of work in a project and the role of collective memory.

Button and Sharrock [22] carried out an ethnographically-informed study on global software development to explain the knowledge that is displayed in the collaborative actions and interactions of design and development. Sharp and Robinson [23] reported on a study of eXtreme Programming carried out in a small company developing web-based intelligent advertisements. The main result being that the XP developers were clearly "agile." This agility seemed intimately related to the relaxed, competent atmosphere that pervaded the developers working in groups.

Singer *et al.* [24] found that software engineers maintain a large telecommunications system through habits and tool usage during software development. This study was hosted in a single company. Despite the software engineers stating that "reading documentation" was what they did, the study found that searching and looking at source code was much more common than looking at documentation. This case shows there is a difference between what practitioners say they do and what they actually do. Ethnographic research might help in highlighting and explaining such discrepancies to make clearer un-remarked aspects of practice [12].

Further, Salviulo and Scanniello [25] conducted an ethnographically informed study with students and professionals to understand the role of comments and identifiers in source code comprehensibility and maintainability. Authors observed the following outcomes: *(i)* professional developers (as compared with students) prefer to deal with source code and identifiers rather than comments, *(ii)* all participants (professionals and students) believed that the use of naming convention techniques when writing identifiers was essential, and *(iii)* all participants stated that the names of identifiers are important and developers should properly choose them.

2.2. Studies on TDD

Although ethnography has been used to study how testing is done within software companies (e.g., [26,27]), there is a lack of

studies regarding specifically TDD and unit testing. Despite being among the core XP methodologies, Sharp and Robinson [23] did not explicitly address them in their ethnography.

A number of quantitative studies have assessed the effectiveness of TDD; the results of these studies have been examined in a number of systematic reviews and meta-analyses [7,28,29]. The results of these analyses have been contradictory regarding the effects on both software products (e.g., defects) and developers (e.g., productivity). Interestingly, one of the secondary studies [3] suggested that *insufficient adherence to the TDD protocol* and *insufficient testing skills* are among the factors hampering industrial adoption of TDD.

Only a few studies have focused on the developer's perception of the TDD practice. For example, Müller and Tichy [30] examined several Agile methodologies, including TDD, within a university course and found that TDD was one of the most difficult practices to adopt because developers felt that it was impractical to write test cases before coding. On the other hand, Gupta and Jalote [31] reported, in a controlled experiment with students, that TDD improved developer's confidence in writing code, with respect to a test-last approach. However, developers needed more upfront design. Moreover, the students perceived that TDD improved their effort when testing as compared to a traditional test-after-code setting (test-last approach).

Pancur et al. [32] reported that students perceived TDD as more difficult to adopt with respect to professionals. In particular, students perceived TDD as a practice that hindered their productivity, efficiency, and the quality of their code. Both students and professionals agreed that TDD helped in devising a better design and prevented bugs, but they also believed that this practice could not replace a quality assurance engineer [7]. Even more interesting, participants also believed that the use of TDD improved confidence by minimizing the fear that existing parts of well-functioning source code would be compromised by the implementation of new features [33].

The current study involved collecting quantitative data from developers IDE. Previous research leverages this type of data in the context of TDD. Conformance—i.e., the extent to which the developers apply the process was an important aspect of this study and was inferred through different heuristics Fucci and Turhan [34] investigated the goodness of process conformance as a predictor of the final quality of source code, as well as the developer's productivity using data from a study with graduate students. Although no significant correlations were found, the subjects achieved an average conformance of 78% (sd = 20), showing that students, already after a short training period, are able to follow the process to a good extent. The participants in our study were trained using a similar material, and over a similar time span as in the study presented in Fucci and Turhan [34].

A study with professionals using TDD [35] leveraged process conformance, along with other metrics related to the developer' skills, to investigate the impact of the practice on the quality of the software as well as the developer' productivity. In industrial settings, the authors showed that developer' conformance was close to 75%; however, 13% of the time the process was not followed at all. A qualitative approach, like the one followed in the ethnography part of this study, can help in revealing the root causes for such behavior.

There are other approaches leveraging low level data collected from the IDE used in large-scale field studies. For example, the work reported by Beller *et al.* [36] proposed a nondeterministic finite state machine to infer TDD development. The authors show that in a study of 461 professional developers, TDD was strictly used by only 4%, while refactoring was the predominant phase occupying 72% of the overall TDD process. Another approach, presented by Hilton *et al.* [37] leveraged code changes between snapshots of the abstract syntax tree. The authors created a tool to visualize the progress of TDD development activities over time as a feedback for practitioners to improve or tune their process. The tool was validated using a dataset of 2601 Java development sessions, yielding an accuracy, measured through F-measure, of 87%. A group of 35 practitioners was able to improve their ability to identify and comprehend their TDD process when using the tool.

This study employs the same tool, Besouro [1] to provide access to more fine-grained development activities for calculating the process conformance metric in previous work (i.e., [34,35]).

Quantitative studies help reveal possible relations among the constructs underlying TDD. However, more in-depth observation and questioning of the practice can be gathered using qualitative approaches, like ethnographies.

While quantitative studies provide objective frameworks for assessing a practice, qualitative studies may enable a deeper understanding of its use. For example, Heikkila *et al.* [38] used a similar multi-method approach to assess the discrepancies between Scrum practices, and their actual application in industry. Existing studies have relied upon non-interactive research methods, such as questionnaires. In quantitative studies, TDD is often compared to a test-last approach (e.g., [10]). In the present study, we employed an ethnographic approach in order to develop a better understanding of TDD, the underlying phenomena, and developer perceptions thereof. We are not interested in comparing TDD to other development techniques or practices. Furthermore, we included both students and professionals in our study, since previous work has found that perceptions of TDD vary between these two groups (e.g., [6]).

3. Our ethnographic study

Qualitative studies are unusual in the software engineering field, where research is dominated by quantitative studies that test hypotheses by means of laboratory experiments [39–41]. In this kind of study, participants are generally split into treatment and control groups, and statistical analysis of gathered data is performed to verify the presence of a statistically significant difference among these groups with respect to a given construct [11]. Despite the dominance of quantitative studies, qualitative studies are considered a necessary complement [42], essential for gaining an understanding of the reasons and the motivations behind the problem under study. Qualitative methods, and in particular ethnography, can motivate and complement quantitative ones by apprising different facets of the software development processes, as well as the methods and tools used [15].

Ethnographic studies are better suited to ask questions such as *how, why* and *what are the characteristics of* Robinson et al. [43]. To this end, researchers attend to the taken-for-granted, accepted, and un-remarked aspects of practice, considering all activities as "strange" so as to prevent the researchers' own backgrounds from affecting their observations [12]. Usually, ethnographic studies are conducted on a small number of subjects, while researchers immerse themselves in participants environment [23]. In some fields of research, such as software engineering, this practice is not always possible (e.g., because of time constraints). In such cases, it is common to adapt ethnographic methods of a shorter timeframe (e.g., [25,43]).

The goal of an ethnographic analysis is to find insights from recurrent themes in the participants activities. The meaning behind the observed activities must be inferred from the details of the collected data [23]. To this end, the steps we performed can be summarized as follows:

1. The observer first reflected upon the experience gained in the immersion and used all of the data to recollect, revisit, and reconsider what was found.
2. The observer and the other researchers discussed the audio recordings, the source code written by the participants, and other artifacts such as the data collected by the IDE plug-in.
3. When a theme appeared to be emerging in a group (i.e., students and/or professional developers), we searched for data in the same group that could contradict this theme. If no contradictory evidence emerged, the theme was pursued.
4. When a theme was pursued in a group, we searched for data in the other group that could contradict this theme. If no contradictory evidence emerged then the theme was pursued also in the other group. That is, the theme holds in both the groups.

This kind of analysis proceeded iteratively as themes were identified, dropped, or validated and then confirmed.

3.1. Definition and context

TDD is an iterative software development practice in which unit tests are defined before production code [1]. Developers repeat short cycles consisting of the following phases:

red - writing a unit test for an unimplemented functionality or behavior;
green - supplying the minimal amount of production code to make unit tests pass and;
refactoring - applying refactoring where necessary, and checking that all tests are still passing.

In the software industry, pair-programming is often used with TDD [44,45]. In our study, we focused on programmer pairs instead of teams or organizations. In this setting, we were interested in exploring the following topics:

How practitioners and novice programmers perceive TDD.
How they approach each phase of TDD.
Why they adhere (or do not adhere) to TDD.
Why they feel more comfortable with one or another of the TDD phases.
How refactoring is carried out in TDD.
Which characteristics the developers believe an application must have so that they can successfully apply TDD.

The participants in our study were 14 graduate students in Computer Science at the University of Basilicata and six professional software developers taking a specialization course at the same university. Both professionals and students were familiar with TLD (test-last development)—i.e., a development technique where unit tests are written after a feature (or a set of related features) is implemented for a given software. This occurs in a traditional approach to software development.

Using graduate students as participants is not a major issue because in the literature (e.g., [46,47]) they are considered not far from novice software developers (see Section 6). In addition, a comparison of these students with professional developers would help us to better understand whether, and under which conditions, graduate students can be considered similar to novice professional developers [48,49]. Such a comparison is not the main point, but was reported as it represents an additional contribution of our study.

The professional developers participating in our study worked in different small/medium sized companies located in Italy. The most experienced among the professionals held a master's degrees in Computer Science, while others held bachelor's degrees in Computer Science. One pair (DG) had in total three years of professional development experience, whereas the other two pairs (RB and ZP) had nine and 13 years of experience respectively. Professionals had knowledge of testing approaches, and techniques (e.g., unit testing, integration testing, and system testing). For the professionals, the course in which this study is embedded represented an on-the-job training about agile software development sponsored by their companies in collaboration with the university. The lecturer devoted the greater part of the course to the introduction of TDD, and its application to real cases. The course lasted for eight weeks (with four hours of frontal instructions per week) and included both homework and classwork.

The students participated in our ethnographically-informed study as part of a series of laboratory exercises conducted within an Information System (IS) course. This course covered elements of software testing, software development, software maintenance, and agile development techniques with a focus on TDD, XP, regression testing, and refactoring. Homework and classwork provided students with the opportunity to practice TDD, regression testing, and a testing framework (i.e., JUnit[1]). Java was the programming language of reference used throughout the class, for both homework and the classwork. Before participating in the study, the students had passed the following courses: Procedural Programming, Software Engineering I, Object-Oriented Programming I and II, and Databases. The students had knowledge regarding the development of object-oriented software systems and/or web-based applications. Students prior knowledge of the IS course can be considered homogeneous. The same lecturer held the specialization course for professionals and the IS course for students

Both pairs of students and professionals worked on MusicPhone—a Java application running on GPS-enabled devices. MusicPhone gives users artist recommendations, and finds upcoming concerts for these artists. This application was primarily chosen for the availability of its source code and because it was used in previous empirical studies on TDD (e.g., [6,50]). When using an incremental approach, such as that adopted in agile development, MusicPhone also represents a good compromise of generality and industrial application at the first stages of development. The total number of classes in the MusicPhone application was 30, while the non-commented lines of source code were 1225. The number of methods and constructors was 157 and 22, respectively. These statistics refer to MusicPhone before pairs worked on it. In particular, we asked pairs to implement *Compute an itinerary for artists*, described as follows:

An itinerary is a list of destinations, where each destination contains artist's concert information and distance to the concert's location from the previous destination. The first destination's distance in the list is the distance from user's current position. The itinerary must be chronologically ordered according to the start date of the concerts in it.

This feature was described by means of a user card (or simply card, from here on) and its confirmations. Confirmations summarize conversations among stakeholders (e.g., constraints of a functionality for the software under development) [51]. In this sense, confirmations are an acceptance test for a story. We opted for a card and its confirmations because a traditional card is a very high-level definition of a requirement that contains just enough information for the developers to reasonably estimate the effort required to implement it. That is, a card contains little information for the implementation of a feature. In agile methodologies, it is customary to flesh out a card (e.g., during the brainstorming with stakeholders) when it has to be implemented. This was why we provided participants with a card with *confirmations*. The story card used in our study is shown in Fig. 1 (top). It contains four confirmations (on the bottom) that allow the developer to better

[1] http://junit.org.

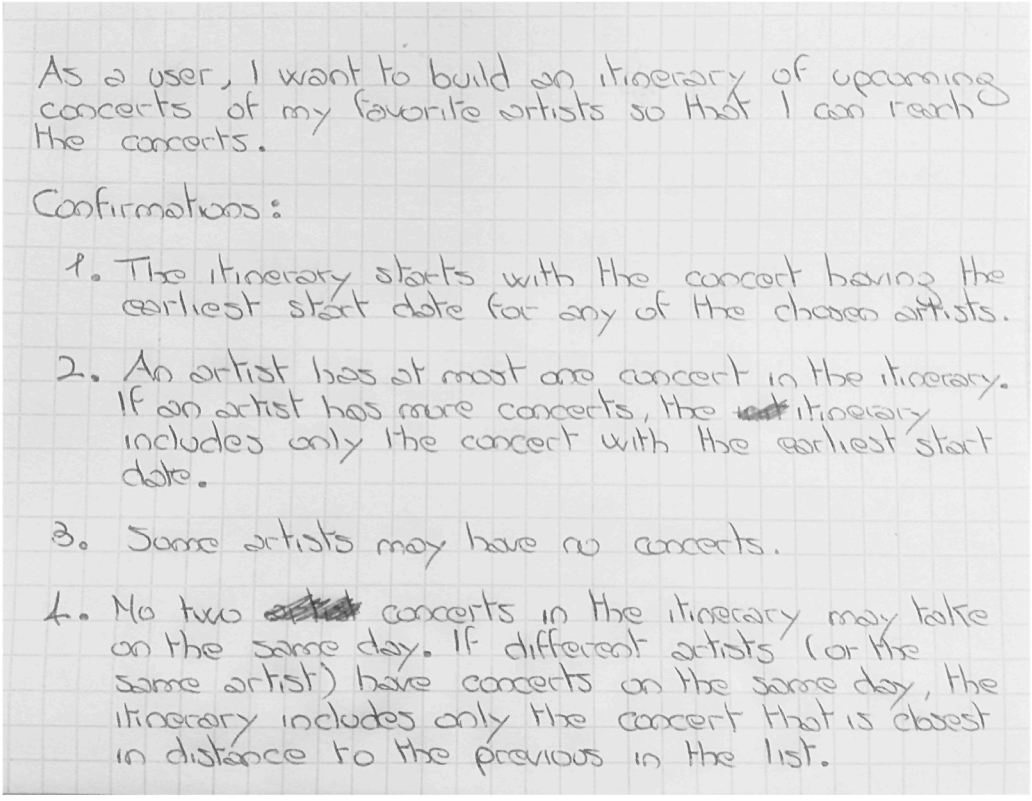

Fig. 1. Story card used in the study.

understand the feature to be implemented and constraints to be tackled.

MusicPhone can be considered a legacy system since its codebase is not covered by tests [52]. The source code was scarcely commented (56 comment lines in total), as is often the case in the context of agile software development, with the goal to produce clean code and working software rather than thorough documentation [53]. In fact, comments are used when the code does not clearly communicate its intent—i.e., comments are used as a substitute for bad code (e.g., [54]). We provided participants MusicPhone architecture documentation. Using such a kind of documentation in agile projects is common in order to avoid *big design upfront* [55]. Both source code comments and documentation were written in the English language.

The participants were familiar with the problem domain of MusicPhone. The lecturer used other parts of this application for homework and classwork when introducing TDD. Therefore, knowledge regarding MusicPhone application features can be considered homogenous. However, pairs were not familiar with the codebase used in the study.

The participants were volunteers; that is, we neither paid students and professionals for taking part in the ethnographically-informed study, nor did we force them. Participants did not know the goals of the study.

3.2. The setting

For the scope of this ethnography, we kept as close as possible to the natural settings in which the developers, working in pairs, would carry on their everyday work activities. The setting could be relevant insofar as the participants work to accomplish a task [23].

Describing the setting is a good practice in ethnographically-informed studies [23]. Fig. 2 shows a pair of participants in the physical settings where they worked on the task. The participants worked on MusicPhone following a fixed schedule. Only the study observer and the pair were present each time. Each pair used the same laptop to carry out the task, and each pair worked during regular working hours. These measures were taken to minimize as much as possible the differences in the study settings.

Fig. 2. A pair of professional developers.

3.3. The study

The study was conducted at the end of the specialization course for professionals and the IS course for students. A single observer (the first author of this paper) conducted this study between May and July 2015. It was based on one-to-one sessions between the observer and each pair. The use of one-to-one sessions is custom-

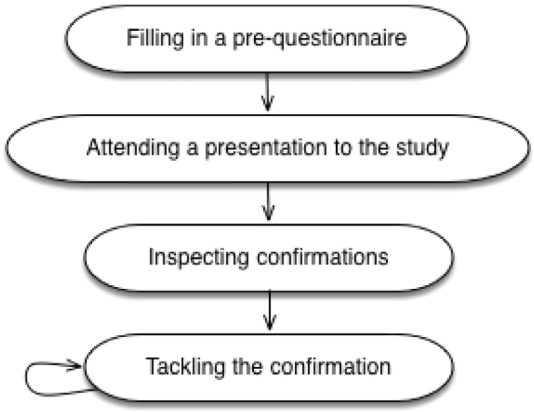

Fig. 3. Steps participants accomplished in the study.

ary in ethnographically-informed studies (e.g., [43]). We conducted our study in Italian to minimize any bias arising from participants' varying levels of familiarity with the English language.

For each session, the observer spent more than three hours working with each pair. As mentioned before, all the pairs had some familiarity with MusicPhone. This scenario is not unlikely in the software industry, where developers often do not exactly know all the source code of a given application, but are familiar with certain parts of it.

The observer, if needed, engaged with the pairs by focusing on both the application, and solution domains of MusicPhone. While interacting with the pairs, the observer did not comment on their working practices and avoided influencing them. However, the interaction between the pairs and the observer is critical because: *(i)* the ethnographic methodological approach encourages participation [12]; *(ii)* it provides the observer with opportunities to appreciate the perspective of developers while they carry out their tasks; and *(iii)* it provides the observer with opportunities to gather information on how a method is applied (TDD, in our case). Data were collected in a variety of forms: contemporaneous field notes, audio recordings of discussions, and copies of various artifacts (e.g., source code and notes). To implement the new feature for MusicPhone, pairs used Eclipse as the IDE. Besouro,[2]—an Eclipse plug-in capable of tracing how developers applied TDD—was installed in the IDEs. This plug-in runs in the background and does not interfere with the regular use of Eclipse.

3.4. Design

We used the design presented in Fig. 3. In particular, we asked the pairs to complete the following steps in sequential fashion:

1) **Filling in a pre-questionnaire:** Each participant was asked to individually fill in a pre-questionnaire. The goal of this questionnaire was to gather information about participants' experience, grade point average, and relevant knowledge necessary for the study. We used this information to get further information on study context (see Section 3.1). The gathered information was also used to select the IDE. Since all the participants had a good level of familiarity with Eclipse, we used it in our study.
2) **Attending a presentation to the study:** The observer introduced the study to each pair, who used a prearranged schema, namely a few sentences to describe what each pair and participant had to do. The observer provided details on neither research topics of interest, nor the objective of the study. At the end of this step, participants could ask questions for clarification.
3) **Inspecting confirmations:** We asked the pairs to carefully read the card, and its confirmations for *Compute an itinerary for concerts*. In particular, pairs had to work on a new feature divided into four confirmations (see Fig. 1). These confirmations were presented in logical order. That is, a pair had to tackle a confirmation before passing to the subsequent one. Each confirmation required the implementation of one or more test cases.
4) **Tackling the confirmation.** We did not suggest any strategy to deal with each confirmation. For example, pairs could freely decide to inspect the entire source code before or after having defined test cases. The observer appointed neither the driver[3] nor the pointer[4] (or navigator) developers; instead, each pair freely chose who the driver and the pointer were. During the implementation, the roles could be swapped.

The observer could provide support for steps 1, 2, and 3 and could clarify concerns related to the study and/or to questions of the pre-questionnaire. On the other hand, the observer become immersed and participated in step 4, joining in conversations and reading the card and its confirmations (to clarify them, if necessary) without disturbing or change the natural setting of the study and while using an informal approach to probe possible issues [56].

4. Findings

In this section, we first present the findings from our ethnographic analysis following a standard approach (i.e., [21]). In particular, we identified the following six themes emerging from our data: *(i)* comprehending legacy code; *(ii)* discussion on the card; *(iii)* TDD phases; *(iv)* trusting test cases; *(v)* TDD Conformance; and *(vi)* working in pairs.

Subsequently, we present findings from a quantitative analysis of the data gathered by Besouro. In particular, we triangulated the ethnography results on pairs' behavior with quantitative data. In the following subsections, we illustrate and detail each theme.

4.1. Comprehending legacy code

The pairs took some time to comprehend the legacy source code before tackling a given confirmation. Both professionals and students behaved similarly. In particular, participants first read a confirmation and then browsed source code to specify the tests needed for that confirmation. In the following example, one pair of professionals, while reading a confirmation, inspected the source code to re-use an existing method:

> *A: In order to find the starting point of the itinerary we should have a look at the Recommender class, I spotted a method we must re-utilize.*
> *B: Let's find it!*

This process of understanding source code was time consuming. However, with each successive confirmation, the pair became more familiar with the MusicPhone source code and thus spent less time working through it. Nevertheless, most of the pairs did not use unit tests to understand existing source code in an explorative manner, but rather relied on visual inspections.

4.2. Discussion on the card

The pairs preferred to discuss an implementation plan for a given confirmation before passing to the subsequent phase of de-

[2] Besouro - https://github.com/brunopedroso/besouro.

[3] The developer in charge of writing code.

[4] The developer in charge of reviewing each line of code as it is typed in.

velopment. The discussion regarded the implementation details rather than the definition of test cases and the identification of refactoring possibilities. For example, one pair of professionals had the following discussion regarding a confirmation:

A: We need to fetch each artist's destinations list, that should be an ArrayList of Destination objects... Do you know how to use an ArrayList?
B: Should we define a test first?
A: First let's find out how to work with an ArrayList.

Often the implementation is envisioned using information such as syntactical structure and the control and data flow of existing source code [57]. In our case, it seemed that participants first built a sort of mental model of the source code to be implemented, and only then wrote test cases. In theory, a developer should be able to define a test only by imagining the interface of the code to be implemented rather than its specific implementation.

4.3. TDD phases

The pairs preferred the green phase because it involved writing production code and the *rewarding* green bar. However, production code was often not entirely covered by the test case defined in the red phase. This violates one of the principles of TDD, which states—"Write only enough [code] to pass the test, no more" [2]. In other words, more production code than necessary was often written.

Pairs were unmotivated during the red phase. They found it difficult to define tests before production code for two reasons: *(i)* defining an oracle implied the difficulty of imagining a concrete scenario under which to test system and *(ii)* arranging the data (e.g., instantiating objects and preparing them) necessary to execute the JUnit asserts.

A: I am pretty sure we need to write a test to calculate the distances between points.
B: Any idea where to find the expected output distance between point A and B?
A: We could get such information by logging the execution.

This is in line with what was shown in Section 4.2. Such difficulty was less remarked for professional developers. The students were not able to adequately define test cases of the correct granularity. It is possible that students were less capable than professionals to imagine the source code to be implemented. This difference could be also due to the way students dealt with the card and its confirmations.

Refactoring was perceived as a risky undertaking; therefore the pairs performed refactoring only when forced to.

A: Here I realize I should do something about this method. If I refactor it now it is going to take forever.
B: If we were to touch anything here, I think it is gonna mess-up our code.
A: Yes, but we should do something about it later.

The pairs performed refactoring only when it was necessary and when all the confirmations were taken into account. In this sense, refactoring was performed mostly when the previously written code was needed for the implementation of the next confirmation. As for refactoring, it was very often skipped because the pairs considered a development cycle (i.e., the sequence of phases in TDD) already completed when the production code was written and all tests were passed (i.e., the green phase).

4.4. Trusting test cases

Test cases were considered a gold standard: written test cases were never modified or updated. That is to say, the pairs believed that the test cases they originally wrote were always correct even though passing a single test case did not imply that the production code was correct.

A: All tests are passing! I guess we are done.
B: Yes, this confirmation seems completed to me.

In addition, tests were never modified in accordance with the evolution of the source code.

4.5. TDD conformance

We observed that TDD was not applied in two cases. In the first case, pairs wrote source code to deal with a confirmation and then moved to the implementation of a new confirmation before concluding the implementation of the former. In the second case, we observed that conformance degree to TDD gradually decreased from the implementation of a confirmation to the next one. We also noted that pairs wrote more production code than was strictly necessary for the implementation of a confirmation (see Section 4.3). The pairs were often getting ahead of themselves by adding source code to implement the next confirmation, before finishing the current one.

4.6. Working in pairs

As customary, the drivers in pair-programming were in charge of approaching problems and implementing solutions, while the pointers verified that the drivers did not make any mistake. We observed a slight difference between professionals and students when working in pairs. Professionals in the role of pointer actively participated in problem solving activities related to the source code, and the implementation of necessary test cases. Student pointers held aloof from such activities, focusing exclusively on their job of verifying that the driver did not make any mistake while dealing with the card and its confirmations.

A: There is an error because we have to implement Comparable to sort this list of objects.
B: Ok! Let me modify the code.

4.7. Additional findings

In general, the only remarkable difference observed between the novice and professional developers was the way they divided a card into a set of sub-tasks. Professionals incorporated a finer grain implementation of the confirmation with respect to the students. However, the professionals adopted the same approach as the students when refactoring.

4.8. Triangulating qualitative results with quantitative data

To better understand the themes, we also analyzed quantitative data Besouro gathered. This Eclipse plug-in identified different types of activities performed by developers (defined in Table 1) on the heuristics defined in Kou et al. [58]. Besouro also gathered the duration of each performed activity. For example, *TL* indicates a test-last cycle—e.g., a developer writes some production code, which is then unit tested. The tool also creates a local git repository in which changes to the files are committed every time an activity is recognized.

In Table 2, we report the themes from our ethnographically-informed analysis, and whether or not they have been confirmed in our quantitative data analysis. For the remainder of this section, we present the obtained quantitative results in connection with the qualitative results. We conclude presenting additional findings from quantitative data.

Table 1
Types of activities recognized within the IDE.

Type	Description
TF	Test-first activity in which production code is written once a test passed.
TL	Test-last activity in which production code is written before a test passed.
TA	Test addition activity in which a new test is added to existing production code, and passed.
PR	Production code activity in which production code is added without the accompanying test.
RG	Regression testing activity in which tests are run but no new code (test or production) is added.
RF	Refactoring activity in which production code is modified and then passes its associated test.

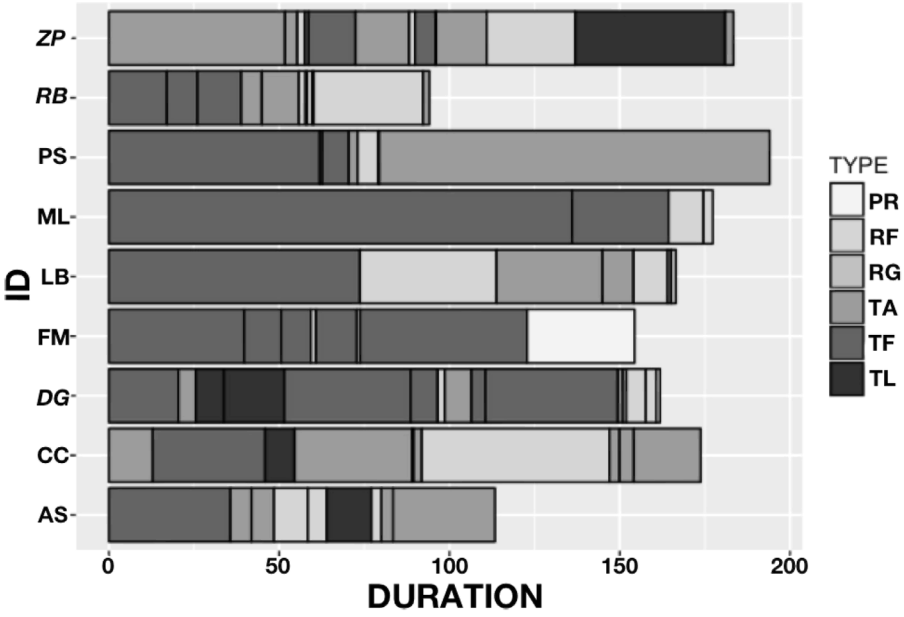

Fig. 4. Development cycles for each pairs (professional pairs ZP, RB, and DG in italics).

Table 2
Summary of triangulated results.

Ethnographic theme	Quantitatively supported
Comprehending legacy code (Section 4.1)	Yes (Section 4.8.1)
Discussion on the card (Section 4.2)	Not applicable
TDD phases (Section 4.3)	Yes (Section 4.8.2)
Trusting test cases (Section 4.4)	Yes (Section 4.8.3)
TDD conformance (Section 4.5)	Yes (Section 4.8.4)
Working in pairs (Section 4.6)	Not applicable

Table 3
Descriptive statistics for the duration (in minutes) of each type of development activity.

Type	n	Total	Mean	Median	Min	Max	Stddev
TF	22	663.82	30.17	18.73	4.12	136.03	30.42
TA	27	403.74	14.95	6.18	1.07	114.63	23.48
RG	6	4.36	0.73	0.41	0.23	1.58	0.61
RF	23	223.43	9.71	2.97	0.55	55.13	14.52
PR	1	31.58	31.58	31.58	31.58	31.58	–
TL	5	92.00	18.4	13.18	8.27	44.07	14.86

4.8.1. Comprehending legacy code

Table 3 reports the descriptive statistics for the duration of each type of development activity recognized during the ethnography study. In terms of total duration, the *TF* activity was predominant. When the activities are visualized in temporal order (i.e., in Fig. 4), the majority of the pairs had a *cold start*. The duration of the initial activity was close to 50 min, indicating that pairs needed time to familiarize themselves with the legacy code before being able to progress. Two pairs (CC and ZP) decided to write unit tests for the existing legacy code (i.e., *TA* activity).

4.8.2. TDD phases

In Section 4.3, we reported that refactoring was not performed as often as other activities. Fig. 4 suggests that refactoring was performed in an inconsistent manner with the expected TDD flow. In Fig. 4, the different types of activities identified by Besouro for each pair were plotted on a temporal dimension. Therefore, each bar corresponds to a pair. The width of the colored boxes constituting a bar represents the duration of the development activity, while the color corresponds to the type of activity. In this way we can have a visual representation of the process. From Fig. 4 it can be observed that the refactoring activities were executed *in bulk*, rather than intertwined with unit testing and development activities. This observation is consistent with Section 4.3 and Table 3.

To investigate further, we extracted the *TF* (i.e., the activity of writing a test case followed by production code) and *RF* activities. Both activities are typical of a TDD approach to software development. We present the distribution of the duration of *TF* and *RF* activities using box-and-whisker plots in Fig. 5. The mean median duration of such activities—reported in the plot by a thick vertical line in each box—is approximately four minutes for *TF*, and 20 min for *RF*. Moreover, Fig. 5 shows that the dispersion of the activity duration (i.e., the difference between the rightmost, and leftmost hinges of the box, respectively representing the 75th and 25th quartiles) is more accentuated for *TF* (inter-quartile range approx. 27 min) than for *RF* (inter-quartile range approx. 13 min). In Fig. 5, the outliers (i.e., points above or below the 1.5 × inter-quartile range threshold) are represented by points. The most no-

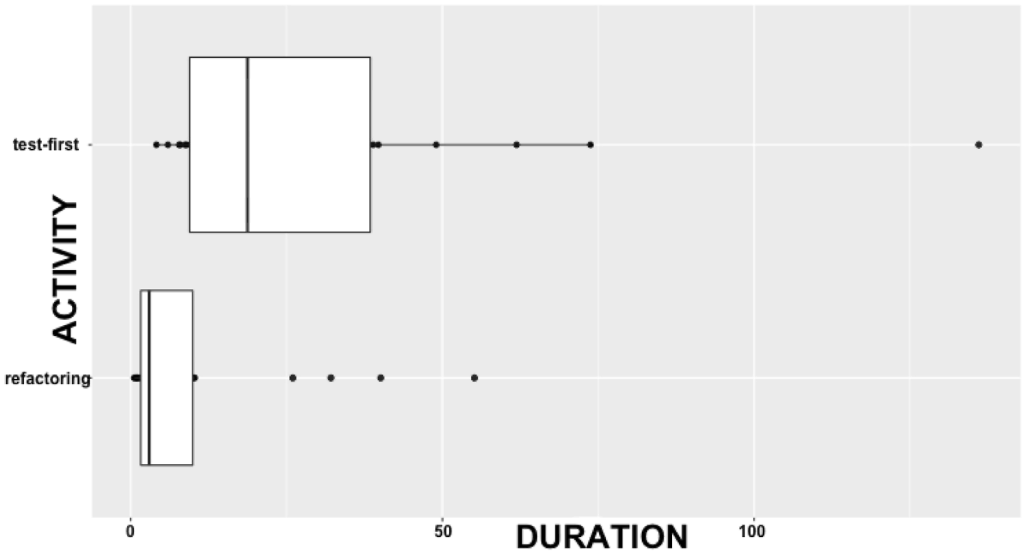

Fig. 5. Differences in activity duration between *TF*, and *RF* performed by the participants.

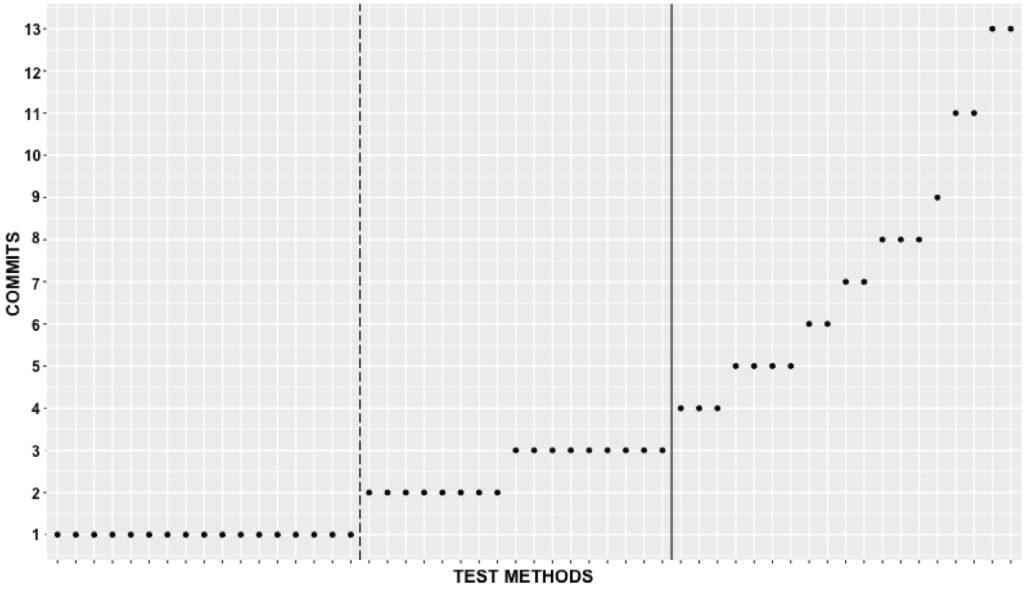

Fig. 6. Unit test methods ordered for number of modifications made in ascending order. The dashed line indicates the 32% percentile, the solid line indicates the 64% percentile.

table outlier refers to a *TF* activity which took over two hours (see Section 4.8.1).

In order to support or reject the observation made in Section 4.3, we compared the duration of the *TF* and *RF* activities using a non-parametric test, because the distribution of activities duration is not normal (Anderson-Darling A = 3.62, p $<<$ .05). The one-tailed version of the test was necessary since our alternative hypothesis is *TF* > *RF*. In particular, the result of a one-tailed Wilcoxon Rank Sum test shows that the time dedicated to *RF* is significantly less ($\alpha = 0.05$) than the one dedicated to *TF* activities (W = 84, p = 2.99e-05).

4.8.3. Trusting test cases

In Section 4.4, we observed that test cases, once created, were seldom updated. From the commits history, we extracted the number of modifications made to JUnit test methods (i.e., methods annotated with `@Test`). The number of modifications (i.e., commits) per each method is presented in Fig. 6. The test methods[5] are ranked in the plot in ascending order, based on the number of commits (i.e., modification) in which they were included. One-third (32%) of the test methods were modified only once (e.g., when created); whereas 64% were committed maximum three times. Moreover, the test method with the largest number of commits (i.e., 13) belonged to the ML pair, which spent most of the time working on the confirmation associated to such test (see Fig. 4). There is evidence supporting our observation that the participants did not often update test cases after their first implementation.

4.8.4. TDD conformance

In Section 4.5, we observed that only in a few occasions the participants deviated from TDD. Leveraging a set of heuristics, val-

[5] The test methods name are not reported for clarity.

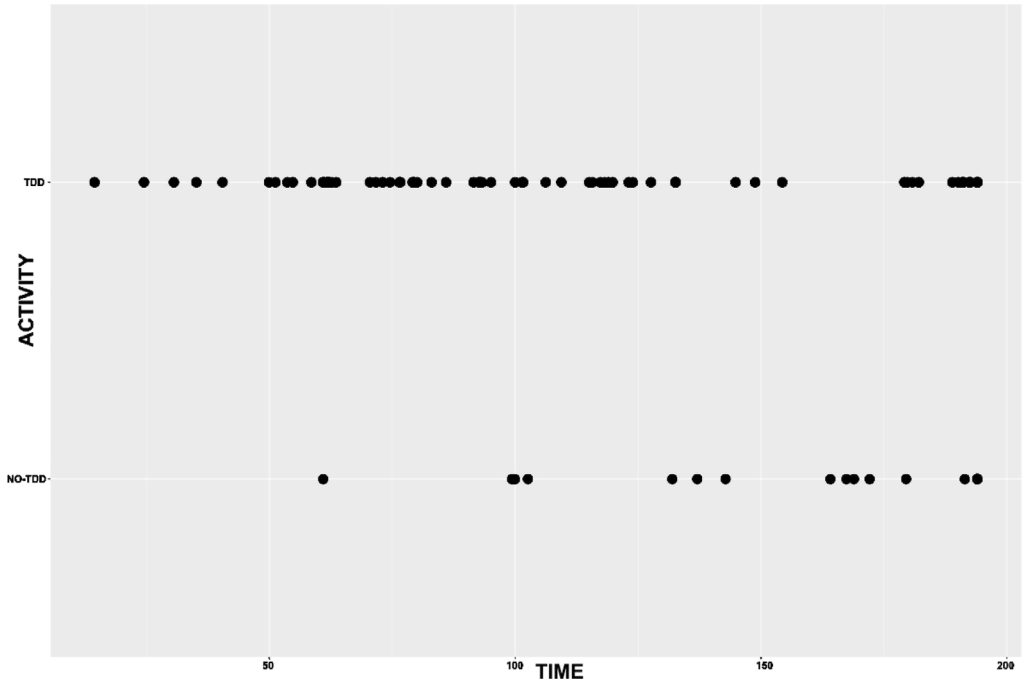

Fig. 7. Timeline of the different activities (TDD, and NO-TDD) during the development.

idated by Kou et al. [58], the Besouro tool labeled an activity as *TDD* or *NO-TDD* compliant. Some particular activities (e.g., *RF, TA*) are context dependent, as they can take place under either a *TDD*, or a *NO-TDD* scenario. A context dependent activity is considered TDD-conformant if both its preceding and following activities are also TDD-conformant. For example:

1. TF → RF → TF
2. TF → TA → TL

sequence 1) includes three *TDD* activities, while sequence 2) includes one *TDD* activity, and two *NO-TDD* activities. The total duration of *TDD* activities was 1168.46 min, while 249.76 min were spent in *NO-TDD* activities. The observation made during the ethnography study is supported by the data, which showed that TDD was followed most of the time. However, we noticed that developers tended to abandon TDD when progressing towards the completion of the user story. In Fig. 7, we present the timeline of the different kind of activities, divided between TDD-conformant and not, taking place in the IDE. Each point represents the time at which the activity was registered (on the x-axis, the zero point corresponds to the start of the study). We normalized the time at which an activity was registered with respect to the largest participant time window (i.e., pair PS). This normalization was necessary to take into account the different amount of time the pairs required to complete the task. The participants started mixing *TDD*, and *NO-TDD* (i.e., points start to be recorded for both types of activities on the y-axis) half-way through the development. Nonetheless, they never completely switched to *NO-TDD*.

Pairs tended to tackle more than they could handle, resulting in prolonged development cycles. TDD advocates that a card should be divided into manageable sub-tasks which should not take longer than 5–10 min to complete [59]. Although TDD was followed most of the time, the average duration of the test-first activities was 30 min. It appears that in some occasions, TDD was preceded by a tacit design phase.

4.8.5. Additional finding

In Fig. 4, we observed that students slightly differed from professionals (Section 4.7) in that professional pairs (i.e., DG, RB and ZP) had shorter, more granular development activities than students pairs.

Fig. 8 reports a box-and-whisker plot of the development activity duration for professional and novice developers. The median durations are similar—approximately eight and 11 min respectively. However, the data for novice developers is more dispersed (inter-quartile range approx. 27 min) than for professionals (inter-quartile range approx. 13 min). The largest outliers are observed for novices. In this case, outliers span over the two hours duration, whereas for professionals the outliers are all below the 55 min threshold. This shows that professionals were more consistent with respect to their development cycle length than novices, while keeping their cycle shorter.

To test this hypothesis, we used a non-parametric test because the distribution of activities duration is not normal (Anderson-Darling $A = 7.85$, $p << 0.05$). The one-tailed version of the test was necessary since our alternative hypothesis is *Novice* > *Professional*. In particular, the results of a one-tailed Wilcoxon rank sum test show the duration of development cycles by professionals to be shorter than novice. However, the small effect size (Hodges–Lehmann estimator = 0.18, CI = [0.06, 0.89]), shows no practical differences.

5. Discussion

The "So What?" factor is relevant in empirical software engineering and ethnography in particular. That is, what significance do the results have for software development? One of the main goals of ethnographically-informed study is to uncover implicit features of practice [23]. What do the results presented in this study tell us about TDD in general? And what do the achieved results tell us about TDD applicability to software evolution tasks in particular?

The developers needed to plan a solution in advance.

The developers built a model of the solution that would later be put in the form of unit tests. As we observed from the gath-

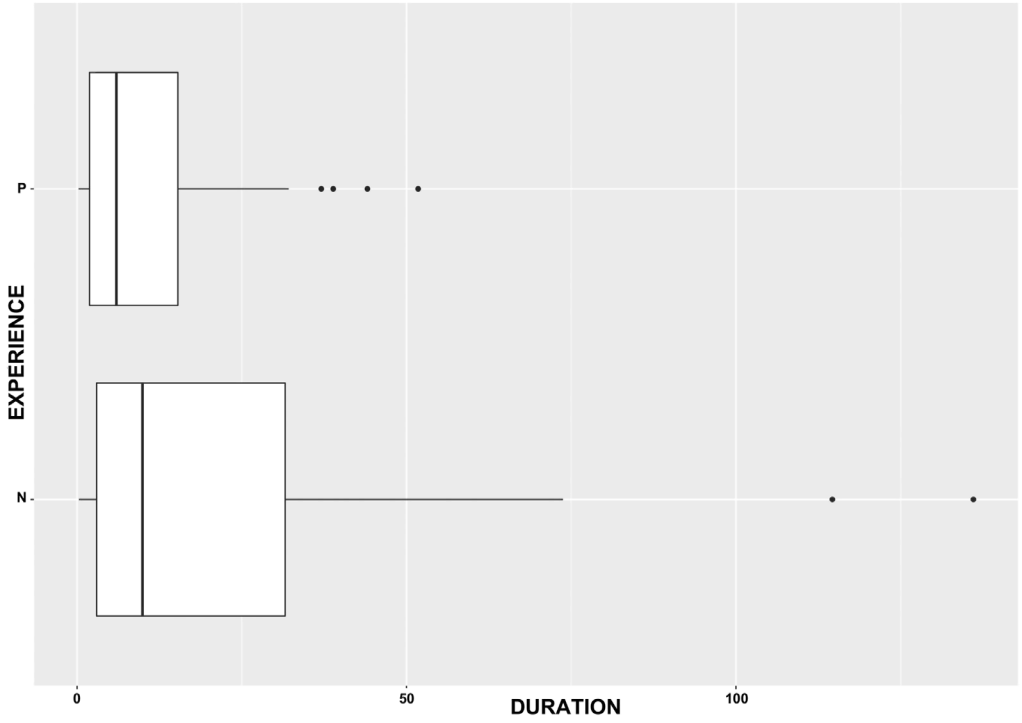

Fig. 8. Differences in activities duration between professional (P) and novice (N) developers.

ered qualitative data (see Section 4.2), the pairs had lengthy discussions in order to attain an implementation model before tackling each confirmation. This behavior is supported by previous research investigating the strategies developers use to comprehend existing code [60,61]. In other words, we believe that the pairs approached the problem in a white box rather than a black box fashion. As we observed in Section 4.2, they conceived and developed unit tests according to the implementation details framed in their minds rather than the intended interface behavior. Although the main impediment in the adoption of TDD is often reported to be the switching from test-last to test-first [3], it seemed that the real problem was switching from a plan-intensive mindset to a lightweight and flexible one. This observation may be aggravated by the cold start due to the presence of legacy code reported in Section 4.8.1. The issue of how to design in the context of TDD is considered a limiting factor [3], as well as the issue of losing sight of the big picture due to the lack of design [62]. The pairs tended to write unit tests that were large and complex rather than relying on small iterations aided by simple unit tests. This tendency was particularly marked for students, whereas professionals were more aware of the benefits of granular iterations—for example, the *TF* activity in Fig. 4. The TDD process does not explicitly include a preliminary planning phase that focuses on dividing the task at hand into sub-tasks of a suitable granularity.

The developers dedicated less time to refactoring, compared to other activities.

From data analysis it appears that root canal refactoring took place—e.g., refactoring was performed in a periodic fashion (towards the end of the task), rather than interleaved with other activities (i.e., floss-refactoring). This is a surprising result because TDD promotes short and frequent refactoring; whereas, in our case, it was postponed for several iterations. Moreover, floss-refactoring was shown to be the preferred refactoring tactic in larger studies (i.e., [63,64]). However, in contrast with such studies, the legacy system the developers have been working on did not have any test, therefore hindering refactoring. When refactoring was attempted, it was always preceded by a test addition phase (i.e., developing a unit test, but without new production code) as shown in Fig. 4. These tests are sometimes called *characterization tests*—e.g., tests that describe the observed behavior, rather than the expected one [52]. In our study, it seems that characterization testing was applied only to a limited extent in order to cover a component before refactoring it. We recommend to apply it to legacy system, or at least to the system sensible parts, as this will result in a set of regression tests that facilitates TDD.

The qualitative data show that, for both student and professional pairs, refactoring was not perceived as a step worthy of effort, whereas the focus was on completing the card confirmations—i.e., getting from the red to the green phase. The developers' lack of concern for internal code quality[6] and refactoring also manifested during a previous focus group in a similar setting [65]. It appears that the only support that TDD offered to their development process was to prompt developers to write unit tests. In this regard, better tool support could be beneficial. The IDE could inform the user when a code smell is detected after the end of each development cycle—i.e., once the unit tests for that feature passed. As TDD becomes more widespread, there is a need for ad hoc tools integrated in the IDE to support the process [3,65].

The developers felt most comfortable with the green phase.

This was to be expected, as this phase is where the software was actually developed, and seemed to be the most rewarding for the pairs. Nevertheless, pairs wrote production code regardless of the associated test boundaries. In other words, they wrote more code than necessary to just pass the test (see Section 4.3). We believe the developers gave priority to the model of the solution they built before starting a TDD cycle, rather than to the test written in the red phase. Hence, we reiterate the idea that pairs gave more importance to the model of the solution they built in their heads, than the tests. We suspect that applying TDD in such a way can be

[6] In this context defined as the code-based properties for creating and maintaining the developed solution.

detrimental because the additional code is likely to remain uncovered and could contain bugs that will be hard to catch later on.

What we observed is a mismatch between how participants first imagined the source code to be implemented and its actual implementation following unit tests. If that were the case, the IDE could support the process by prompting the user to take action with respect to the parts of the system with poor test coverage. Nevertheless, we could not collect evidence to substantiate such a claim, and so this remains a subject which we will address in future investigations—for example, using on-line coverage metrics together with the activities.

The developers made few changes to the unit tests.

A controlled experiment [66] showed that TDD developers tend to make several changes to test code, more than to production code, despite their experience. This relates to the iterative nature of TDD, in which the design emerges gradually from the test though continuous refactoring. For example, after a number of TDD iterations, some part of the source code needed to be generalized to remove repetitions. The application of a refactoring (e.g., extract class) changes the architecture that should be appropriately mirrored in the test code. Moreover, the correspondence between the structure of production code and test code (e.g., API naming) can be altered during TDD [37]. For example, if the name of a method within a class is changed (i.e., renaming), then a refactoring to the test(s) for that method is also necessary.

We observed the few modifications to the test code. We believe it to be associated with the general lack of refactoring applied to the production code. As stated before, the legacy nature of the system may have resulted in a small amount of refactoring, which in turn influences the modifications made to a test. In this regards, refactoring should be enforced for unit tests as well, as they represent the core of the TDD practice.

Additional Finding

As an additional result, we report that professional and novice developers apply a similar process. Although we found a difference between the cycle duration of professionals and novices, the small effect size shows that such a difference has no practical impact. Interestingly, a similar result was reported in the only study, to the best of our knowledge, that focused on developer cycles duration in the context of TDD [66]. However, as no effect size is reported in [66], it is difficult to assess its practical implications. In general, our results are in line with what is reported in a longitudinal study with participants of mixed levels of experience [67]; after an initial ramp-up period, novice and experienced developers applied TDD in a similar fashion.

Given Fig. 4 and our observations, the pair that best applied TDD was RB. They did not suffer from a cold start, but rather started by applying TDD to smaller sub-tasks than the other pairs. They also emphasized refactoring, although most of it was left to the end of the development session.

5.1. Significance of the results

The results have implication, regarding TDD as a software engineering practice. In particular, the mental model created by the developers does not suit the granular nature of TDD—which, for example, is showed to have positive effects on software quality [68]. We propose the integration of another practice[7] to address this issue, alongside with a more granular specification of the requirements that would better fit the process.

Regarding software evolution, our study showed that applying TDD to a legacy system that is not covered by a test suite is difficult. TDD leverages extensive refactoring in order to make a solution (both architectural and algorithmical) emerge. Therefore, using TDD for a system that hinders refactoring is problematic. What we observed from our participants—and recommend when applying TDD to a legacy system—is the use of characterization tests.

Finally, professionals and novices alike are able to apply TDD after few weeks of practice. Students with experience in testing (like in the case of our study), but not experienced with TDD, can be representative of professional developers under the same conditions. This makes them suitable participants for similar studies regarding this practice.

6. Limitations

In this section, we discuss possible limitations of our study. Regarding the timing of the study, the duration was approximately four hours (in total) for each pair, or roughly half of a normal working day. Although we were able to observe how developers use TDD during the initial development phases of a new feature of a legacy system, the study does not include long-term observations. Thus, we may be excluding some important elements from our results.

The use of students may affect generalizability. Neither the students nor the professionals were experts in TDD, although they have been practicing it for the duration of the course (approximately two months). There is evidence that at least the more experienced developers can learn TDD with little effort [67]. Therefore, our findings represent a setting in which new developers join a brownfield project in which the use of TDD and pair-programming is enforced (e.g., within an agile company). Nevertheless, we acknowledge a difference between such settings and the ones in our study. Usually a pair consists of one new developer and a developer already experienced in TDD.

Social factors should be taken into account when evaluating qualitative and quantitative findings (e.g., evaluation apprehension). In order to address this concern, the first author (the observer) used an informal approach to interact with pairs. To mitigate social factors, students were not evaluated based on their participation in the study. Both students and professionals, from a convenience sample, voluntarily participated in the study. This may influence the results, since voluntary participants are generally well motivated.

The chosen implementation task might effect the observed quantitative and qualitative results. Indeed, the use of an application different from MusicPhone could lead to different results. The use of commercial applications is the subject of our future work. In fact, we plan to involve some companies of our industrial contact networks in a research project on the topics concerned with the study presented here. Quantitative results could also be biased due to the metrics Besouro gathered, yet this tool has been empirically validated. The results indicate a good accuracy to classify development activities [58,69]. For example, Kou et al. [58] showed that the accuracy rate of Besouro is 89% [58] when classifying development activities as TDD or NO-TDD.

Statistical analysis could also affect conclusion validity. We addressed this threat by using robust statistical methods and carefully checking the assumptions of the statistical tests. The quantitative results might suffer from low statistical power—i.e., there could be a risk that an erroneous conclusion is drawn. However, these threats are minor here because the main goal of our quantitative analysis is to support outcomes from the ethnographically-informed part of our study.

7. Conclusion

We have shown the results of an ethnographically-informed study to investigate how students and professional developers ap-

[7] http://alistair.cockburn.us/Elephant+carpaccio.

ply TDD. In addition, we also analyzed quantitative data about the activities carried out by developer's in their IDE.

In our study, we kept the settings as close as possible to the everyday work. Based on qualitative information, we have observed that: *(i)* refactoring is not performed as often as TDD requires, and it is considered less important than other phases, *(ii)* implementation is considered the most important phase; *(iii)* unit tests are often not up-to-date; and *(iv)* participants first imagine the source code to be implemented and then write test cases. An analysis of the quantitative data allowed us to confirm findings: *(i)*, *(iii)*, and *(iv)*. It is worth mentioning that available quantitative data did not allow us to confirm, nor contradict, all the qualitative findings because some are exclusively qualitative (e.g., how participants consider TDD phases).

Future work should address how refactoring is performed during a TDD cycle and how tools can support it. The traditional TDD cycle can be improved by adding an additional phase focused on splitting the task into simpler, finer-grained sub-tasks more suitable to be framed in a unit test during the red phase. In addition, we observed that developers tended to write more code than necessary to pass the unit test at the hand. This leaves part of the source code uncovered by tests. It could be interesting to investigate "when" and "why" this happens and what is the effect of the presence of uncovered source code on software quality. Finally, it also stands to reason that a better IDE support for the green phase (e.g., coverage metrics for each TDD cycle) can be beneficial.

Acknowledgment

This research is supported in part by the Academy of Finland Project no. 278354. We would like to acknowledge Dr. Lucas Layman and Dr. Hakan Erdogmus, who designed the task used in the study. We thank the students and the professional developers for their participation in our ethnographically-informed study.

References

[1] K. Beck, Test-Driven Development: By Example, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[2] D. Astels, Test Driven Development: A Practical Guide, Prentice Hall Professional, 2003.

[3] A. Causevic, D. Sundmark, S. Punnekkat, Factors limiting industrial adoption of test driven development: A systematic review., in: Proceedings of International Conference on Software Testing, IEEE Computer Society, 2011, pp. 337–346.

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

[5] D. Fucci, B. Turhan, On the role of tests in test-driven development: a differentiated and partial replication, Empirical Softw. Eng. 19 (2) (2014) 277–302.

[6] I. Salman, A.T. Misirli, N. Juristo, Are students representatives of professionals in software engineering experiments? in: Procedings of International Conference on Software Engineering, 2015, pp. 666–676.

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

[8] B. Turhan, L. Layman, M. Diep, H. Erdogmus, F. Shull, How Effective is Test-Driven Development, in: Making Software: What Really Works, and Why We Believe It, 2010, pp. 207–217.

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

[10] M. Siniaalto, P. Abrahamsson, A comparative case study on the impact of test-driven development on program design and test coverage, in: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, ACM/IEEE Computer Society, 2007, pp. 275–284, doi:10.1109/ESEM.2007.35.

[11] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer, 2012.

[12] H. Sharp, H. Robinson, M. Woodman, Software engineering: community and culture, IEEE Softw. 17 (1) (2000) 40–47, doi:10.1109/52.819967.

[13] S.E. Sim, Evaluating the evidence: Lessons from ethnography, in: Proceedings of the Workshop on Empirical Studies of Software Maintenance, 1999, pp. 66–70.

[14] M. Hammersley, P. Atkinson, Ethnography: Principles in Practice, Taylor & Francis, 2007.

[15] H. Sharp, C. deSouza, Y. Dittrich, Using ethnographic methods in software engineering research, in: Proceedings of the International Conference on Software Engineering, ACM, New York, NY, USA, 2010, pp. 491–492.

[16] H. Sharp, Y. Dittrich, C.R.B. de Souza, The role of ethnographic studies in empirical software engineering, IEEE Trans. Softw. Eng. 42 (8) (2016) 786–804.

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

[18] D. Shapiro, The limits of ethnography: Combining social sciences for CSCW, in: Proceedings of the Conference on Computer Supported Cooperative Work, 1994, pp. 417–428, doi:10.1145/192844.193064.

[19] A. Crabtree, T. Rodden, P. Tolmie, G. Button, Ethnography considered harmful, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, in: CHI '09, ACM, New York, NY, USA, 2009, pp. 879–888, doi:10.1145/1518701.1518835.

[20] P. Beynon-Davies, Ethnography and information systems development: ethnography of, for and within is development, Inf. Softw. Technol. 39 (8) (1997) 531–540.

[21] P. Beynon-Davies, D. Tudhope, H. Mackay, Information systems prototyping in practice, J. Inf. Technol. 14 (1) (1999) 107–120, doi:10.1080/026839699344782.

[22] G. Button, W. Sharrock, Project work: the organisation of collaborative design and development in software engineering, Comput. Supported Cooperative Work 5 (4) (1996) 369–386.

[23] H. Sharp, H. Robinson, An ethnographic study of xp practice, Empirical Softw. Eng. 9 (4) (2004) 353–375, doi:10.1023/B:EMSE.0000039884.79385.54.

[24] J. Singer, T. Lethbridge, N. Vinson, N. Anquetil, An examination of software engineering work practices, in: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research, IBM Press, 1997, pp. 21–32.

[25] F. Salviulo, G. Scanniello, Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals, in: Proceedings of International Conference on Evaluation and Assessment in Software Engineering, ACM, 2014, pp. 48:1–48:10.

[26] D. Martin, J. Rooksby, M. Rouncefield, I. Sommerville, 'good' organisational reasons for 'bad' software testing: An ethnographic study of testing in a small software company, in: 29th International Conference on Software Engineering (ICSE'07), 2007, pp. 602–611, doi:10.1109/ICSE.2007.1.

[27] J. Rooksby, M. Rouncefield, I. Sommerville, Testing in the wild: the social and organisational dimensions of real world practice, Comput. Supported Cooperative Work 18 (5–6) (2009) 559–580, doi:10.1007/s10606-009-9098-7.

[28] H. Munir, M. Moayyed, K. Petersen, Considering rigor and relevance when evaluating test driven development: a systematic review, Inf. Softw. Technol. (2014).

[29] Y. Rafique, V.B. Misic, The effects of test-Driven development on external quality and productivity: A Meta-Analysis, IEEE Trans. Softw. Eng. 39 (6) (2013) 835–856.

[30] M. Muller, W. Tichy, Case study: extreme programming in a university environment, in: Proceedings of the 23rd International Conference on Software Engineering, 2001, pp. 537–544, doi:10.1109/ICSE.2001.919128.

[31] A. Gupta, P. Jalote, An experimental evaluation of the effectiveness and efficiency of the test driven development, in: Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on, 2007, pp. 285–294, doi:10.1109/ESEM.2007.41.

[32] M. Pancur, M. Ciglaric, M. Trampus, T. Vidmar, Towards empirical evaluation of test-driven development in a university environment, in: EUROCON 2003. Computer as a Tool. The IEEE Region 8., 2003, pp. 83–86.

[33] A. Geras, M. Smith, J. Miller, A prototype empirical evaluation of test driven development, in: Software Metrics, 2004. Proceedings. 10th International Symposium on, 2004, pp. 405–416, doi:10.1109/METRIC.2004.1357925.

[34] D. Fucci, B. Turhan, M. Oivo, Conformance factor in test-driven development: initial results from an enhanced replication, in: 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, England, United Kingdom, May 13–14, 2014, 2014, pp. 22:1–22:4, doi:10.1145/2601248.2601272.

[35] D. Fucci, B. Turhan, N. Juristo, O. Dieste, A. Tosun-Misirli, M. Oivo, Towards an operationalization of test-driven development skills: an industrial empirical study, Inf. Softw. Technol. 68 (2015) 82–97. http://dx.doi.org/10.1016/j.infsof.2015.08.004.

[36] M. Beller, G. Gousios, A. Panichella, A. Zaidman, When, how, and why developers (do not) test in their ides, in: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 179–190.

[37] M. Hilton, N. Nelson, H. McDonald, S. McDonald, R. Metoyer, D. Dig, Tddviz: Using software changes to understand conformance to test driven development, in: Agile Processes, in Software Engineering, and Extreme Programming - 17th International Conference, XP 2016, Edinburgh, UK, May 24–27, 2016, Proceedings, 2016, pp. 53–65, doi:10.1007/978-3-319-33515-5_5.

[38] V. Heikkilä, M. Paasivaara, C. Lassenius, Scrumbut, but does it matter? a mixed-method study of the planning process of a multi-team scrum organization, in: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, IEEE, 2013, pp. 85–94.

[39] J. Segal, A. Grinyer, H. Sharp, The type of evidence produced by empirical software engineers, in: Proceedings of the workshop on Realising evidence-based software engineering, ACM, 2005, pp. 1–4, doi:10.1145/1082983.1083176.

[40] A. Jedlitschka, M. Ciolkowski, D. Pfahl, Reporting experiments in software engineering, in: Guide to advanced empirical software engineering, Springer, 2008, pp. 201–228.

[41] J.C. Carver, N. Juristo Juzgado, M.T. Baldassarre, S. Vegas Hernández, Replications of software engineering experiments, Empirical Softw. Eng. 19 (2) (2014) 267–276.

[42] C.B. Seaman, Qualitative methods in empirical studies of software engineering, IEEE Trans. Softw. Eng. 25 (4) (1999) 557–572, doi:10.1109/32.799955.

[43] H. Robinson, J. Segal, H. Sharp, Ethnographically-informed empirical studies of software practice, Inf. Softw. Technol. 49 (6) (2007) 540–551, doi:10.1016/j.infsof.2007.02.007.

[44] B. George, L. Williams, A structured experiment of test-driven development, Inf. Softw. Technol. 46 (5) (2004) 337–342.

[45] K. Zieliriski, T. Szmuc, Preliminary analysis of the effects of pair programming and test-driven development on the external code quality, Softw. Eng. 130 (2006) 113.

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

[47] G. Scanniello, M. Risi, Dealing with faults in source code: Abbreviated vs. full-word identifier names, in: Proceedings of International Conference of Software Maintenance, IEEE Computer Society, 2013, pp. 11–21.

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

[49] M. Höst, B. Regnell, C. Wohlin, Using students as subjects—a comparative study of students and professionals in lead-time impact assessment, Empirical Softw. Eng. 5 (3) (2000) 201–214, doi:10.1023/A:1026586415054.

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

[51] D. Astels, G. Miller, M. Novak, A Practical Guide to eXtreme Programming, Prentice Hall, 2002.

[52] M. Feathers, Working Effectively with Legacy Code, Prentice Hall, 2004.

[53] M. Fowler, J. Highsmith, The agile manifesto, Software Dev. 9 (8) (2001) 28–35.

[54] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.

[55] D. Garlan, F. Bachmann, J. Ivers, J. Stafford, L. Bass, P. Clements, P. Merson, Documenting Software Architectures: Views and Beyond, 2nd ed., Addison-Wesley Professional, 2010.

[56] C. Passos, D.S. Cruzes, T. Dybå, M. Mendonça, Challenges of applying ethnography to study software practices, in: Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, in: ESEM '12, ACM, 2012, pp. 9–18, doi:10.1145/2372251.2372255.

[57] D.C. Littman, J. Pinto, S. Letovsky, E. Soloway, Mental models and software maintenance, J. Syst. Softw. 7 (4) (1987) 341–355.

[58] H. Kou, P.M. Johnson, H. Erdogmus, Operational definition and automated inference of test-driven development with zorro, Autom. Softw. Eng. 17 (1) (2010) 57–85.

[59] K. Beck, Test Driven Development: By Example, Addison Wesley, 2003.

[60] T. Roehm, R. Tiarks, R. Koschke, W. Maalej, How do professional developers comprehend software? in: Proceedings of the 2012 International Conference on Software Engineering, in: ICSE 2012, IEEE Press, Piscataway, NJ, USA, 2012, pp. 255–265.

[61] A.J. Ko, B.A. Myers, M.J. Coblenz, H.H. Aung, An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks, IEEE Trans. Softw. Eng. 32 (12) (2006) 971–987, doi:10.1109/TSE.2006.116.

[62] A. Begel, N. Nagappan, Usage and perceptions of agile software development in an industrial context: An exploratory study, in: Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on, IEEE, 2007, pp. 255–264.

[63] Q.D. Soetens, S. Demeyer, Studying the Effect of Refactorings: A Complexity Metrics Perspective, IEEE, 2010.

[64] E. Murphy-Hill, C. Parnin, A.P. Black, How we refactor, and how we know it, IEEE Trans. Softw. Eng. 38 (1) (2012) 5–18.

[65] G. Scanniello, S. Romano, D. Fucci, B. Turhan, N. Juristo, Students' and professionals' perceptions of test-driven development: A focus group study, in: Proceedings of the 31th Annual ACM Symposium on Applied Computing, in: SAC '16, ACM, New York, NY, USA, 2016, pp. 1422–1427.

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

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

[68] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, N. Juristo, A dissection of test-driven development: does it really matter to test-first or to test-last? IEEE Trans. Softw. Eng. PP (99) (2016), doi:10.1109/TSE.2016.2616877. 1–1

[69] K. Becker, B. de Souza Costa Pedroso, M.S. Pimenta, R.P. Jacobi, Besouro: a framework for exploring compliance rules in automatic TDD behavior assessment, Inf. Softw. Technol. 57 (2015) 494–508.