A qualitative study on the first perceptions on applying TDD in medium size projects

Joelma Choma
National Institute for Space Research
São José dos Campos, Brazil
jh.choma@hotmail.br

Eduardo M. Guerra
National Institute for Space Research
São José dos Campos, Brazil
emguerra@gmail.com

Tiago Silva da Silva
Federal University of São Paulo
São José dos Campos, Brazil
silvadasilva@unifesp.br

***Resumo*—Test-Driven Development (TDD) is one of the most popular agile practices among software developers. We have investigated the software developers' perceptions when applying TDD for the first time. We performed a survey to gather (1) the main perceived benefits, (2) the difficulties encountered and, (3) the perception about the quality improvement of software as soon as developers experiment TDD as technique for software design and development. We analyzed the qualitative data using the Grounded Theory (GT) procedures. Our findings suggest that the main difficulties mentioned by novice in TDD refer to thinking about tests before the implementation and to develop through small steps. The programmers realized the improvement on software quality related to perceived benefits such as design improvement, high test coverage, and security in code refactoring.**

***Keywords*-test-driven development; test-first programming; TDD**

I. Introduction

Test-driven development (TDD) [1] is a technique for developing and designing software widely adopted by agile software development teams. TDD was proposed by Kent Beck in the late 1990's initially as part of Extreme Programming. Forcing the programmer to think about many aspects of the feature before coding it, this practice suggests an incremental development in short cycles by first writing unit tests and then writing enough code to satisfy them [2]. The cycles of TDD consists of following three simple steps repeatedly: (1) write a test for the next bit of functionality you want to add; (2) write the functional code until the test passes; and (3) refactor both new and old code to make it well structured [1].

Many studies have highlighted the benefits of TDD in software quality by comparing this practice with others software development approaches. Nonetheless, there are few studies exploring how this practice has been applied by practitioners. Many experienced programmers in TDD declare that it is a practice that increases the confidence in the result of the work, ensures fewer defects in code and good design, which allows them to provide better quality products [3–5]. However, beginners programmers in TDD may experience difficulties in applying this practice for the first time. As an effect of these initial difficulties, when developers start practicing TDD, they did not feel too productive [6]. Moreover, many of them state that it is necessary to struggle to understand the purpose of testing [7]. In contrast, most of them acknowledge the benefits when they spend far less time searching for bugs and/or trying to evolve the software.

In this paper, we have investigated the software developers' perceptions when applying TDD for the first time. The study aimed to answer the following research questions:

- RQ1: What are the difficulties pointed out by software developers during the first application of TDD?
- RQ2: What are the main benefits of TDD perceived by software developers who applied the technique for the first time?
- RQ3: What is the perception of the professionals about the software quality improvement using TDD?

The remainder of this paper is organized as follows. Section 2 provides an overview of the related work. Section 3 explains the research methodology. Section 4 presents the study profile. Section 5 presents the results of qualitative study. Section 6 provides conclusions, limitations and directions for future work.

II. Related Work

Empirical studies about the effects of TDD have been done from the industry and academia alike [8]. In the academia, a series of studies including surveys and experiments have been conducted with students through upper-level undergraduate, graduate, and professional training courses in order mainly to measure the productivity – effectiveness and efficiency comparing TDD to other development methodologies – and to verify the acceptance of TDD by both beginners and mature programmers (see e.g. [7,9–12])

Janzen and Saiedian [13] carried out five quasi-controlled experiments in the industry (3) and academia (2) in undergraduate and graduate software engineering courses – to verify whether TDD really improve software design quality. They designed a study to compare two approaches: TDD and Test-Last Development (TLD). In both approaches, programmers wrote automated unit tests and production code in short and rapid iterations. All groups used Java to develop Web applications in a single domain. All industry developers had computing degrees and a minimum of six years' professional development experience, while in the academia 59% of the participants were novice programmers. However, it is not clear whether some of participants were applying TDD for the first time.

George and Williams [14] complemented a study about efficiency and quality of test cases with a survey to gather perceptions that 24 professional pair programmers had about TDD practice. On average, the survey results indicate that 80% of professional programmers felt that TDD was an effective practice, and 78% stated that practice improves programmers' productivity. In addition, the results also indicated that the practice of TDD facilitates a simpler design, and that the lack of initial design is not an obstacle. However, for some of the programmers, the transition to the TDD mindset is the greatest difficulty.

Aniche et al. [6] conducted an open session about TDD through an informal chat with a practitioners group from industry during an Agile Meeting Conference. The goal of their study [6] was to assess the inexperienced practitioners opinion upon TDD, to understand how they were applying it in their routines, and to know about their point of view regarding to the effects of TDD in software design. According to these researchers, the audience of this study was classified as an experienced group of software developers who were starting to practice TDD – wherein only one of ten participants had never practiced TDD before.

In this study, we are mainly investigating the perceptions of programmers when applying TDD for the first time.

III. Research Method

In order to investigate the software developers' perceptions when applying TDD for the first time, we conducted a survey. Survey are an empirical strategy often employed for collecting information aiming to describe, compare or explain, for example, effects and opinion of a sample of users about one particular technique. The results from the survey are then analyzed, and its conclusions may be generalized to the population from which the sample was taken [15].

Through this survey, we gathered quantitative and qualitative data applying a questionnaire with open and closed-ended questions. We analyzed the qualitative data based on the Grounded Theory (GT) [16]. GT is a qualitative research method that uses a systematic set of procedures in order to develop and inductively derive a theory grounded from a given phenomenon. First, the main elements are identified from a dataset, and afterwards they are encoded. The coding process suggested by [16] is divided into three steps: open coding, axial coding and selective coding. In the present study, however, we performed only the two first steps, since it was possible to obtain the answer to our research questions after the execution of the open and axial coding phases.

IV. Study profile

In this section, we present the context in which this study was carried out, as well as the participants' profile and the characteristics of the projects where these participants applied TDD.

A. Sujects and context

The survey was carried out with 12 students of master and doctoral degree in a graduate course of Applied Computing from the National Institute of Space Research in Brazil. In the Agile Projects discipline, all the subjects received training about TDD based on Java programming language and JUnit framework.

During the course, the subjects were required to develop an application of any type using TDD. In addition, they could freely to choose the programming language, development environment, and the unit tests tools. At the end of the experience, the subjects answered a questionnaire to provide information about their projects, as well as about the development technique that was applied.

Table I shows the questions related to how long the subjects had experience in programming, how long they had experience with chosen language for the development of the application, and whether they had previously used TDD for developing and designing software. As we may notice from last question, none of the participants had applied TDD before.

Tabela I
Subjects profile

Subject #	Programming experience (years)	Experience on programming language (years)	Have you ever used TDD before?
S1	2	2	no
S2	3	1	no
S3	5	2	no
S4	5	2	no
S5	5	0,3	no
S6	15	10	no
S7	9	6	no
S8	13	0,5	no
S9	10	9	no
S10	14	0	no
S11	12	11	no
S12	12	10	no

Regarding the experience in programming, participants had at least 2 years of experience in software development, and 50% of them had more than 9 years of experience. With regards to the experience with the chosen programming language for the project, 17% of participants (2 of 12) had almost no experience, while 33% of participants (4 of 12) had from 1 to 2 years experience, and 42% participants (5 of 12) had a long time of experience with the language. There was only one participant who had no experience with the chosen language, however, he was one of the most experienced programmers who used the opportunity to learn a new programming language.

B. Projects characterization

The applications were developed for different purposes. Table II shows a brief description of each one.

The majority of the projects (50%) were developed in Java. The other programming languages used were C++, Python, PHP and JavaScript. As to the project type, seven projects were part of academic work in the area of research of the developer,

Tabela II
PROJECTS CHARACTERIZATION

#	Project	Programming language	Type *	Was it partially implemented?
S1	System for decoding avionics data bus	Python	1	yes
S2	Model-based Testing Tool	Java	3	yes
S3	Annotation Validation System	Java	3	yes
S4	System for E-Commerce	Java	2	yes
S5	Implementation of parametric curves	C++	3	no
S6	System for generation of geospatial data catalog	C++	1	no
S7	Management system for cafeterias and restaurants	C++	2	yes
S8	Extraction of historical software metrics	JavaScript	3	no
S9	Web service for conversion of XML models	Java	3	no
S10	Implementation of training algorithm	Java	3	no
S11	Software for acquisition of weather forecast data	PHP	2	no
S12	Drawing application for digraphs	Java	3	no

* 1 - Industrial Project | 2 - Personal Project | 3 - Academic Project

three projects came from industry professionals, and other two were personal projects. Of all the projects, five of them were already partially implemented.

The subjects provided the metrics of their software projects. Table III presents the metrics related to (i) the size of the each application That can be measured by the number of classes, number of methods, and number of lines-of-code (LOC) in the implementation and tests, (ii) the development effort measured by total of LOC (implementation + test) divided per time spent in hours, and (iii) the percentage of test coverage. Additionally, the subjects mentioned which tools they used to support the unit tests.

V. QUALITATIVE ANALYSIS

Aiming to answer our research questions, we have inserted in the questionnaire the following questions:

- Q1 - What were the main difficulties did you have in the development using TDD? Please, tell us about how you really felt during the software development.
- Q2 - What are the main benefits did you realize with TDD? Please, tell us about how you really felt when using it.
- Q3 - By comparing the code of this project with the code that you have developed from other development methods, have you noticed an improvement in the software quality? Please, try to show us objectively what evidence of this improvement you could highlight.

In this section, we present the results of the qualitative analysis of the participants' answers to these questions[1].

A. Difficulties in applying TDD

Analyzing the participants' answers to the issue related to difficulties in developing the software through TDD, we identified four categories of difficulties: (i) lack of practice, (ii) doubts and lack of confidence, (iii) lack of patience to take small steps, and (iv) control and decision making.

There is little empirical evidence showing if TDD in fact increases or decreases productivity [17]. In our study, at least 50% of programmers attributed a low productivity due to lack of practice: *"Lack of practice in the use of TDD, at first, delayed software development"* [S11]. One of the participants pointed out that his difficulty was to keep pace with TDD: *"Unconsciously, I was jumping steps"* [S8]. TDD requires a mindset transition, because it prescribes behaviors which are not common among developers: *"Practicing TDD requires a new habit of thinking about the test before encoding the functionality"* [S3]. In addition, it is very important that programmers understand what is the real purpose of the tests to avoid unnecessary work: *"In developing the first tests, I did many unnecessary tests"* [S4].

Some of programmers stated that their difficulties were due to doubts and insecurity: *"The most difficult thing for me was thinking about the test before having the functionality implemented, that is, how to start?"* [S3]. One recurrent doubt was: *"How do you know where to start the project, when you do not have any class or interface defined"* [S6]. The practice to think about the test before implementing the code without having a design up front can result insecurity: *"I did not know if I should developed the TDD right away, or if I should seek by a class model more appropriate for the problem"* [S5].

Once test methods should be easy to read and understand, Kent Beck suggests "baby steps" [1]. This practice consist in to write tests for the least possible functionality, simplest code to pass the test, and always do only one refactoring at a time. Discussing about how to do baby steps in real world [6], some of programmers believe that "baby steps" should be done all the time and for every situation, while other programmers claim that there is no need to go that slow. Farther, with more experience, you can decide by bigger steps.

We can noticed that, some programmers had difficulty practicing "baby steps": *"I wanted to implement the main functionality as fast as possible"* [S5]. Especially novice programmers can be more anxious: *"I did not have the patience to do the baby steps, I found the process a bit slow"* [S4]. Further, it is easier to measure the steps when the developer has more experience: *"It was not possible for me to develop my application in baby steps"* [S2].

Working with TDD requires discipline and control:*"I ne-*

[1]The authors translated the participants' responses originally in Portuguese into English.

Tabela III
SOFTWARE METRICS

#	Classes	Methods	LOC	Test LOC	Spent hours	Effort *	Coverage %	Test tools
S1	5	17	132	246	30	12.60	100	PyCharm
S2	11	42	2,669	400	9	341.00	96.2	JUnit
S3	7	27	103	136	50	4.78	85.3	JUnit
S4	47	236	1,214	959	60	36.22	95.8	JUnit
S5	9	125	359	603	30	32.07	96.7	GTest
S6	38	296	1,627	1,619	72	45.08	75.7	Google Test
S7	146	2,320	11,316	2,762	176	79.99	80.0	QTestLib
S8	6	85	830	654	15	98.93	98.7	Mocha
S9	9	27	843	307	16	71.88	87.2	JUnit
S10	4	28	285	140	9	47.22	98.4	JUnit
S11	9	32	463	101	14	40.29	71.4	PHPUnit
S12	26	108	1,109	579	32	52.75	57.6	JUnit

* Total of LOC per spent hours in implementation

eded to police myself to avoid implementing something that did not yet done the test fail" [S6]. Regarding the creation of tests a difficulty may be to decide on how to group the tests: *"I did not know if I isolated each test or joined test in one, which corresponded to the same scenario implemented by the class"* [S5].

Other difficulties were mentioned by the participants. However, they were more related to other technical difficulties than related to the development method itself, such as: the lack of IDE support [S4], configuration of tools to perform unit testing with C ++ [S6], doubts about how to implement a given application requirement [S12]. There was still one participant who stated that he had no difficulty in applying TDD [S7].

B. Key benefits of TDD

Concerning the perceived benefits on TDD, we identified seven categories: (i) security and confidence, (ii) better understanding of requirements, (iii) design improvement, (iv) lean coding, (v) easy maintenance, (vi) greater coverage of tests and encouraged refactoring, and (vii) bug detection and reduction.

Overcoming the initial difficulties, programmers gain greater security in development and recognize this benefit: *"Greater security in the implementation of new and small functionalities of the software"* [S3]. Mainly because everything that is built already has a ready test: *"Project becomes more secure, since everything that has been implemented has already been tested [...] tests that were previously created ensure the structure and consistency of the code"* [S2]. The same way confidence increases: *"There is an increase in confidence in development"* [S12] and *"I could know that an exception would actually be thrown, if something wrong happened"* [S5]. Trust can attributed to baby steps: *"Thinking about small pieces one at a time, this avoided stopping the bigger problem"* [S12].

Due to the necessary reflection about features for write tests first, the developer begins to better understand each requirement: *"During test writing, the functionality becomes clearer"* [S9], and *"During the creation process of testing, I tried to think lean form in relation to the requirements"* [S10]. Certainly, this practice also allows: *"Anticipate some problems in the development"* [S11].

Programmers perceive improvements in design due to "baby steps" : *"Because of the baby steps I was forced to think in a simpler way, and the classes have been less coupled"* [S4], where *"The integration between the separately created modules is done in a less complicated way"* [S2]. Further, decoupling is also perceived as a effect of the use of Mock Objects: *"Decoupling of classes by the use of dependency injection [is a] reflection of the use of Mocks"*. Mock Objects allows developers to write the code under test as if it had everything it needs from its environment, guiding interface design by the services that an object requires, not just those it provides [18].

Many programmers pointed out other benefits related to code quality: *"The code became less complex"* [S2],*"[...] clean, clear and better organized"* [S1]. Also, they have highlighted some of the causes that led to clean and lean coding: *"implementation objectivity"* [P9], *"Encode only what you need"* [S6], *"Everything that was programmed was used"* [S2], and *"I discarded methods that would be irrelevant to the project"* [S11]. A good quality code certainly makes software maintenance easier: *"The changes tend to be simpler"* [S7], and *"Easy to adding or removing features"* [S1].

Programmers recognized that a good test coverage leads to less traumatic refactoring: *"Due to great coverage of tests achieved by the use of the TDD, we were able to include new features and then run the test suite to see if it did not break some thing"* [S4]. Hence, *"The refactorings were less fearful"* [S5]. It was possible *" Easily view the already tested code keeps passing the tests"* [S12]. This effect encourages code refactoring towards continuous improvement: *"we are encouraged to change the code when necessary, because the tests give more assurance about a greater guarantee of operation of the whole"* [S12]. The higher the test coverage, the more security there will be during the refactoring process: *"Using TDD you get a higher level of coverage with tests, so you feel safer to do refactoring and include new features. I really felt confident about doing the refactorings and evolving the patterns. The classes that I developed with TDD had almost 100% coverage, while others did not"* [S4].

Finally, regarding the detection and reduction of bugs the programmers stated that some benefits of TDD are: *"Faster*

identification of failures introduced in the code" [S3], *"Less time spent on debugging code execution* [S12], and *"The possibility to validate the code in small parts since the beginning of the development allows that the emergence of bugs in production reduces greatly"* [S10].

C. Perceived improvement in software quality

All participants noticed improvement in the software quality that was developed through TDD. We can note that the highlighted evidences by them were directly related to perceived benefits. We identificated three categories of evidences about improvement in software quality: (i) Code quality: clean, lean and readable, (ii) decrease in bugs, and (iii) test coverage and refactoring.

Almost all participants (9 of 12) highlighted evidence related to the quality of the code. The quality of the code - clean, lean and readable - has an impact on software design: *"Without doubt, the code has improved greatly, an evidence is the amount of "ifs"that has decreased greatly, so the readability of the code has improved"* [S4], *"Apparently, the classes got leaner and less coupled, with less unused code"* [S6], *"he code became more modular, with useful code and better defined features"* [P8], and *"The methods got smaller [...] The code was much easier to understand"* [P2]. In addition, other participants highlighted the quality of the code with a positive impact on software maintenance: *"I would emphasize maintainability because of the better organization and clarity of the written code. Looking at the tests it is possible to quickly identify the relations between classes and the operation of the methods"* [P1], and *"I would point out that test cases also have a software documentation role, and such documentation maintains consistency with source code"* [P12].

Only one programmer could not identify improvements in code quality using TDD compared to Test Last [S3]. Nevertheless, he noted a greater comfort in the implementation of new features, since the tests were implemented and due a reduction of the time to identify faults introduced in the code. This evidence refers to the category of "decrease in bugs". The number of bugs is an important measure of software quality: *"I noticed that the final version (...) had fewer bugs than at other times"* [S5], and *"Many defects that I had not seen in the first implementation, I have found out through the TDD"* [S2]. It is important to emphasize that a greater effort to create the tests before implementation can be offset by less spent time during bug fixes: *"The time spent fixing bugs was much smaller than it usually was in other projects"*[S7].

Regarding the third type of evidence, participants perceived improvements in software quality due to tests wide coverage supporting a safe refactoring: *"Automated testing favors change without fearing to "break"other parts of the software"* [P12], and *Refactoring securely methods, ensuring its operation"* [Q9].

VI. Conclusion, Limitations and Future Work

TDD has been one of the most experienced development techniques within software engineering studies and agile methodologies where one of the main concerns has been to evaluate its efficiency and effectiveness in relation to other techniques. Comparing it with most of the related work, our study was not restricted to a controlled environment. Instead, it involved a variability of projects with different sizes and purposes, and using different programming languages and support tools. From this context, we presented results of a survey with programmers facing their first experience with TDD.

The main difficulties mentioned by novice in TDD refer to thinking of unit tests before implementation, and develop through small steps, especially for experienced programmers. Like any new practice, TDD will involve a learning curve which have an impact on productivity. Thereby, studies aimed at evaluating the efficiency and effectiveness of TDD need to consider the programmers' experience, once novice programmers in TDD need time to understand, to familiarize themselves, to gain practical experience, and thereafter improve their productivity.

Regarding benefits, programmers realized that TDD helps create a better design from classes less coupled and a code cleaner, clearer and better organized. In addition, they emphasized that a greater test coverage and refactoring encouraged by TDD ensures a software with fewer bugs and easier to maintain. Such benefits are mentioned once again to highlight the improvement perceived by them as to product quality.

The main limitations of this study are related to the sample size and the professional experience of each participants. The sample may not be representative in order to generalize the findings on this paper. Thus, this survey needs to be replicated with other professional programmers in order to confirm our results. Despite none of the programmers had experience with TDD before, it may be that professional experience of each one has influenced their particular opinions and perceptions. This issue can be better analyzed in future works.

Acknowledgment

We thank everyone who participated in the survey.

Referências

[1] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2002.

[2] E. Guerra and M. Aniche, "Achieving quality on software design through test-driven development," *Software Quality Assurance: In Large Scale and Complex Software-intensive Systems*, p. 201, 2015.

[3] L. Crispin, "Driving software quality: How test-driven development impacts software quality," *Ieee Software*, vol. 23, no. 6, pp. 70–71, 2006.

[4] M. F. Aniche and M. A. Gerosa, "Most common mistakes in test-driven development practice: Results from an online survey with developers," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*. IEEE, 2010, pp. 469–478.

[5] F. Shull, "Data, data everywhere..." *IEEE Software*, vol. 31, no. 5, 2014.

[6] M. F. Aniche, T. M. Ferreira, and M. A. Gerosa, "What concerns beginner test-driven development practitioners: a qualitative analysis of opinions in an agile conference," in *2nd Brazilian Workshop on Agile Methods*, 2011.

[7] C. Desai, D. Janzen, and K. Savage, "A survey of evidence for test-driven development in academia," *ACM SIGCSE Bulletin*, vol. 40, no. 2, pp. 97–101, 2008.

[8] R. Jeffries and G. Melnik, "Guest editors' introduction: Tdd–the art of fearless programming," *IEEE Software*, vol. 24, no. 3, pp. 24–30, 2007.

[9] D. S. Janzen and H. Saiedian, "A leveled examination of test-driven development acceptance," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 719–722.

[10] A. Gupta and P. Jalote, "An experimental evaluation of the effectiveness and efficiency of the test driven development," in *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 285–294. [Online]. Available: http://dx.doi.org/10.1109/ESEM.2007.20

[11] J. H. Vu, N. Frojd, C. Shenkel-Therolf, and D. S. Janzen, "Evaluating test-driven development in an industry-sponsored capstone project," in *Information Technology: New Generations, 2009. ITNG'09. Sixth International Conference on*. IEEE, 2009, pp. 229–234.

[12] L. Huang and M. Holcombe, "Empirical investigation towards the effectiveness of test first programming," *Information and Software Technology*, vol. 51, pp. 182–194, 2009.

[13] D. Janzen and H. Saiedian, "Does test-driven development really improve software design quality?" *Ieee Software*, vol. 25, no. 2, pp. 77–84, 2008.

[14] B. George and L. Williams, "A structured experiment of test-driven development," *Information and software Technology*, vol. 46, no. 5, pp. 337–342, 2004.

[15] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[16] J. Corbin and A. Strauss, *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[17] M. Pančur and M. Ciglarič, "Impact of test-driven development on productivity, code and tests: A controlled experiment," *Information and Software Technology*, vol. 53, no. 6, pp. 557–573, 2011.

[18] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, objects," in *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. ACM, 2004, pp. 236–246.