Results from an Ethnographically-informed Study in the Context of Test Driven Development

Simone Romano
University of Basilicata
Viale Dell'Ateneo 10
Macchia Romana
Potenza, Italy
simone.romano@unibas.it

Davide Fucci
University of Oulu
Pentti Kaiteran katu 1
Oulu, Finland
davide.fucci@oulu.fi

Giuseppe Scanniello
University of Basilicata
Viale Dell'Ateneo 10
Macchia Romana
Potenza, Italy
giuseppe.scanniello@unibas.it

Burak Turhan
University of Oulu
Pentti Kaiteran katu 1
Oulu, Finland
Burak.Turhan@oulu.fi

Natalia Juristo
Facultad de Informatica,
Universidad Politecnica de Madrid
Campus de Montegancedo,
28660 Boadilla del Monte
Madrid, Spain
natalia@fi.upm.es

ABSTRACT

Background: Test-driven development (TDD) is an iterative software development technique where unit tests are defined before production code. Previous studies fail to analyze the values, beliefs, and assumptions that inform and shape TDD.
Aim: We designed and conducted a qualitative study to understand the values, beliefs, and assumptions of TDD. In particular, we sought to understand how novice and professional software developers, arranged in pairs (a driver and a pointer), perceive and apply TDD.
Method: 14 novice software developers, i.e., graduate students in Computer Science at the University of Basilicata, and six professional software developers (with one to 10 years work experience) participated in our ethnographically-informed study. We asked the participants to implement a new feature for an existing software written in Java. We immersed ourselves in the context of the study, and collected data by means of contemporaneous field notes, audio recordings, and other artifacts.
Results: A number of insights emerge from our analysis of the collected data, the main ones being: *(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, *(iv)* participants first build a sort of mental model of the source code to be implemented and only then write test cases on the basis of this model; and *(v)* apart from minor differences, professional developers and students applied TDD in a similar fashion.
Conclusions: Developers write quick-and-dirty production code to pass the tests and ignore refactoring.

CCS Concepts

•Software and its engineering → Software development techniques;

Keywords

Ethnographically-informed Study, Qualitative Study, Test Driven Development

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted.
Copyright is held by the owner/author(s).
EASE '16, June 01-03, 2016, Limerick, Ireland
ACM. ISBN 978-1-4503-3691-8/16/06
DOI: http://dx.doi.org/10.1145/2915970.2915996

This work is licensed under a Creative Commons Attribution-NoDerivs International 4.0 License.

1. INTRODUCTION

Test-driven development (TDD) is an iterative software development practice within agile methodologies [1]. Some software organizations have been quick to adopt TDD, while others are still evaluating its benefits in terms of cost, quality, and productivity [8, 29]. A number of primary (e.g., controlled and quasi-experiments) and secondary (e.g., systematic literature reviews) empirical studies have been published. The primary studies (e.g., ([11, 32]) have been quantitative in nature, and have produced contrasting or inconclusive results [39]. The secondary studies summarize the empirical research results regarding TDD by aggregating, to varying extent, the evidence from controlled experiments, quasi-experiments, and case studies [8, 29, 39, 42].

TDD has been marginally investigated from a qualitative point of view and from the perspective of the developer [23, 41]. Qualitative studies, unlike quantitative ones, inquire into the underlyinge reasons and motivations behind a given phenomenon [43]. In this paper, we present the results of an ethnographically-informed study involving students and professional software developers. Our goal is to gain insights into how they 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 [17]. 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 the participants, working in pairs, to add a new functionality to an existing software implemented in Java. The software is a complex, industrial-like case of which participants had some knowledge. We immersed ourselves in the study environment and participated in conversations, and asked the participants how they were applying TDD to perform the assigned implementation task. We collected data by means of contemporaneous field notes, audio recordings of discussions, and copies of artifacts produced by the participants during the study. Information about participants' conformance to TDD was also gathered through an automated tool installed in the participants' integrated development environment (IDE).

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

We first discuss ethnographically-informed studies in software engineering, and then papers reporting investigations regarding TDD.

2.1 Ethnographically-informed Studies

Little ethnographic research exists in the field of software engineering. Beynon-Davies [4] observed that ethnographic research may be useful for capturing knowledge about intangible or unquantifiable aspects of the software life cycle. Later, Beynon-Davies *et al.* [5] used ethnographic methods to investigate the negotiated order of work and the role of collective memory in rapid application development. Button and Sharrock [6] carried out an ethnographically-informed study in global software development with the goal of explaining knowledge that is displayed in collaborative actions and interactions of design and development. Sharp and Robinson [37] used ethnographic methods in their study of eXtreme Programming (XP) focusing on developers in a small company implementing web-based intelligent advertisements. Their result suggested that XP developers were clearly "agile."

Singer *et al.* [40] studied the behavior of software engineers responsible for maintaining a large telecommunications system at a particular company. The authors examined developers' habits and tool usage during software development. The authors discovered a discrepancy between what developers claimed to do when performing maintenance operations, and what they actually did. In particular, developers claimed to solve problems by "reading documentation", while in fact Singer *et al.* found that more often than not they resolved issues by looking up and copying source code. Later, Salviulo and Scanniello [33] conducted an ethnographically-informed study with students and professionals to understand the role of comments and identifiers in source code comprehensibility and maintainability. Outcomes can be summarized as follows: *(i)* professional developers (with respect to students) prefer to deal with identifiers rather than comments, *(ii)* all participants believed essential the use of naming convention techniques for specifying identifiers, and *(iii)* all participants stated that the names of identifiers are important and developers should properly choose them. Ethnography can thus be a useful tool for detecting and explaining such a kind of discrepancies, and to make clearer un-remarked aspects of a practice [38]. Indeed, this is the main motivation behind our use of ethnography in the present study.

2.2 Empirical studies on TDD

Several quantitative studies have assessed the effectiveness of TDD, and the results of these studies have in turn been examined in a number of systematic reviews and meta-analyses [26, 30, 42]. The results of these analyses have been contradictory regarding both software products (e.g., defects) and developers (e.g., productivity). Interestingly, one of the secondary studies [8] 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 perception of developers regarding the TDD practice. For example, Mueller and Tichy [25] examined several Agile methodologies, including TDD, within a university course and found that TDD was one of the most difficult techniques to adopt because developers felt that it was impractical to write test cases before coding. On the other hand, Gupta and Jalote [16] reported that students felt more confident that testing effort applied by using TDD would yield better results than a traditional test-after-code setting (test-last approach), while feeling the need for some upfront design. Pancur et al. [27] reported that students perceived TDD as more difficult for professionals to adopt. 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 prevents bugs, but they also believed that this practice cannot replace a quality assurance engineer [39]. 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 [15].

While quantitative studies provide objective frameworks for assessing TDD effectiveness, qualitative studies may enable a deeper understanding of TDD and its use. 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., [41]). 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., [32]).

3. ETHNOGRAPHIC STUDY

Qualitative studies are considered a necessary complement to quantitative investigations [36], essential for gaining an understanding of the reasons and motivations behind the problem under study. Among qualitative methodological approaches, ethnographic studies are better suited to ask questions such as *how*, *why* and *what are the characteris-*

tics of [31]. Usually, such investigations are conducted on a small number of subjects, while researchers conduct study by immersing themselves in the study environment [37]. 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 to the shorter timeframe [31, 33].

3.1 Definition and Context

TDD is an iterative software development technique where unit tests are defined before production code. Developers repeat short cycles consisting of: *(i)* writing a unit test for an unimplemented functionality or behavior, namely the red phase; *(ii)* supplying the minimal amount of production code to make unit tests pass, namely the green phase; *(iii)* applying refactoring where necessary, and checking that all tests are still passed after refactoring [1].

Often pair-programming is used together with TDD [14, 44]. For this reason, in the present 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

what are the characteristics the developers believe an application must have so that they can successfully apply TDD

In order to mitigate subjective assumptions, we considered all the activities related to our study as "strange", as Sharp *et al.*'s suggestion [38].

The participants in the 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), a more traditional development technique where unit tests are written after a feature (or a set of related features) is implemented in a given software.

In the literature, graduate students are considered not far from novice software developers (e.g., [20, 34]). However, a comparison with professional developers would help in better understanding whether and under which conditions this assumption can be considered true [7, 18]. This may not be considered as the main point here, but it might represent an additional contribution of our study.

The Professional developers participating in this study worked in different small/medium sized companies located in southern Italy. The most experienced among the professional held a Master degree in Computer Science, while others held Bachelor degrees in Computer Science. All the professionals had knowledge of testing approaches and techniques (e.g., unit testing, integration testing, and system testing) before participating in our study. At the time of the study they were attending a refresher course on agile software development. The lecturer devoted the greater part of the course to the introduction of TDD and to its application it to real-life 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 study as part of a series of optional laboratory exercises conducted within an Information System (IS) course. This course covered elements of software testing, software development, software maintenance, agile development techniques with a focus on TDD, XP, regression testing, and refactoring. Homework and classwork provided students with opportunities 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. Their prior knowledge can be considered rather homogeneous. The same lecturer held both the training course for professional adjournment and IS.

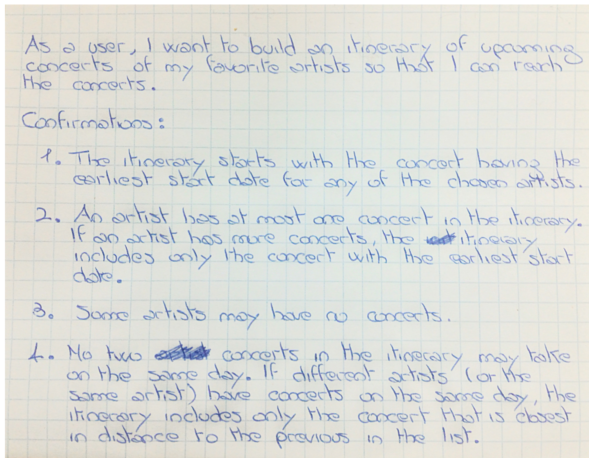
Pairs worked on MusicPhone—an application written in Java which runs on GPS-enabled devices. MusicPhone gives the user recommendations for artists he/she may like, and finds upcoming concerts for such artists and bands. MusicPhone was primarily chosen for the availability of its source code and because it was used in previous empirical studies on TDD (e.g., [12, 32]).

The total number of classes in the existing application was 30, while the non-commented lines of source code were 1,225. The number of methods and constructors was 157 and 22, respectively. The participants worked on a legacy system, i.e., an existing codebase that is not covered by tests [9]. We asked pairs to implement a new feature for MusicPhone. This feature, *Compute an itinerary for artists*, can be 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. We chose this kind of representation 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. In other words, a card contains little information for the implementation of a requirement. 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*. Confirmations summarize conversations among stakeholders, namely they revolve around those they have been reached on a given aspect (e.g., constraints of a functionality for the software underdevelopment) [19]. In this sense, they are a sort of acceptance test for a story. The more formal and unambiguous confirmations are, the better.

[1] http://junit.org



As a user, I want to build an itinerary of upcoming concerts of my favorite artists so that I can reach the concerts.

Confirmations:

1. The itinerary starts with the concert having the earliest start date for any of the chosen artists.
2. An artist has at most one concert in the itinerary. If an artist has more concerts, the itinerary includes only the concert with the earliest start date.
3. Some artists may have no concerts.
4. No two concerts in the itinerary may take on the same day. If different artists (or the same artist) have concerts on the same day, the itinerary includes only the concert that is closest in distance to the previous in the list.

Figure 1: Story card used in the study

Figure 2: A pair of professional developers

The story card used in our study is shown in Figure 1 (top). It contains four confirmations (on the bottom) that allow the developer to better understand the feature to be implemented and constraints to be tackled.

MusicPhone source code was scarcely commented (56 comment lines in total), as is often the case in the context of agile software development, where the goal is to produce clean code and working software is preferred over documentation [10]. We provided the participants with the documentation of MusicPhone architecture. The use of such documentation in agile projects is common in order to avoid big design upfront [13]. Both source code comments and documentation were written in the English language.

All participants were familiar with the problem domain of MusicPhone because the lecturer previously used parts of the same system for homework and classwork when introducing TDD. Used parts of MusicPhone did not contain the source code we asked the participants to implement in our ethnographically-informed 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. Describing the setting is a good practice in ethnographically-informed studies, as the spatial organization could be relevant insofar for developers working to accomplish a given task [37]. Figure 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 observer and the pair were present each time. All the pairs used the same laptop to carry out the task. These measures were taken to minimize any possible bias arising from differences in physical settings.

3.3 The Study

The study was conducted by a single observer (the first author of this paper) between May and July 2015, and was founded on one-to-one sessions between the observer and each pair. The use of one-to-one sessions is almost customary in ethnographically-informed studies (e.g., [31]). We conducted our study in Italian to minimize any bias arising from participants' varying levels of familiarity with the spoken English language.

The observer spent more than three hours working with each pair. As mentioned before, all the pairs had some familiarity with some parts of MusicPhone source code which were previously the object of homework and classwork. This scenario is not unlike in industry; 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—without conditioning their work habit—by focusing on both the application and solution domains of MusicPhone. Such interaction is critical because: *(i)* the ethnographic methodological approach encourages participation [38]; *(ii)* it provides the observer with opportunities to appreciate the perspective of developers while carrying out assigned tasks; and *(iii)* it provides the observer with opportunities to gather information on how a method is applied (TDD in our case). The observer in our study (as customary) avoided influencing pairs' task execution.

Data were collected in a variety of forms, including contemporaneous field notes, audio recordings of discussions, and copies of various artifacts (e.g., source code and notes). In addition, we also gathered information on developer's conformance to TDD. This was possible because pairs used Besouro,[2] an Eclipse plug-in capable of tracing how developers applied TDD. This plug-in runs in the background and does not interfere with the use of Eclipse.

3.4 Design

The study was organized in the following four steps:

1. *Pre-questionnaire.* Each participant was asked to fill out a pre-questionnaire to gather information about their experience in the industry, grade point average, and knowledge. We used this information to get further information on participants, namely the context of the study (see Section 3.1). Gathered information was also used to select the IDE. We opted for Eclipse because participants had a good level of familiarity with this IDE.

2. *Introduction to the study.* The observer introduced the study to each pair. A prearranged schema was

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

employed, namely a few sentences to describe what participants had to do. The observer did not provide details on research topics of interest. At the end of this step, participants could ask questions for clarification.

3. *Inspecting confirmations.* In this step, pairs were asked to carefully read the card and its confirmations for *Compute an itinerary for concerts.* Pairs were working on a new feature divided into 4 confirmations (see Figure 1), presented in logical order. 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 with a confirmation.* The observer did not suggest any strategy to deal with this implementation task. 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. Each pair freely choose who was the driver and who the pointer. During the implementation the roles could be swapped.

In our study, the observer immersed himself and participated in step 4–joining in conversations and reading the card and its confirmations (to clarify them, if necessary). He did not disturb or change the natural setting of our study. An informal approach was used to probe possible issues in a naturalistic manner [28]. The observer possibly provided support for steps 1, 2, and 3. He could clarify concerns related to the study and/or to questions of used pre-questionnaires.

4. FINDINGS

Our analysis followed a standard approach (i.e., [5]). We identified the main themes emerging from our data.

4.1 Ethnographic Analysis

The goal of an ethnographic analysis is to find insights from recurrent themes. The meaning behind the observed activities must be inferred from the details of the collected data [37]. In this process, the observer must first reflect upon the experience gained in the immersion and used all of the data to recollect, revisit, and reconsider what was found, he then discussed them with the other researchers. In this case, the discussion was based on audio recordings and source code written by the participants, and other artifacts such as the data collected by the IDE plug-in.

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 then the theme was pursued. This kind of analysis proceeded iteratively as themes were identified, dropped, or validated and then confirmed. This approach required a considerable degree of effort, especially in the validation of potential themes with respect to the collected data. Potential themes were identified during and after conducting the study.

In the following subsections, we illustrate and detail the themes that emerged in our analysis.

[3]The developer in charge of writing code.

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

4.1.1 Dealing with legacy code

Before tackling a confirmation, pairs took some time to comprehend the legacy source code. Both students and professionals 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.

4.1.2 Discussion on the card

The pairs preferred to discuss an implementation plan for a given confirmation before passing to the subsequent phase of development. The discussion regarded the implementation details rather than the definition of test cases and identification of refactoring possibilities. For example, one pair of professionals had the following discussion regarding confirmation 1:

> *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 mental model is built using information such as syntactical structure and the control and data flow of existing source code [22]. 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 on the basis of this model. This point is very interesting and deserves future research. In theory a developer should be able to define a test only by imagining the interface of the code to be implemented rather than it entirely. Our work sets the stage for further work in this direction.

4.1.3 TDD phases

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.

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; *(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 4.1.2. This difficulty was less clear for professional developers. The students very often were not able to adequately define test cases of the right 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.

They performed refactoring only when it was needed 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) completed already when the production code was written and all tests were passed (i.e., the green phase).

4.1.4 Trusting test cases

Test cases were considered a silver bullet: 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.

4.1.5 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 4.1.3). Pairs were often getting ahead of themselves by adding source code to implement the next confirmation, before finishing the current one.

4.1.6 Working in pairs

As is customary, driver developers were in charge of approaching problems and implementing solutions, while pointer developers verified that drivers did not make any mistakes. We observed a slight difference between professionals and students when working in pairs. Whereas 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 mistakes 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.

4.2 A Quantitative Look Inside the Study

To better understand the themes described in Section 4.1, we also analyzed quantitative data gathered by Besouro. The Eclipse plug-in is able to identify different types of activities performed by the developers (defined in Table 1) and their duration based on the heuristics defined in [21]. For example, RG indicates that regression testing is performed without adding new code (test or production).

Table 2 presents the descriptive statistics for the type of development activities recognized during the study. In terms of total duration, the test first activity was predominant. When the activities are visualized in temporal order (i.e., in Figure 3), the majority of the pairs had a *cold start*. The duration of the initial activity was close to 50 minutes, indicating that pairs needed time to familiarize themselves with the legacy code before being able to progress, as noted in Section 4.1.1 and 4.1.2. Two pairs (CC and ZP) decided to start by writing tests for the existing code, in order to understand it.

Pairs tended to attempt to tackle more than they could handle, resulting in prolonged development cycles. TDD advocates that a card should be divided into manageable subtasks which should not take longer than 5 to 10 minutes to complete [2]. Although TDD was followed most of the time, the average duration of the test-first activities was 30 minutes. It appears that TDD was preceded by a tacit design phase. Moreover, refactoring was performed in a manner inconsistent with the expected TDD flow. In line with the what reported in Section 4.1.3, Table 2, and Figure 3, the refactoring activities were executed *in bulk*.

Although we do not attempt statistical inference, it can be observed from Table 3 and Table 4, that the professional pairs (DG, RB and ZP in Figure 3) had shorter, more granular activities than the students. The professionals had a more *agile* mindset, whereas the students might have been influenced by the exposure to waterfall and big design upfront strategies in their academic curricula, maturated before this course.

Although the professionals were able to apply TDD without the need of an upfront design, and able to divide a card into a set of manageable tasks, they adopted the same approach as the students for the refactoring.

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 [37]. 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 the execution of software evolution tasks

Pairs seemed not to be concerned about the internal qual-

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.

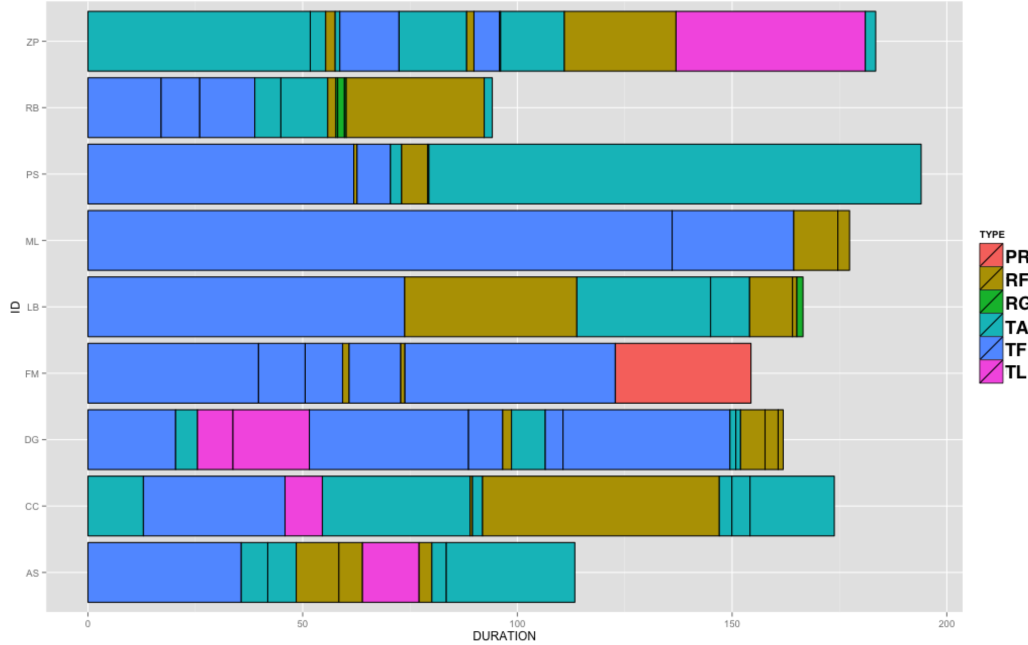


Figure 3: Development cycles for each pairs

Table 2: Descriptive statistics for 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

Table 3: Descriptive statistics for each type of development episode (Professionals)

TYPE	n	total	mean	median	min	max	stddev
TF	10	167.06	16.71	13.31	4.12	38.88	12.26
TA	13	123.73	9.52	5.08	1.07	51.77	1367
RG	4	2.65	0.66	0.42	0.23	1.58	0.61
RF	9	75.92	8.44	2.22	1.18	32.12	11.86
PR	-	-	-	-	-	-	-
TL	3	70.12	23.27	17.78	8.27	44.07	18.54

ity,[5] since they skipped refactoring and focused on completing a user card. The lack of concern regarding refactoring had also manifested during a focus group we had previously run in a similar setting [35]. In the participants' view, the only goal of TDD was to prompt them to write unit tests. The pairs needed to plan how to develop their solution in advance by building a mental model of the solution that would later be put in form of unit tests. In other words, we believe that the pairs approached the problem in a white box rather than a black box fashion, i.e., they conceived and developed unit tests according to the implementation details framed in their minds rather than the intended interface behavior.

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

Although the main impediment in the adoption of TDD is often reported to be the switching from test-last to test-first [8], it seemed that the real problem was switching from a plan-intensive mindset to a lightweight and flexible one. The issue of how to design in the context of TDD is considered a limiting factor [8], as well as the issue of losing sight of the big picture due to the lack of design [3]. 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. The professionals in our study, although new to the practice, were more disciplined with respect to the students. This was the main difference between

Table 4: **Descriptive statistics for each type of development episode (Students)**

TYPE	n	total	mean	median	min	max	stddev
TF	12	496.76	41.4	34.34	7.77	136.03	36.62
TA	14	280.01	20	7.86	2.35	114.63	29.55
RG	2	1.71	0.86	0.86	0.28	1.43	0.81
RF	14	147.51	10.54	4.24	0.55	55.13	16.37
PR	1	31.58	31.58	31.58	31.58	31.58	-
TL	2	21.88	10.94	10.94	8.70	13.18	3.17

students and professionals, and may be a deciding factor in the adoption/non-adoption of TDD. [24, 32]. 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. We believe that clearly adding such phase to the process, for example by integrating it with another practice,[6] would be beneficial in this regard.

Refactoring was not perceived as a step worthy of effort. It was often performed at last, and postponed for several iterations. This was observed for both student and professional pairs. In this regard, better support from the tool 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 [8, 35].

Pairs felt most comfortable with the green phase, i.e., when writing production code that would make a failing test pass. 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's boundaries. In other words, they wrote more code than necessary to just pass the test, giving priority to the mental model of the solution they built at the beginning of the task. 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 detrimental. What we observed is a mismatch between the mental solution and its implementation in the form of 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.

We observed another behaviour regarding TDD in the red phase; pairs never changed or removed existing unit tests. In this regards, we believe that refactoring should be enforced not only for production code, but also (and most importantly so) for unit tests, as they represent the core of the TDD practice. Although the process employed by pairs seemed to differ in several respects from the one proposed by Beck [2], a traditional test-last approach was deliberately followed only in few situations.

We previously described the shortcomings the pairs faced when applying TDD to a legacy system. Given Figure 3 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 until the end.

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

6. LIMITATIONS

In this section, we discuss possible limitations of our study. Regarding the timing of the study, the duration was approximately 3 hours for each pair, roughly half of a normal working day. Although we were able to observe how developers use TDD during the initial development phase of a new feature of a legacy system, the study omits the remainder of the process up to completion and implementation of the feature. Thus we may be excluding some important elements from our observations.

Regarding the pairs, neither students nor the professionals were experts in TDD. 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-certified company). Nevertheless, we acknowledge a difference between such settings and the ones in our study, in that usually a pair consists of one new developer, and one already experienced in TDD.

Finally, social factors should be taken into account when evaluating the findings (e.g., evaluation apprehension). In order to address with this concern, one of the authors (the observer) immersed himself in the study and used an informal approach to interact with pairs. To mitigate social factors, students were not evaluated based on the results they achieved in our study. Participation in the study was done on voluntary basis. Although the observer did not work together with the pairs, his presence was comparable to that of a project manager, and thus unlikely to have biased our study results.

7. CONCLUSION

In this paper, we report the results of an ethnographically-informed study conducted to investigate how students and professional developers apply TDD to software evolution tasks. We kept as close as possible to the natural settings in which developers, working in pairs, would normally carry on their everyday work activities. Based on collected data, we have identified and confirmed some themes that can be summarized in the following results: *(i)* refactoring is not performed as often as TDD requires and is considered less important than other phases, *(ii)* the most important phase is the implementation of production code, *(iii)* unit tests are not up-to-date, *(iv)* participants first imagine the source code to be implemented and then write test cases; and *(v)* students and professionals slightly differed in how they worked in pairs and in their application of TDD.

More qualitative studies are necessary to understand how TDD is currently practiced, as well as its shortcomings and strengths. We believe that ethnographically-informed studies are needed in companies which have adopted and are adopting the practice. Nevertheless, our results already set the stage for a number of future investigations. For example, future work should address how refactoring is performed during a TDD cycle, and how tools can support such activity. The traditional TDD cycle can be improved by adding an additional phase focused on splitting the task at hand into simpler and finer-grained sub-tasks, more apt to be framed in a unit test during the red phase. The developers tended to write more code than necessary to pass the unit test at hand; thus leaving part of the code uncovered by tests. It

stands to reason that better tool support for the green phase (e.g., coverage metrics for each TDD cycle) can be beneficial. In conclusion, we observed a shallow application of TDD by both professionals and novices. This can be problematic for researchers assessing the impact of TDD, since the practice they are observing may be substantially different from the one proposed by Beck [2]. One should therefore be cautious about the detrimental effects that may arise when TDD is exercised in such way.

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.

8. REFERENCES

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

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

[3] A. Begel and 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*, pages 255–264. IEEE, 2007.

[4] P. Beynon-Davies. Ethnography and information systems development: Ethnography of, for and within is development. *Information & Software Technology*, 39(8):531–540, 1997.

[5] P. Beynon-Davies, D. Tudhope, and H. Mackay. Information systems prototyping in practice. *Journal of Information Technology*, 14(1):107–120, Mar. 1999.

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

[7] J. Carver, L. Jaccheri, S. Morasca, and F. Shull. Issues in using students in empirical studies in software engineering education. In *Proceedings of the International Symposium on Software Metrics*, pages 239–. IEEE Computer Society, 2003.

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

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

[10] M. Fowler and J. Highsmith. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

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

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

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

[14] B. George and L. Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337–342, 2004.

[15] A. Geras, M. Smith, and J. Miller. A prototype empirical evaluation of test driven development. In *Software Metrics, 2004. Proceedings. 10th International Symposium on*, pages 405–416, 2004.

[16] A. Gupta and 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*, pages 285–294, 2007.

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

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

[19] R. Jeffrie. *Essential XP: Card, Conversation, Confirmation*. 2001.

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

[21] H. Kou, P. M. Johnson, and H. Erdogmus. Operational definition and automated inference of test-driven development with zorro. *Automated Software Engineering*, 17(1):57–85, 2010.

[22] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 7(4):341–355, 1987.

[23] A. Marchenko, P. Abrahamsson, and 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*, pages 13–22. Springer, 2009.

[24] G. Melnik and F. Maurer. A cross-program investigation of students' perceptions of agile methods. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 481–488. IEEE, 2005.

[25] M. Muller and W. Tichy. Case study: extreme programming in a university environment. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 537–544, May 2001.

[26] H. Munir, M. Moayyed, and K. Petersen. Considering rigor and relevance when evaluating test driven development: A systematic review. *Information and Software Technology*, 2014.

[27] M. Pancur, M. Ciglaric, M. Trampus, and T. Vidmar. Towards empirical evaluation of test-driven development in a university environment. In *EUROCON 2003. Computer as a Tool. The IEEE Region 8.*, pages 83–86, 22-24 September 2003.

[28] C. Passos, D. S. Cruzes, T. Dybå, and M. Mendonça. Challenges of applying ethnography to study software practices. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '12, pages 9–18. ACM, 2012.

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

[30] Y. Rafique and V. B. Misic. The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis. *IEEE Transactions on Software Engineering*, 39(6):835–856, 2013.

[31] H. Robinson, J. Segal, and H. Sharp. Ethnographically-informed empirical studies of software practice. *Inf. Softw. Technol.*, 49(6):540–551, June 2007.

[32] I. Salman, A. T. Misirli, and N. Juristo. Are Students Representatives of Professionals in Software Engineering Experiments? In *Procedings of International Conference on Software Engineering*, pages 666–676, 2015.

[33] F. Salviulo and 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*, pages 48:1–48:10. ACM, 2014.

[34] G. Scanniello and 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.

[35] G. Scanniello, S. Romano, D. Fucci, B. Turhan, and 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*, SAC '16, New York, NY, USA, 2016. ACM.

[36] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Softw. Eng.*, 25(4):557–572, July 1999.

[37] H. Sharp and H. Robinson. An ethnographic study of xp practice. *Empirical Softw. Eng.*, 9(4):353–375, 2004.

[38] H. Sharp, H. Robinson, and M. Woodman. Software engineering: Community and culture. *IEEE Softw.*, 17(1):40–47, Jan. 2000.

[39] F. Shull, G. Melnik, B. Turhan, L. Layman, M. Diep, and H. Erdogmus. What Do We Know about Test-Driven Development? *IEEE Software*, 27(6):16–19, 2010.

[40] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research*, pages 21–. IBM Press, 1997.

[41] M. Siniaalto and 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*, pages 275–284. ACM/IEEE Computer Society, 2007.

[42] B. Turhan, L. Layman, M. Diep, H. Erdogmus, and F. Shull. How effective is test-Driven Development. *Making Software: What Really Works, and Why We Believe It*, pages 207–217, 2010.

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

[44] K. Zieliriski and T. Szmuc. Preliminary analysis of the effects of pair programming and test-driven development on the external code quality. *Software engineering: evolution and emerging technologies*, 130:113, 2006.