Dissertation

Master in Computer Engineering – Mobile Computing

ANALYSIS OF THE IMPACT OF TEST BASED DEVELOPMENT TECHNIQUES (TDD, BDD, AND ATDD) TO THE SOFTWARE LIFE CYCLE

Luis Alberto Cisneros Gómez

Leiria, September 2018

This page was intentionally left blank

Dissertation

Master in Computer Engineering – Mobile Computing

ANALYSIS OF THE IMPACT OF TEST BASED DEVELOPMENT TECHNIQUES (TDD, BDD, AND ATDD) TO THE SOFTWARE LIFE CYCLE

Luis Alberto Cisneros Gómez

Dissertation developed under the supervision of Catarina Isabel Ferreira Viveiros Tavares dos Reis, Professor at the School of Technology and Management of the Polytechnic Institute of Leiria and co-supervision of Marisa da Silva Maximiano, Professor at the School of Technology and Management of the Polytechnic Institute of Leiria.

Leiria, September 2018

This page was intentionally left blank

Acknowledgements

First at all, I want to thank to God for giving me intelligence, fortitude and illuminating this academic progress.

To my mother, Kevin, my grandmother and my sisters; who have given me their unconditional affection and, always have been supporting me to continue forward in the more difficult moments of my life.

At the Polytechnic of Leiria for offering me a space where I could complete my master´s degree studies.

To my thesis supervisors: Catarina Reis PhD. and, Marisa Maximiano PhD., who have collaborated in the development of my work.

To my thesis coordinator, Antonio Quiña MsC., who allowed me to execute the experimentation in the Universidad Técnica del Norte.

To Filipe Mota Pinto PhD., for his orientation to decide my specialization studies.

To Oscar Dieste PhD., for his cooperation in the planning of the experiment.

For all the teachers who have been part of my academic training, sharing me their knowledge and experiences.

To my friends with whom I shared experiences, opinions, confidences, and celebrations.

For all you, thank you very much.

Luis Alberto Cisneros Gómez

This page was intentionally left blank

Resumen

Dentro del mundo del desarrollo de software, existe la necesidad constante para crear productos de calidad que sean capaces de enfrentar los retos en un ambiente de requerimientos cambiantes. La industria en esta área es muy consciente de esto, y para ello, hace uso de metodologías de desenvolvimiento de software como: tradicional o ágil. El desarrollo ágil representa un alejamiento de los enfoques tradicionales, permitiendo la creación de aplicaciones de manera incremental e iterativa, y que se ajustan a los requisitos cambiantes de los clientes. Por esta razón, últimamente las empresas han adoptado el uso de sus prácticas y técnicas, ej.: Test-Driven Developement (TDD), Acceptance Test-Driven Developement (ATDD), Behavior-Driven Development (BDD), entre otras.

Estas técnicas prometen principalmente mejorar la calidad de software y la productividad de los programadores; por lo cual, se han realizado varios experimentos, especialmente de TDD, dentro de la academia como en la industria; que muestra resultados variantes (unos con efectos positivos y otros no tanto). Además, programadores expertos, han puesto en práctica estas técnicas en la creación software logrando resultados satisfactorios debido a las ventajas ofrecidas por su utilización. Este trabajo tiene como objetivo principal verificar el impacto producido por las técnicas de desarrollo de software basado en pruebas (TDD, ATDD y BDD), analizando sus primordiales promesas. Se ha realizado una investigación sobre estas técnicas, con la intención de entender las fortalezas y debilidades de cada una de ellas.

Con la finalidad de observar la efectividad de TDD y BDD, se ejecutó un experimento en la academia, tomando en cuenta una formación y un entrenamiento apropiado para implantar el conocimiento suficiente sobre las mismas. A partir de los resultados obtenidos, fue posible comprender que las técnicas estudiadas aseguran la calidad del producto desarrollado y mejoran la productividad de los programadores; validando sus efectos dentro del desarrollo de software.

Palabras clave: Ingeniería de Software, Calidad de Software, Testing, Test-Driven Development, Behavior-Driven Development, Acceptance Test-Driven Development.

This page was intentionally left blank

Resumo

No mundo do desenvolvimento de software, existe uma necessidade permanente de criar produtos de qualidade capazes de enfrentar desafios em ambientes com necessidades diversas. A indústria nesta área está muito consciente disso, e assim, faz uso de metodologias de desenvolvimento de software: tradicional ou ágil. O desenvolvimento ágil distancia-se das abordagens tradicionais, permitindo a criação de aplicações de forma incremental e iterativa e, assim, ajustando-se às necessidades variáveis e mudanças dos clientes. Por essa razão, as empresas adotaram, recentemente, o uso destas práticas e técnicas, como por exemplo: Test-Driven Development (TDD), Acceptance Test-Driven Development (ATDD), Behavior-Driven Develoment (BDD), entre outros.

Estas técnicas prometem melhorar a qualidade do software e a produtividade dos programadores; portanto, foram realizadas muitas experiências no mundo académico e na indústria, especialmente usando TDD; que mostram resultados variáveis (alguns com efeitos positivos e outros não tanto). Além disso, há registo de programadores séniores que colocaram essas técnicas em prática na criação de software, obtendo resultados satisfatórios devido às vantagens oferecidas pelo seu uso. O principal objetivo deste trabalho é verificar o impacto produzido pelas técnicas de desenvolvimento de software baseadas em testes (TDD, ATDD e BDD), analisando suas promessas primordiais. Foi realizada uma pesquisa bibliográfica com a finalidade de compreender os pontos fortes e fracos de cada uma dessas técnicas.

Com o intuito de observar a eficácia do TDD e do BDD, planeou-se uma experiência num cenário académico, levando em consideração toda a formação e treino adequados para a necessária implementação dos conhecimentos. Com os resultados obtidos foi possível entender que as técnicas estudadas garantiram a qualidade do produto desenvolvido e melhoraram a produtividade dos programadores; validando os seus efeitos no desenvolvimento de software.

Keywords: Engenharia de Software, Qualidade de Software, Testes, Test-Driven Development, Behavior-Driven Development, Acceptance Test-Driven Development.

This page was intentionally left blank

Abstract

Within the world of software development, there is a permanent need to create quality products that are capable of facing challenges in environments of changing requirements. The industry in this area is aware of this, and so, it makes use of software development methodologies such as: traditional or agile. Agile development represents a distancing from traditional approaches, allowing the creation of applications incrementally and iteratively and, thus, adjusting to the changing requirements of customers. For this reason, companies have recently adopted the use of its practices and techniques, e.g.: Test-Driven Development (TDD), Acceptance Test-Driven Development (ATDD), Behavior-Driven Development (BDD), among others.

These techniques promise mainly to improve the quality of the software and the productivity of the programmers; therefore, many experiments, especially using TDD, have been made within the academy and the industry; which shows variant results (some with positive effects and others not so much). In addition, expert programmers have put these techniques into practice in software creation, getting satisfactory results due to the advantages offered by its use. The main objective of this work is to verify the impact produced by the techniques of software development based on tests (TDD, ATDD and BDD), analyzing its primordial promises. A literature research has been conducted in order to understand the strengths and weaknesses of each one of these techniques.

With the intention of observing the effectiveness of TDD and BDD, an experiment was planned in an academic scenario, considering education and appropriate training to implement enough knowledge about them. With the results obtained, it was possible to understand that the techniques studied ensured the quality of the product developed and improved the productivity of the programmers; validating its effects within software development.

Keywords: Software Engineering, Software Quality, Testing, Test-Driven Development, Behavior-Driven Development, Acceptance Test-Driven Development.

.

This page was intentionally left blank

List of figures

This page was intentionally left blank

List of tables

This page was intentionally left blank

List of acronyms

ATDD	Acceptance Test-Driven Development
BDD	Behavior-Driven Development
DDD	Domain Driven Design
DEV	DEVeloper
ERP	Enterprise Resource Planning
JVM	Java Virtual Machine
LD	Lean Development
QA	Quality Assurance
RAD	Rapid Application Development
REST	REpresentational State Transfer
RUP	Rational Unified Process
SOLID	Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion
TDD	Test-Driven Development
XP	eXtreme Programming

This page was intentionally left blank

Table of Contents

1. Introduction

In this section, the aspects that comprise this work, the scope in which it is developed, the motivation for its realization, and a short description of the organization of the document are presented.

1.1. Relevance

Nowadays, it´s important that software development companies have a methodology that is prepared to ensure the quality of the product and allows and allows its maintenance over time. Software development presents many challenges to work groups, ranging from understanding the needs of customers, to providing solutions that guarantee the expectation of users and improve productivity throughout the software development life cycle. In response to this and within "agile", techniques that provide flexibility within the creation of software are presented, allowing programmers to focus more on the coding and not much on the maintenance of the code, fixing bugs, among others.

This dissertation aims to study the techniques of software development based on tests: Test-Driven Development (TDD), Acceptance Test-Driven Development (ATDD) and Behavior-Driven Development (BDD); in order to analyze the impact of the application of these techniques in the quality of the software produced. Therefore, the research will focus on: conduct a literature review about techniques, describe their characteristics, compare these techniques, analyze the main tools found in the market, and execute an empirical experiment to show the impact in the software quality and developer productivity with the mentioned techniques.

1.2. Context

There are many software development methodologies, which have advantages and disadvantages as well. The choice of an appropriate methodology for a specific project requires a good analysis and evaluation of the characteristics of the project such as commercial needs. Some well-known methodologies are: Agile, Crystal Methods, Lean Development (LD), Rapid Application Development (RAD), Rational Unified Process (RUP), Waterfall and V-Model [1]; and the current trend of use is inclined by Agile and

its practices (Extreme Programming, Scrum, Kanban, TDD, ATDD, BDD, among others).

Agile methods of software development emerged as an alternative to traditional approaches, with the purpose of spending less time on documentation and guiding problem solving in an interactive and iterative way. In addition, they describe a set of principles used by other methodologies, emphasizing: adaptive planning, evolutionary development, early delivery, continuous improvement and flexible response to change [2], [3].

In the software industry some projects do not offer the expected features and value, that the business owner needs. The 2016 "CHAOS Report" study [4], executed by the Standish Group, shows that 52% of the projects were deficient (exceeded the budget or did not deliver all the requested characteristics); while 19% of the projects failed completely. This shows that it’s common to have a waste of effort that translates into the lack of quality of the software product developed, that does not solve the problem for which it was created, resulting in the loss of both money and time. So, it is very important to manage our resources carefully and, try to find the best solution when choosing the software development process. The benefits of the TDD, ATDD and BDD techniques try to mitigate these problems.

1.3. Motivation

Software plays an important role in people's lives. Throughout time it has managed to spread, so that almost everything we use, contains some sort of application or software product; therefore, the number of software projects and also their complexity, are increasing.

Along my short professional experience, I observed some factors in software projects such as: the conflict of communication and comprehension between technical and non-technical actors, how tedious it is to build software tests after developing the software, the difficulty of non-technical actors to capture the technical aspects, the validation of the requirements to ensure that the actual developments, are exactly what the project owner wants; among others. For this reason, and based on my experience using TDD, I consider very important to study the techniques TDD, ATDD and BDD; due to the fact

that these software methods try to increase the chances of success using processes to guide the stakeholders of the project.

1.4. Document Structure

The present dissertation is organized into six chapters, that will be presented below:

Chapter 1, the current section, has presented the relevance, context and motivation of the work.

Chapter 2 contains the bibliography related to the dissertation; that is, it talks about software testing, agile development and TDD, ATDD and BDD techniques. An expert or more experienced professional that is familiar with the basic concepts of Software Testing can skim or skip this chapter.

Chapter 3 provides a review of the papers that are relevant to this research on evidence-based development techniques.

Chapter 4 presents the empirical experiment about the application of test based development techniques, to show the impact in the quality of the software and in the developer productivity.

Chapter 5 provides the statistical analysis of the data obtained from the experiment and, responds to the suggested hypothesis.

Chapter 6 contains the conclusions and recommendations of the study, its limitations and future research opportunities.

2. Software Testing

This chapter is about software testing; at the beginning, its importance is mentioned and, then some essential concepts are presented such as principles, types, and methods, among others. In addition, it talks about agile testing and evidence-based development techniques.

2.1. Fundamentals of Testing

Software testing is the process of executing a program or application with the intention of finding software errors [5]. They are part of a quality assurance process that measures the effectiveness of the software in terms of defects in functional and non-functional requirements (reliability, usability, efficiency, maintainability and portability).

The need for its existence is based on human nature, software products (applications) are generated by work groups made up of ordinary people (programmers, business analysts, systems analysts, testers and other collaborators); therefore, errors can be made within their life cycle, causing malfunctions and generating failures that can be costly or dangerous. Not all of them are obvious at first sight and can be inserted in [6]: the stage of survey and requirement analysis, the coding stage (business logic or user interface), and / or documentation (electronic or printed).

The tests are the precise method to verify and validate the software in its life cycle: they provide important information about their status, point out defects and errors, guarantee product quality, and increase the client's reliability and satisfaction with the product.

Testing principles

The following are principles that have been suggested in the last 40 years and offer common general guidelines for all tests [5]:

- ***Principle 1: Tests show the presence of defects*** - Through the application of the software tests the presence of errors can be demonstrated, but they can't prove the absence of the same; that is, they reduce the likelihood of hidden defects;
- ***Principle 2: The exhaustive tests do not exist*** - Testing all combinations of inputs and conditions is impossible (except in trivial cases). Instead of conducting extensive tests, the tests should be focused on risks and priorities;

- ***Principle 3: Early tests*** **-** The test activities should start as soon as possible in the software life cycle or system development, in order to find, correct and prevent defects without forgetting to focus on the defined objectives;
- ***Principle 4: Grouping defects*** **-** Normally, most defects detected during pre-launch tests and most operational failures are concentrated in a small number of modules. This is a phenomenon that indicates errors tend to cluster, due to complex areas and/or delicate areas in their development;

  It is important to mention that groupings will not always be the same, so it is important to focus on key points as an information base to evaluate the risks and plan the tests;
- ***Principle 5: Paradox of the pesticide*** **-** The continuous repetition of the same test cases will eventually stop finding new defects. To overcome this "pesticide paradox", test cases must be periodically examined and, new tests written in order to evaluate different parts of the software in order to detect more defects;
- ***Principle 6: The tests depend on the context*** **-** The tests are carry out differently depending on the context applied; that is, not all software systems carry the same level of risk and, not all problems have the same impact when they occur. For example, the method to test critical security software varies with respect to an e-commerce site;
- ***Principle 7: Fallacy of absence of errors*** **-** If when building a system and creating it, errors are detected and corrected; It does not mean that this software can be cataloged as excellent. Finding and correcting the defects will not help if the built system is not usable, does not meet the expectations, and needs of the users. Individuals and organizations that buy and use software as daily help are not interested in the defects or their quantity, unless they are affect directly by the software's instability.

Software test levels

Software tests are usually made at different levels throughout the development and maintenance processes; these are distinguished according to the purpose of the test (a module, group of modules or complete system). The TABLE I describes the existing test levels.

TABLE I. Software test levels

Test level	Description
Component tests	Also known as unit tests, aim to locate defects and check the operation of software modules, programs, objects, classes, etc., which can be tested separately [7]. The programmers are the ones who write the code of a program, but not always, they are those who carry out the unit tests.
Integration tests	Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing [7]. There are two levels of integrations tests: component integration and system integration. In addition, various approaches are used for integration testing: Incremental Approach, Bottom-up Integration, Smoke Testing, Top-down Integration, Regression Testing, and Sandwich Integration (combination between top-down and bottom-up).
System test	The system tests are designed to test the behavior of a system or product in its entirety [8]. They also evaluate external interfaces to other applications, utilities, hardware devices or operating environments.
Acceptance tests	Designed to verify that the system meets the requirements demanded by the user. The acceptance tests are focused on building trust, to demonstrate that the system or product is ready for the deployment of its "x" version, and do not concentrate on the search for defects [5]. These can occur at different moments of the life cycle, thus adopting the following forms: User acceptance tests, Operational acceptance tests, and Evidence of contractual acceptance and regulations tests.

Software test types

When creating a software or an application, it is important to perform several types of tests to ensure that the product is complete, safe and efficient. They are explaining in the TABLE II.

TABLE II. Software test types

Test type	Description
Functional testing	"Functionality" of the system is understood as "what the system does" [6]. These tests are intended to analyze and exercise in different ways the functions of a component of an application in order to check if it works well under normal circumstances, or in unusual ways.

Non-functional testing	Non-functional tests evaluate the quality characteristics of a system, subsystem or component [6]. Something "non-functional" refers to aspects that may not be related to a specific function or user action, such as scalability, security, performance, portability, among others [5].
Structural Testing (white-box)	Evaluate the architecture of a system or component; that is, white-box tests involve tracking possible execution routes through the code and determining which input values would force the execution of those routes [9].
Behavioral Testing (black-box)	A test method that examines the functionality of an application without having specific knowledge of the internal functioning of the system, without having access to the source code and without knowing its architecture. The tester could know what is entered and what is the expected result, but not how the results are achieved [10].
Testing related to changes (confirmation and regression testing)	The confirmation tests are to validate that the modifications introduced in the application are present and, that it works correctly, or to confirm corrections of defects introduced in the alterations [8]. The regression tests allow checking the effects of the changes introduced in the functional parts of a system [8]. These defects can be in the software object of the tests, or in any other associated or non-associated component.
Gray-box Testing	It represents the combination of white-box and black box testing, evaluating the functionalities and the correct functioning of a system [11]. That means that a tester has sufficient knowledge about the system and its functionalities and, knows (though not widely) its internal mechanisms (data structure and algorithms used). However, it does not have access to the source code.

Manual and Automated testing

There are types of tests used for test software during its life cycle. Next, they are described.

Manual testing is a technique where the test engineer prepares the test cases manually; that is, without the use of any automated tool or script, in order to detect a defect in the analyzed software.

Manual tests are a laborious activity that requires a certain set of qualities such as: patience, observation, speculation, creativity, innovation, and a lot of skill. These are difficult to apply to large applications with huge data sets.

There are different stages within manual tests: unit tests, integration tests, system tests and user acceptance tests. In addition, this type of tests can include exploratory tests, since the testers explore the software for the identification of errors.

In [12], the main problems that manual tests have are presented:

- ***Time consuming and tedious*** **-** Since test cases are executed by human resources so it is very slow and tedious;
- ***Huge investment in human resources*** **-** As test cases need to be executed manually so more testers are required in manual testing;
- ***Less reliable*** **-** Manual testing is less reliable as tests may not be performed with precision each time because of human errors;
- ***Non-programmable*** **-** No programming can be done to write sophisticated tests which fetch hidden information.

Automation testing is a technique that uses scripts and/or tools in order to test the software entire life cycle, saving time and providing efficiency to the tests. This process involves automating the manual testing process, and allows to re-run the test scenarios that were performed manually quickly and repetitively. The goal of automation is to reduce the number of test cases that will be run manually and not eliminate all manual tests.

The following are the main benefits of automated tests [12]:

- ***Fast*** **-** It is faster than the manual testing;
- ***Cost Effective*** **-** Test cases are executed by using automation tools so less tester is required in automation testing;
- ***Repeatable*** **-** The same test case (record and replay) can be re-executed using testing tools;
- ***Reusable*** **-** Test suits can be re-used on different versions of the software;
- ***Programmable*** **-** Testers can program sophisticated tests that bring hidden information;

- ***Comprehensive*** **-** Testers can build test suites of tests that cover every feature in software application;
- ***More reliable*** **-** Automation tests perform precisely same operation each time they are run;
- ***Test Coverage*** **-** Wider test coverage of application features.

2.2. Agile Testing

In the world of software development, "Agile", is the ability to create and respond to change in order to succeed in an uncertain and turbulent environment [13]. In recent years, agility has taken the world of development and software testing by assault. In fact, the VersionOne's annual State of Agile survey (2017) [14], reported that currently 94% of organizations use agile techniques in some way or another. However, respondents report that this adoption is not always widespread within organizations (60% declare that teams do not practice it), which means that there is still a long way to go in terms of adoption and maturity.

Another highlight is the success rate of agile projects (98%), the respondents said that the main benefits of adopting agile were: accelerated delivery, better visibility of the project, better team productivity and management of the changing priorities. One of the most reasons to make use of agility.

Agile Testing, is a software test practice that follows the principles of agile software development [15]. This practice involves all the members of the project team, especially the testers. Software tests are interlaced with all phases of development, and are performed simultaneously throughout the software life cycle.

In addition, the active contribution of the testers with the rest of the team, allows the construction of quality software; according to the customer's requirements, with better design and coding. It should be noted that "Agile Testing" covers all levels of tests and all kinds of tests.

Agile Manifest

In the 1990´s there were several people who were talking about changing the way to write software. All conversations reached a critical point in 2001, when a number of

software development luminaries, including the likes of Martin Fowler, Kent Beck, Bob Martin, Ken Schwaber, Jeff Sutherland, and Dave Thomas met in a cottage in the Snowbird ski resort in the Wasatch Mountains of Utah [16]. What came out from this meeting became known as the "Agile Manifesto".

Next, the manifest is presented textually [17]:

> *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> *• Individuals and interactions over processes and tools.*
>
> *• Working software over comprehensive documentation.*
>
> *• Customer collaboration over contract negotiation.*
>
> *• Responding to change over following a plan.*
>
> *That is, while there is value in the items on the right, we value the items on the left more.*

In addition to this manifest, twelve principles of agility were created to broaden its declaration. These are the following [21]:

- To satisfy the customer through continuous and early delivery is the highest priority;
- Changes to requirements are welcome, even in late phases of development;
- Deliver software working frequently, from a couple of weeks to a couple of months, preferring shorter periods;
- Developers, managers and clients must work together daily, throughout the project;
- Build projects around motivated people, giving them the environment and support they need, and trusting that they will carry out the work;
- The most efficient and effective method of transmitting information among a team of developers is face-to-face conversation;
- Having software that works is the primary measure of progress;

- The agile process promotes sustainable development. The sponsors, developers and users must be able to maintain a constant work rhythm permanently throughout the project;
- Continuous attention to technical excellence and good design improve agility;
- Simplicity is essential;
- The best architectures, requirements and designs arise from self-organized teams;
- At regular intervals, the team should reflect on how to be more effective, and adjust their behavior according to it.

Agile Development

Agile development is based on an incremental and iterative approach. Instead of in-depth planning at the beginning of the project, agile methodologies are open to changing requirements over time and encourage constant feedback from end users. They are divided into iterations and the objective of each of them is to produce a functional product.

Continuous integration is the key to the success of agile development. The tests become an essential component of all phases of development, ensuring the continuous quality of the product. Communication is of the maximum importance and customer requests are received when it is necessary, giving the customer satisfaction that all requirements are considered and that the quality of the product is available throughout the development.

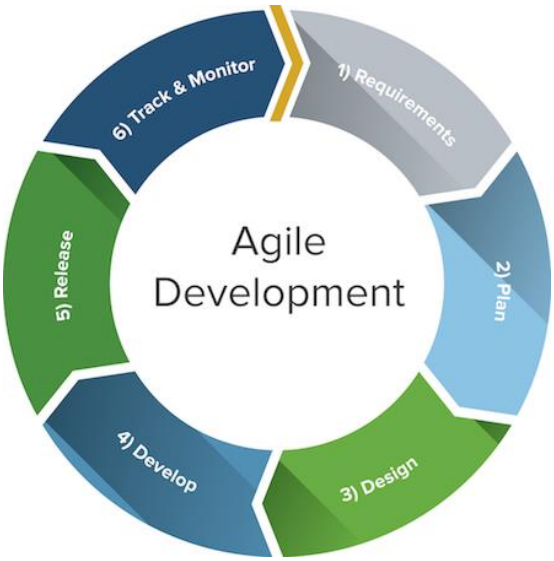


Figure 1. Agile development cycle [18]

The phases shown in the Figure 1, may not happen in succession; they are flexible and evolving constantly and, many of them occur in parallel. [18].

- **Planning -** once an idea is considered viable, the team meets to identify the characteristics, prioritize each characteristic and assign them to an iteration;
- **Requirements analysis -** stakeholders and key users meet to identify business requirements that are quantifiable, relevant and detailed;
- **Design -** it is prepared from the identified requirements and the team considers how the product or solution will look, deciding a test strategy or a plan to proceed;
- **Implementation or coding -** development of features and iterations of programming for implementation;
- **Testing -** Test the code with the requirements to ensure that the product really meets the customer's needs. This phase includes unit tests, integration tests, system tests and acceptance tests;
- **Deployment -** Delivery of the product to customers. Once customers start using the product, they may encounter new problems that the project team should address in future iterations.

Traditional vs. Agile

The approaches Traditional as Agile, have similarities present below [19]:

- Share the same phases. Both worlds achieve the same life cycle activities: planning or analysis of requirements, specifications or design, construction or coding, testing, start-up and maintenance [20];
- Share the same goal. In fact, that both groups have the same goal: to deliver a quality product in a predictable, efficient and responsive manner;
- Use many of the same principles. Traditional and Agile development make use of the same based principles to build and manage software: planning the work, lock down requirements to prevent scope creep, institute multiple reviews, move forward in a step-by-step, and use the necessary documentation to capture all details;

- Use the same basic building blocks. Both approaches also work with the same basic programmatic building blocks: scope, cost, schedule and performance;

The differences between the Traditional versus Agile methodology are presented in TABLE III. They are summarized through the investigation found in [9], [18], [21]–[24].

TABLE III. Traditional vs. Agile

	Traditional approach	**Agile approach**
Planning	Plan according to the full set of requirements.	Iteratively plan whit a smaller set of requirements.
Customer availability	Requires customer involvement only at milestones.	Prefers customer available throughout project.
Process	Sequential. Structured, strict and rigid.	Flexible. Iterative and incremental.
Delivery	One-time at the end of the project.	Continuous delivery throughout the project.
Orientation	Process	People
Scope	Works well when scope is known in advance, or when contract terms limit changes.	Welcomes and accept changes. Works well when scope is not known in advance.
Feature prioritization	"All or nothing approach". Do everything agreed ensure the customer gets everything they asked for. It increases risk of failure.	Prioritization by value ensures the most valuable features are implemented first, thus reducing risks. Funding efficiency maximized.
Team	Team coordination / synchronization is limited to handoff points. More roles.	Prefers smaller dedicated teams with a high degree of coordination and synchronization. Less roles.
Documentation	Each phase of the development process is properly documented and reviewed.	Find the right balance between documentation, development and discussion.
Success criteria	Conformance to requirements.	Business value delivered to the customer.
Quality	Testing happens only after the completion of the development.	Testing team works in parallel with the development team which helps to find defects as soon possible. Has techniques where test are created first

		before code, such as Test-Driven Development or Behavior-Driven Development.
Testing automation	Automation is not usual practice.	Continuous automated testing which ensures better quality.

Agile Methodologies and Frameworks

There are several Agile Methodologies and Frameworks that support Agile Development. They are explained in the TABLE IV:

TABLE IV. List of Agile Methodologies and Frameworks

Methodology / Framework	Description
Extreme Programming (XP)	Is an agile software development framework that aims to produce higher quality software, and higher quality of life for the development team [25].
Feature Driven Development (FDD)	This iterative and incremental process of software development combines the industry's best practices into one approach [18]. FDD has five activities: develop a general model, create a list of characteristics, plan by element, design by element and compile by function.
Adaptive System Development (ASD)	Represents the idea that projects must always be in a state of continuous adaptation [18]. ASD has a cycle of three repetitive series: speculate, collaborate and learn.
Dynamic Systems Development Method (DSDM)	DSDM is one of the leading Agile approaches, bringing together the agility and flexibility necessary for successful organizations today within a framework of the appropriate level of project governance [26].
Lean Software Development (LSD)	LSD takes Lean Manufacturing and Lean IT principles and applies them to software development; it can be characterized by seven principles: eliminate waste, amplify learning, decide as late as possible, deliver as quickly as possible, train the team, build integrity and see the whole [18].
Kanban	The Kanban Method is a means to design, manage, and improve flow systems for knowledge work [27]. Kanban allows organizations to start with their existing workflow and drive evolutionary change.

Scrum	Scrum is a process framework used to manage product development and other knowledge work [28]. It provides a means for teams to establish a hypothesis of how they think something works, try it out, reflect on the experience, and make the appropriate adjustments; that is, when the framework is used properly.

2.3. Test-Driven Development

Test-Driven Development (TDD), created by Kent Beck (inventor of Extreme Programming and JUnit); refers to a style of programming where three activities are intertwine closely: coding, testing (in the form of unit tests) and design (in the form of refactoring) [29]. Its main idea is to perform initial unit tests for the code that must be implemented; that is, first codify the test and, subsequently, develop the business logic [30]. In TDD, tests are written by the developers based on the requirements specified by the clients.

Beck, offers arguments on why it would be beneficial to convert TDD into a tool for software development [31]:

- Increases software quality;
- Highly reusable code is achieved;
- Teamwork is easier, allowing trusting colleagues, even if they have less experience;
- Multiply communication among team members;
- The people in charge of quality assurance acquire a smarter and more interesting role;
- Writing the test before the code, forces to generate the minimum of necessary functionality;
- The review of a project developed through TDD, provides more technical documentation, which will be consulted when it understands what mission each part of the system performs.

It is important to know that, like any technique, it is not a magic wand and does not give the same result to an expert software architect as to an inexperienced junior programmer; however, it is useful for both and for any of the team members [20].

Principles on which TDD is based

Some principles on which TDD is based are the so-called SOLID principles [30]. Next, a brief description is presented:

- **Single Responsibility Principle (SRP**) **-** SRP mentions that each class or module will have a sole responsibility. In practice, most classes should have only one or two methods; perhaps returning to this principle as the most important of all, the simplest and at the same time the most complicated to carry out [20];
- **Open/Closed Principle (OCP**) **-** A software entity (class, module or function) must allow to be extended without needing to be modified. Since the software requires changes and some entities depend on others, modifications in the code of one of them can generate undesirable cascade side effects [20];
- **Liskov Substitution Principle (LSP**) **-** LSP is a collection of guidelines for creating inheritance hierarchies in which a client can reliably use any class or subclass without compromising the expected behavior [32]. Explained in another way, each class that inherits from another can be used as their father without needing to know the differences between them;
- **Interface Segregation Principle (ISP**) **-** When we apply the SRP we also use the ISP. The ISP argues that we should not force classes (or interfaces) to depend on classes or interfaces that do not need to be used [20]. Such imposition occurs when a class or interface has more methods than it needs for itself;
- **Dependency Inversion Principle (DIP) -** Techniques for dealing with collaborations between classes producing a reusable code, sober and ready to change. DIP indicates that a module A should not depend directly on another module B, but on an abstraction of B (interface or class) [20].

Cycle

To apply Test-Driven Development, follow the mantra "RED, GREEN, REFACTOR" presented in the Figure 2(B). TDD flow is presented in Figure 2(A), and consists of the following steps [33]: (1) select a user story, (2) write a test that fulfills a small task of the user story and that produces a failed test, (3) write the production code necessary to implement the feature, (4) execute the pre-existing tests again, where if any test fails, the code is corrected and the test set is re-executed, (5) the production code and the tests are refactored, and, finally (6) start the cycle again.

Normally, TDD flow will run several cycles (three to five) very fast, then you will find yourself slowing down and spending more time on refactoring [3]. Then you will accelerate again, 20-40 cycles in an hour is not unreasonable

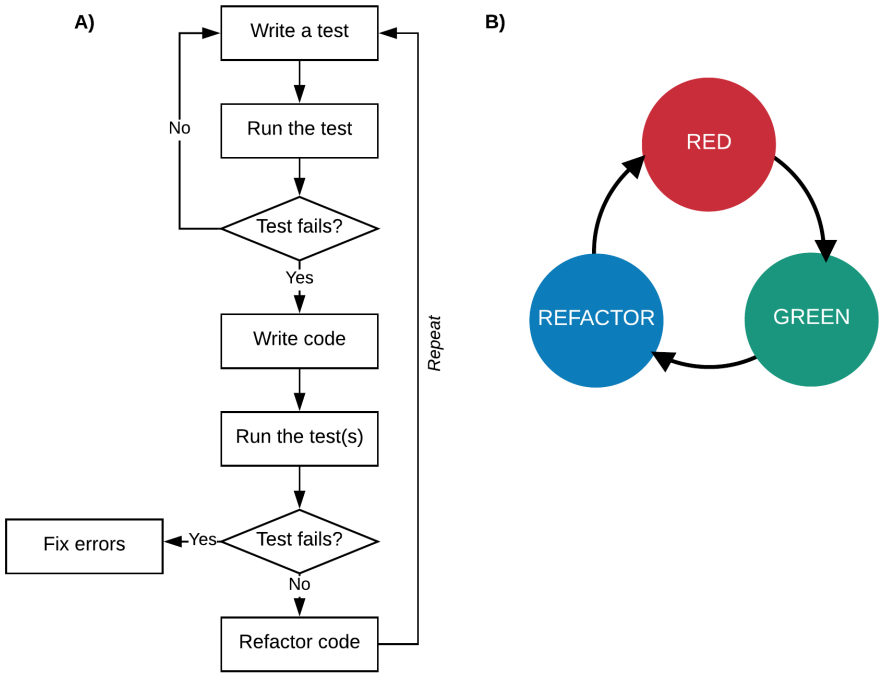


Figure 2. TDD flow and mantra (based on [3], [33]*)*

The TDD steps are easy to learn, but the mentality takes a while to assimilate. Until you do, the technique will seem clumsy, slow and uncomfortable.

Common benefits and mistakes

The expected benefits with the application of TDD are [30]:

- Higher quality in the developed code;
- Design oriented to the needs;

- Simplicity, we focus on the specific requirement;
- Less redundancy;
- Higher productivity (shorter debugging time);
- The number of errors is reduced.

Typical individual mistakes include [29]:

- Forgetting to run tests frequently;
- Writing too many tests at once;
- Writing tests that are too large or coarse-grained;
- Writing overly trivial tests, for instance omitting assertions;
- Writing tests for trivial code, for instance accessors.

Typical team pitfalls are [29]:

- Partial adoption: only a few developers on the team use TDD;
- Poor maintenance of the test suite: most commonly leading to a test suite with a prohibitively long running time;
- Abandoned test suite: sometimes as a result of poor maintenance, sometimes as a result of team turnover.

Main Tools

The main tools used to apply this technique are summarized below:

- **JUnit -** It is an open source framework designed for the purpose of writing and running repeatable tests in Java [34];
- **XUnit.Net -** It is a free, open source, community-focused unit testing tool for the .NET Framework [35];
- **VBUnit -** It is a tool for creating unit tests for Visual Basic and COM objects [36];
- **CUnit -** It is a light system for the writing, administration and execution of tests in C [37];
- **HtmlUnit -** It is "GUI-Less browser for Java programs" [38]. It has good JavaScript support and is able to work with AJAX libraries, simulating a browser like Chrome or Firefox.

As you can see, all the tools are part of the "xUnit" family, thus being the main purpose of generating unit tests; to specify, design and verify the code of an application or system.

2.4. Acceptance Test-Driven Development

Acceptance Test-Driven Development (ATDD), also known as Story Test-Driven Development (STDD), is a technique similar to TDD, but at a different level. This technique involves team members with different perspectives (clients, development, tests) who collaborate to write acceptance tests before the implementation of a functionality [39].

Collaborative discussions to generate acceptance tests are often referred to as the "Three Amigos" [40], and represent the three perspectives of the technique: client or business (what problem are we trying to solve?), Development (how can we solve that problem?), and tests (what about this, what could happen?).

Acceptance tests are specifications for the desired behavior and functionality in application or system; they tell us, for a given user story, that the system handles certain conditions and contributions, and with what kind of results [41].

Acceptance tests can be generally short and somewhat informal; but they have general properties to highlight [41]:

- **Owned by the customer -** The tests are the property of the client whose main objective is to specify acceptance criteria for user stories. When the client writes the acceptance tests, it helps prevent to developers write them and it makes them contain the correct definition of the functionality;
- **Written together with the customer, developer, and tester -** As the client is the owner of the acceptance tests, so to speak, he should not be the only one to write them, especially when they are initiators in the matter. The importance of providing help and support allows increasing inclusion and communication in the process of creating stories as evidence. With this we will have the client in his role of expert in the domain, the developer in the role of a technical expert and the evaluator in a role that combines a bit of both; that is, everything covered;

- **Focus on the what, not the how -** It is one of the important characteristics that makes user stories appropriate and focus on describing the value that the client provides. User stories strive to convey needs and wants, what and why, and give little attention to implementation. This is how it helps to change the defective perception of showing what we can offer, instead of what the client wants;
- **Expressed in the language of the problem domain -** It is a fundamental requirement for the client to participate in the development of acceptance tests using the domain language. By using this language, you disperse too much technical jargon in the tests and you become less vulnerable to having a demanding error in a production release. In addition, when domain language is used, the changes that need to be made in tests during refactoring are typically nonexistent or, at best, trivial;
- **Concise, precise, and unambiguous -** If the language of the domain is used, the tests are simple and consistent. Each acceptance test verifies a single aspect or scenario relevant to a story of the user in question. Keeping the evidence clear, easy to understand and easy to translate to executable tests (ambiguity), will allow to avoid errors and work better with them.

ATDD is a way of facing implementation in a totally different way to traditional methodologies, where the work of the business analyst is transformed to replace texts of requirements written in natural language (our language), by executable examples arising from the consensus among the different team members and the client [20].

Cycle

The algorithm is similar as the presented in TDD, but with greater stride. In ATDD the list of tests of each story is written within a meeting that includes product owners (clients), developers and quality managers (testers). The whole team must understand what is to be done and why, to be specified in such a way that certifies that the software will do what it should do. As there is no way to decide the acceptance criteria, the different members of the team support each other to create them.

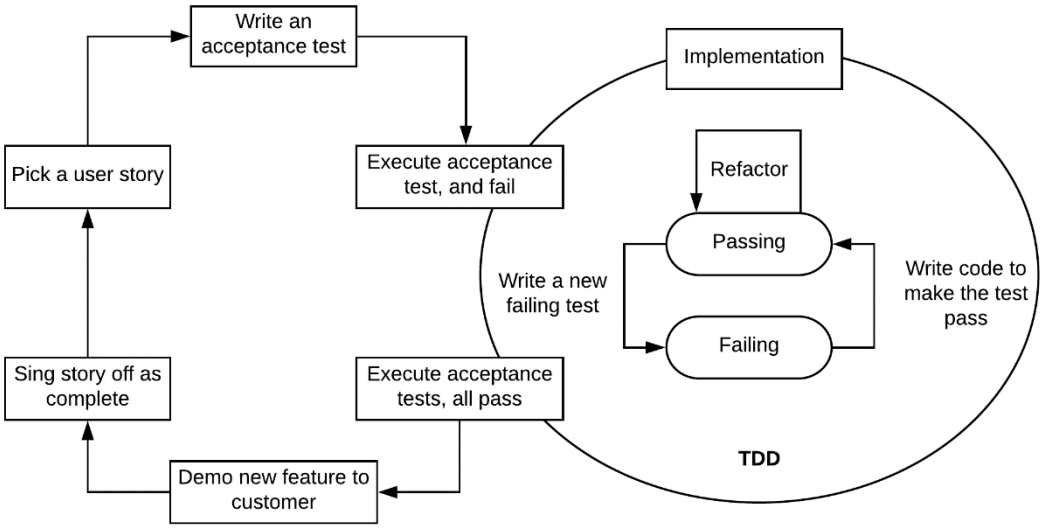


Figure 3. ATDD flow (based on [41], [42]*)*

As shows Figure 3, the ATDD flow highlights 3 important activities, which are [41]:

- **Pick a user story -** The first step is deciding which story to work with, this process is not simple; but, fortunately, most times there will already be some relative priorities for all the stories in the workload of our iteration. The simplest way to do this is always to select the story considered the most important and it is usually found at the top of the stack;
- **Write acceptance tests -** The tester worked with the client to find specifications that accurately describe the expected behavior corresponding to a requirement. All variations of behavior are specified through concrete examples with clearly defined inputs and outputs, and are added to acceptance criteria. The developer should build the test and examples based on the acceptance criteria; and automate them if is necessary;
- **Implementation -** The developer designs and implements the functionality that makes the system comply with the requirements and, it is done when the system approves the acceptance test. The precise processes used to implement the functionality are beyond the scope of ATDD.

Benefits

In [43], some benefits are reported when using ATDD:

- **Offers more focus on the client's needs -** The acceptance test criteria are the basis for development, making the client's objectives persist focused on the

construction of the software. ATDD forces developers to think about the customer's perspective and focus on their needs;

- **Improves collaboration among stakeholders** - Product owners, business analysts, developers and testers work since the moment that user's history is written and continues until the code developed meets the acceptance criteria;
- **Allows the quick resolution of problems -** In ATDD, the tests are not an isolated activity before the deployment of the application, helping the early identification of errors and accelerating the resolution of problems;
- **Provides easy administration -** ATDD occurs in several small iterations, which means that the equipment needed for the projects is small. This facilitates communication and collaboration (sharing knowledge and skills) among team members, such as managing resources and infrastructure compared to large development teams.

Risks

ATDD is considered an improvement of TDD, and as such it is assumed that certain TDD failures are avoided. Despite it, when ATDD is implemented in a project, there are certain factors that can become problems. Based on the Ron Quartel reference [44], the crucial factors are presented:

- **Human factor -** Keep in mind that all people who are part of the project are involved fully, especially Product Owners. Product Owners are people at the business level and they need to have an understanding and fluent communication with the team members that build the product. In addition to them, the DEV and QA must be involved; otherwise, ATDD will not be effective;
- **Tools -** Many times, there is a lot of emphasis on the tool that will be used or how it can be integrated with others, losing focus on the criteria of "what we are trying to solve" and "how we are going to build it";
- **Regarding the tests -** The tests are one of the pillars in which the practice is sustained and, we have to keep in mind that they are of great help and should not turn against the development team. It must be avoided at all costs a difficult to modify beginning and slow in execution; for which early and continuous

refactoring is suggested. In addition, refactoring should eliminate those tests that no longer apply to the case or are no longer necessary.

Main Tools

As mentioned previously, the automated acceptance tests are created within ATDD. In [20], Carlos Blé talks about the use of some tools for the implementation of the technique:

- **Selenium -** It is a set of different software tools each with a different approach to supporting test automation [45];
- **Fitnesse -** This tool allows us to specify and verify the acceptance criteria of an application, generating a wiki server, which serves as a source of documentation [46];
- **Concordion** - It is an open source framework for Java, which allows to automate specifications based on examples (SbE), another name of ATDD [47];
- **Robot Framework** - Is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD), it has easy-to-use tabular test data syntax and it utilizes the keyword-driven testing approach [48].

2.5. Behavior-Driven Development

Behavior-Driven Development (BDD), is a synthesis and refinement of software engineering practices; that help teams generate and deliver higher quality software quickly [49], [50]. Its base represents some agile practices and techniques, including, in particular: Test-Driven Development (TDD), Acceptance Test-Driven Development (ATDD), and Domain Driven Design (DDD).

BDD augments these techniques with the following tactics [50]:

- Using the "Five Why's" principle in each proposed user story, so that its objective is clear for commercial results;
- Implement only those functionalities that directly contribute to commercial expectations;

- Describe behaviors in a single notation, being accessible to all the work team (domain experts, testers and developers), to improve communication;
- Put into practice all the techniques that are part of BDD, up to the lowest levels of software abstraction and paying attention to the distribution of behavior.

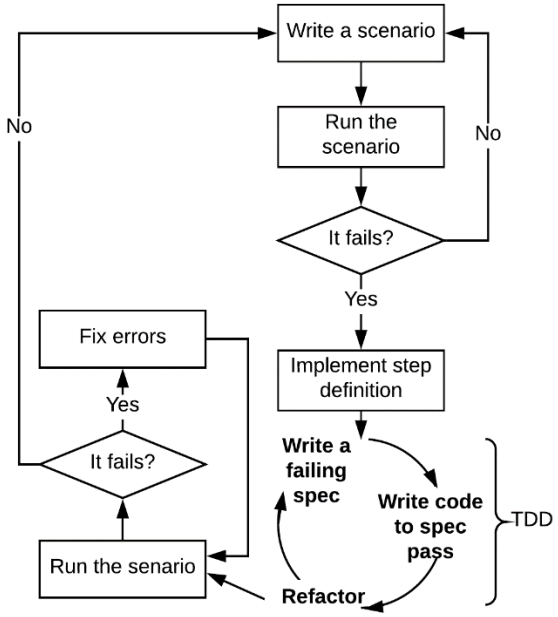


Figure 4. BDD flow

In practice [51], BDD is very similar to TDD (see Figure 4), since the main stakeholders have knowledge and programming skills. However, in many organizations, the technique is used to extend the contribution in the development of the solution, including the commercial part and end user who may have little knowledge of software development. Due to this more extended feedback cycle, BDD can be used in integration and continuous delivery environments easily.

In addition, tests are written by the end user or the owner of the product or the analyst, together with the testers; and developers only generate the code needed to pass these tests. This process should be:

1. The business person specifies behaviors they want to see in the system. The developer asks questions based on their understanding of the system, while also writing down additional behaviors needed from a development perspective. Ideally, both parties can refer to the list of current system behaviors to see if this new feature will break existing features.

2. Follow the next steps: (1) select a specification, (2) write the test referred to the specification, (3) write the simplest code to pass the test, (4) pass the specification, and finally, (5) refactoring to eliminate duplication.

Gherkin

Within BDD, the user stories possess the acceptance criteria, from which acceptance tests are derived, and they can be written directly in a specific domain language called "Gherkin". This language is very similar to natural language, so experts in the domain can read and understand its writing [52].

Gherkin is a very important advantage, since BDD not only proposes to automate the tests, but also to improve communication between domain experts and software developers.

```
Feature: Customer displays order

  Part of the "Making an Order" epic

  As a Customer
  I want to display the order
  in order to review the contents of my order and its price easily

  Scenario: Order is empty
    Given that the order is empty
    When the customer displays the order
    Then no order items will be shown
    And "0" will be shown as total price
    And there will only be possible to add a beverage
```

Figure 5. Gherkin sintax [52]

The Figure 5, shows a file with the syntax of this domain language; and it has the following specifications:

- **Feature** - represents the name of the functionality that we are going to test. It must be a clear and explicit title. Here is a description in the form of user history: "As a [role], I want [characteristic], in order to [the benefits]";
- **Scenario** - describes the specific contexts and results of a user story. It includes the necessary steps to put the system in the state that you want to test. There will be a scenario for each test where that functionality will be specified;
- **Given** - marks the context; that is, the preconditions for the scenario;

- **When** - specifies the actions that will be execute when the test is launch;
- **Then** - details the expected result, the validations to be made.

In addition, there is the possibility to create a ***feature*** for each file and, within it create different test scenarios.

Benefits

When BDD is implemented in a software development project, some benefits emerge that will be mentioned below [49]:

- Reduction of monetary costs, thanks to the fact that there is a greater concentration on what the business needs and expects from the product;
- Reduction of costs referring to errors and defects produced in each iteration of implementation, since all team members know from the beginning of the project what the final behavior of the system is;
- The changes are easier and safer, because there is a constant feedback that allow tracking the status of the project;
- The requirements are addressed and developed quickly, this is mainly due to the use of automated tests and the fact of knowing and understanding what the system is expected to do.

Main Tools

There are also various toolkits supporting BDD, such as: JBehave, Cucumber Behat, Specflow and RSpec. To follow, a brief definition of each one is presented.

- **Cucumber** - Open source tool for executable specifications, designed to ensure acceptance tests are easy to read by any member of the team [53]. It provides support for multiple programming languages such as Ruby, Java, .Net, Php, Javascript, among others;
- **SpecFlow** - Open Source BDD framework for .Net, which like Cucumber uses the Gherkin language [54]. It has support for popular testing frameworks such as MSTest, NUnit (2 and 3), xUnit 2 and MbUnit;

- **Behat -** Open source framework that makes possible Behavior-Driven Development (BDD), in software projects for PHP [55];
- **JBehave -** BDD framework for the Java language, which was created by Dan North; it also provides plugins for integration with Eclipse, as well as Ant and Maven [56];
- **RSpec -** BDD framework for the Ruby language, inspired by JBehave, which is composed of multiple libraries designed to work together or independently with other tools such as Cucumber [57].

2.6. When could be apply the techniques?

Test-Driven Development, is the base for the creation of the other techniques mentioned above (ATDD and BDD); for which the factors that limit its implementation are detailed[58]:

- Few expertise in software development. Despite receiving training, the lack of professional experience shows a relevant factor in the adoption of new knowledge. It is very important to consider a correct training on TDD, so that the programmers achieve the necessary skills to put into practice it;
- Insufficient testing skills. To start up Test-Driven Development, first of all, it´s necessary to have the proper control over testing, since to create cases of quality tests it is required effort and dexterity;
- Lack of interest on the part of the developers. Sometimes there may be few adhesions to the use of TDD due to factors such as: reduced development times, few disciplines or not noticing its benefits in the short term. This may be because the technique is seen as slow at the beginning or because developers find it tedious to be restricted to following a structured process for creating software;
- Increase of development time. The most of studies show an increase in the time needed to implement a set of given requirements; due to this, the maturity of the teams is fundamental to the adoption of new practices and / or techniques within an organization;

- Limitations in the domain of tools. It is vital that when implementing this technique, you have the appropriate tools for the development of it and, in addition, take into account the time necessary for the training of the team in the use of them;
- Legacy code. Test-Driven Development in its original form does not express anything about how to work with legacy code, moreover, in many cases it presupposes that all the code is done from zero. This is strange in the most organizations that want to adopt TDD and it can be a problem when starting the technique; Well, it could find problems about how the new changes will affect the old ones, causing concerns between the developers.

These factors must be considered for the implementation of any of the techniques, and if within the organization they are easy to overcome, it is possible to apply them in most cases.

In addition, if it´s necessary a quick feedback of the code, it can be used TDD; if it´s necessary to know that it is building the right thing or if the different pieces of code developed are integrated as expected, it can be applied BDD or ATDD. Also, the creation of acceptance criteria in ATDD can be reused in the BDD scenarios.

2.7. TDD vs ATDD vs BDD

The comparison of the techniques, was made through the investigation found in books [20], [31], [41], [49], [52], [59] and blogs/sites [60]–[68]. It's important to mention that the terms ATDD and BDD generate a lot of confusion; because they were techniques proposed by different people in similar years, and some authors mention that they are the same while others don't. In my case, I think that ATDD and BDD are closely related, and there are these distinct terms exist to stress some differences in approach that lead to similar outcomes. Also, Wes Williams, author of the GivWenZen framework, mentions about the application of these techniques [68]: *"Having worked on a project that was using 'ATDD', in 2005 I think, we had the same goals then as BDD without the Given-When-Then language"*. So, the analyzing is presented in TABLE V.

TABLE V. TDD, ATDD and BDD comparison.

	TDD	**ATDD**	**BDD**
Objective	Focused on testing,	Focused on capturing the	Focused on the

	helps software developers to produce better quality and maintenance code.	requirements within the acceptance tests and uses them to drive the development. It brings the customer with a quick feedback of the progress of the development.	behavioral aspect of the system (why should the code be written? and how should it behave?). It unites the distances between the client and the developer.
Users involved and scope	Allows the approach between the developer and the tester to create well written unit of code (module, class, function).	Communication and shared vision between business users, developers, and testers to ensure requirements are well constructed and documented.	Combination of TDD and ATDD. This allows much easier collaboration with non-techie stakeholders.
Steps	Step 1: Select a user story; Step 2: Write a test that fulfills a small task of the user story and that produces a failed test; Step 3: write the production code necessary to implement the feature test; Step 4: Refactor the code to eliminate duplication. (cycle that repeats until the functionality is ready)	Step 1: Select a user story; Step 2: Write Acceptance Test based on acceptance test criteria; Step 3: Write the simplest code to pass the test; Step 4: Refactor the code to eliminate duplication. (cycle that repeats until the functionality is ready)	Step 1: Select a specification; Step 2: Write the test referred to the specification; Step 3: Write the simplest code to pass the test; Step 4: Pass the specification; Step 5: Refactor the code to eliminate duplication. (cycle that repeats until all functionalities are ready)
Input documentation	User requirements and documentation of requirements analysis are the base for development and	The Acceptance Criteria added the Scenarios, generate Acceptance Tests.	Specification document in native language with given, when, then. Also, the acceptance criteria

	testing.		specified.
Test automation	It's required.	Not required but needed for regression purposes.	It's required.
Test mapping	Each functionality should have a test implementation.	Each story should have an acceptance test.	Each story should have a behavior test.
End users focus	TDD tests are technical and should be understood by developers and testers.	ATDD tests are readable and focused for customers.	BDD tests are understandable for IT team and customers.

Tools analysis

TABLE VI collects the main tools used for TDD, ATDD and BDD techniques. The official documentation of each of them provided the necessary information for their analysis.

TABLE VI. TDD, ATDD and BDD tools analysis

	TDD	**ATDD**	**BDD**
xUnit frameworks	X	X	X
Mocking tools	X		
Selenium		X	X
Fitnesse		X	
Concordion		X	X
Robot Framework		X	
Cucumber		X	X
SpecFlow			X
Behat			X
JBehave			X
Rspec			X
Jasmine			X

TDD uses unit tests, for the creation of this type of tests there are xUnit frameworks available for many programming languages and development platforms. Also, these frameworks can be applied for the construction of acceptance tests in the ATDD and BDD techniques. In TDD, is important that things are tested in isolation. To achieve

this, we need to ensure that dependencies to external code are isolated using mocking tools.

ATDD requires the definition of functional level acceptance tests before writing the code; Fitnesse, Concord and Robot Framework are tools for acceptance testing and permit to apply this technique.

In BDD, the behaviors represent both the specification and the test cases. The language used to write the behaviors of systems are Gherkin, and tools like SpecFlow, Behat, JBehave, Jasmine, Rspec, Cucumber and Concordion perfectly understand it.

Some tools can be use in ATDD or BDD cases. Sometimes, if the user will be actively involved writing/validating the acceptance tests, they should be constructed with natural language; tools like Cucumber and Concordion support it. In addition, Selenium can be used in both cases, if the acceptance criteria or the behavioral aspects involve the UI.

3. State of Art

The present section contains the collection of related and relevant works for this research.

3.1. Systematic review methodology

A systematic review represents the greatest effort to collect and synthesize scientific evidence on a specific topic, through a method that ensures the biases and limitations are the minimum possible. To achieve a systematic review, we will use the 5-step methodology explained by Khalid S. Khan et al. in [69].

The 5 steps are summarized below:

- ***Step 1: Formulation of the question*** **-** The problems addressed in the review should be specified in the form of clear, unambiguous and structured questions before beginning the revision work;
- ***Step 2: Identification of the relevant work*** **-** An extensive search of the studies should be applied, selecting multiple resources without language restrictions. These studies must meet certain minimum requirements (selection criteria). The selection criteria (inclusion and exclusion) are based on the asked question;
- ***Step 3: Evaluate the quality of the studies*** **-** Having the inclusion and exclusion criteria, the review now focuses on the studies that have the most reliable results. This should be tested by reviewing the validity of each study individually, using some method of critical analysis. These detailed quality assessments will be used to explore heterogeneity and inform decisions about the adequacy of the meta-analysis (Step 4);
- ***Step 4: Summing up the evidence*** **-** The synthesis of data consists of the tabulation of the characteristics of the study, the quality and the effects, as well as the use of statistical methods to explore the differences between the studies and combine their effects (meta-analysis);
- ***Step 5 Interpreting the findings*** **-** The review ends with the elaboration of conclusions and for this the quantity and quality of the work extracted from the literature are taken into consideration. The risk of publication bias and related

biases should be explored. The exploration of heterogeneity should help determine if the overall summary can be relied upon.

3.2. Formulation of the question and identification of relevant work

First, we considered the following keywords: TDD, BDD, ATDD, software development techniques; which will be used to find relevant information within the academic search engines: Google Scholar, Elsevier and Springer Link.

For the second step, the selection criteria to apply in order to filter information will be;

- *Studies in the period 2012 – 2017;*
- *Studies in English, Spanish, or Portuguese;*
- *Studies that contain the keywords in the title;*
- *Studies that contain information of an analysis, comparison, or application about the techniques.*

The use of the inclusion/exclusion criteria mentioned are represented within Figure 6, allowing the most reliable information to be selected.

Finally, the in-depth evaluation of the risk of several biases within our studies (Step 3) are represented in TABLE VII, in which a hierarchy of quality is developed, according to the degree to which the studies meet the criteria.

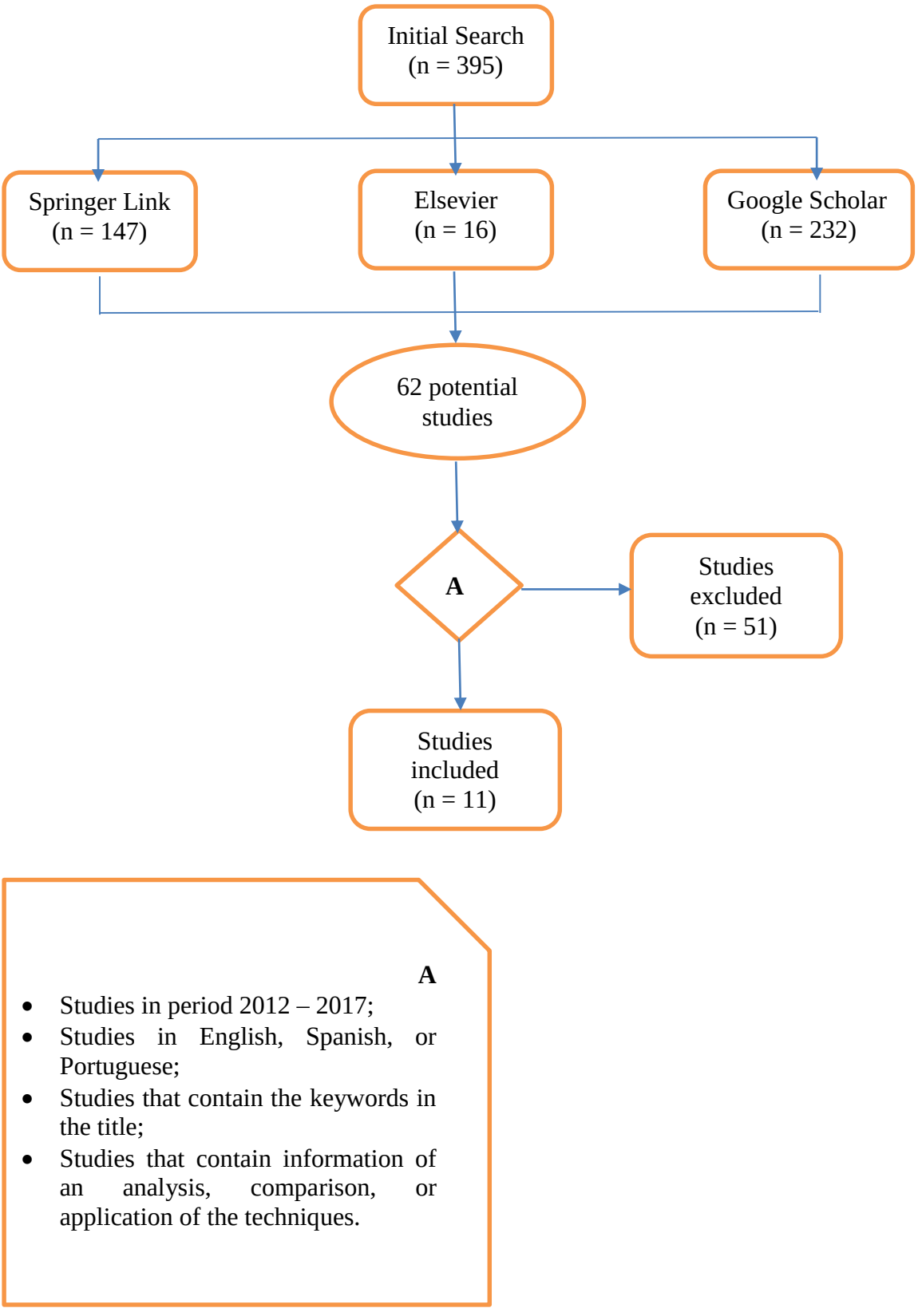


Figure 6. Systematic Literature Review: Step 2 - Identification of the relevant work

TABLE VII. List of the selected scientific articles

Title	Authors	Data base	Year	Number of Citations
Test-Driven Development: the state of the practice	Hammond, Susan, and David Umphress.	Google Scholar	2012	22
A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?	D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo	Google Scholar	2016	20
A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development	Rahman, Mazedur, and Jerry Gao	Google Scholar	2015	12
An experimental evaluation of test driven development vs. test-last development with industry professionals	Hussan Munir, Krzysztof Wnuk, Kai Petersen and Misagh Moayyed	Google Scholar	2014	7
A Study of Test Techniques for Integration with Domain Driven Design	Santos, Eloisa Cristina Silva, Delano Medeiros Beder, and Rosângela A. Dellosso Penteado.	Google Scholar	2015	3
Applying Acceptance Test-Driven Development to a Problem Based Learning Academic Real-Time System	Luiz Felipe Simoes Hoffmann; Luiz Eduardo Guarino de Vasconcelos; Etiene Lamas; Adilson Marques da Cunha; Luiz Alberto Vieira Dias	Google Scholar	2014	3
Agile Test Automation: Transition Challenges and Ways to Overcome Them	Venkat Moncompu	Elsevier	2013	1
Implementing Behavior-Driven Development in an Open Source ERP	Rogerio Atem de Carvalho, Fernando Luiz de Carvalho e Silva, Rodrigo Soares Manhães, Gabriel Lima de Oliveira	Springer Link	2013	1
Efectividad del Test-Driven Development: Un Experimento Replicado	O Dieste, ER Fonseca, G Raura, P Rodríguez	Springer Link	2015	1
A Study of Tools for Behavior-Driven Development	Okolnychyi, Anton, and Konrad Fögen	Google Scholar	2016	0
Test-Driven Development - Beneficios y Desafíos para el Desarrollo de Software.	Vaca Pablo Andrés, Maldonado Calixto, Inchaurrondo Claudia, Peretti Juan, Romero Soledad, Bueno Matías, Cagliolo Marcelo	Google Scholar	2014	0

3.3. Evaluation, Synthesis and Interpretation of the findings

The selection of the scientific articles was made using the methodology explained in section 3.1, considering the criteria specified in the previous section. Then, each article was classified considering if it belonged to a study about the techniques, or to the application of them in the development of a project. In order to understand the context of the investigations, each article was carefully read; allowing identifying its main argument, as the fundamental concepts and ideas proposed. Finally, summaries were made that provide brief and condensed descriptions of each investigation.

The tests are a fundamental part in the development of any software project; there is an extensive literature on tests, including their relationship with agile development. Agile methods and their associated practices, for example: Test-Driven Development (TDD), Acceptance Test-Driven Development (ATDD) or Behavior-Driven Development (BDD), are widely used in the industry and they have been subject repeatedly to empirical and practical studies.

The application of agile development has increased in recent times; despite it, challenges persist in the implementation of testing methodologies and difficulties in their adoption, limiting the desired commercial results. V. Moncompu [70] discusses these challenges in order to understand the good practices that can be applied within organizations. The common challenges within the agile transition are: the integration of the client throughout the life cycle of the software, the difficulty in automating traditional tests, the incorrect estimation of tasks, and the loss of focus in the execution of proposed tasks in a sprint due to the emergence of new tasks and sometimes "off-topic" tasks. Countering the above, it is stated that: the agile development provides integration practices which facilitate the communication with the client; work groups receives professional guidance to acquire the necessary new knowledge and, uses them effectively, estimating in the most real way possible, the efforts carried out by each task without worrying about your margin of error; and promote the use of Kanban to limit the work in progress and lead to the completion of the stories started.

The limited scientific knowledge about TDD has promoted that some researchers carry out experimental studies or systematic reviews about it. In respect to the

aforementioned, in the work of O. Dieste [33], a TDD analysis is conducted, to assimilate the effect that this technique produces in the experience of the developers (code quality or productivity). Based on an experiment conducted by the Polytechnic University of Madrid (2014) [33], where TDD is compared with ITL (Iterative Test-Last); a replica of the experiment is executed, as similar as possible, with 17 students that are a part of a software engineering master´s degree. The experiment had phases of training (students learn about the techniques) and implementation (students solve code katas), where it was observed that the effectiveness produced by TDD was lower than ITL. The differences were not significant because both productivity as well as quality improved in half of the cases. This lead to the conclusion that TDD does not produces benefits immediately and it is of the utmost importance that the subjects exposed to the technique receive an intensive instruction so that the effects are evident in the future.

Incremental Test-Last *consists of the development of small portions of the production code followed immediately by the performance of unit tests* [71]. *The ITL flow is presented in Figure 7*

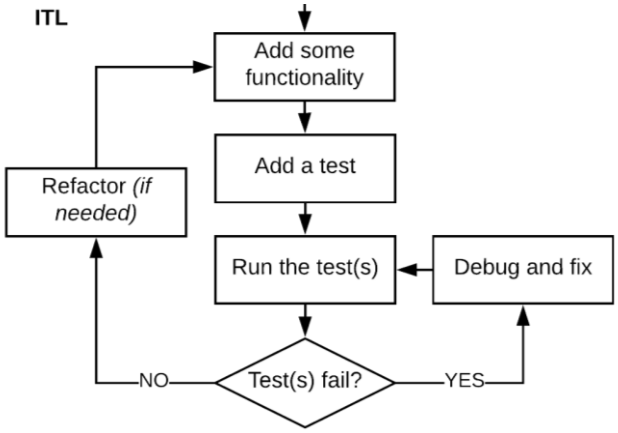


Figure 7. Incremental Test-Last flow.

A ***code kata*** *is an exercise in programming which helps a programmer hone their skills through practice and repetition* [72]. *Each kata is a short exercise (perhaps 30 minutes to an hour long), and can be coded in many different ways*

The research directed by Munir et al. [73] was developed in the industry with professional Java developers, with previous knowledge of software testing. It aimed to visualize the impact produced by TDD on the quality of internal code, the quality of external code and productivity, when compared to TLD (Test-Last Development). For

this purpose, a programming exercise consisting of 7 user stories was executed, allowing the participants to put into practice the aforementioned techniques. The results of the analysis by the number of approved test cases, McCabe's Cyclomatic complexity [74], branch coverage, number of lines of code per person per hour, and number of user stories implemented per person per hour; showed slightly significant improvements in favor of TDD, especially in reducing the number of defects. In terms of productivity, it suggests that subjects who used TDD achieved a productivity average slightly lower than TLD. This indicates that the adoption of TDD requires compliance with the guidelines of all aspects of software development, and adequate training to improve the skill set of the tests.

There is also a recent study designed by Fucci et al. [75], where TDD is compared to ITL through a controlled experiment with professionals within software companies (two in Europe and one in Asia). To achieve a more exact qualification of the effect produced by the techniques within quality and productivity, four characteristics are formulated: sequencing, granularity, uniformity, and refactoring effort. The resolution of programming exercises of different levels of difficulty, revealed that the improvements found in TDD were associated with granularity and uniformity. The remaining characteristics did not have a relevant influence within the experiment. Thus, the benefits of TDD are due to the fact of encouraging stable and precise steps that improve the focus and flow of software development, which in turn promises to improve quality and productivity.

Some cases of business software projects, where TDD is implemented, were analyzed with the intention of discovering the impact derived from the implementation of the technique [76]. It was observed that TDD is applied to all types of projects, both small and large; since the systems are divided into modules, where the methodology is applied, without any issues. Relevant positive aspects were found: developers acquire habits that make them write better code, increasing the reliability of the code allows generating higher quality applications. There is increased productivity and improvements in the communication between customers and developers. On the other hand, there were challenges regarding the previous experience of the developers, since TDD is simpler to implement in teams with medium to high experience. Therefore, the implementation of TDD requires experience in development, commitment, discipline and perseverance to achieve the best results.

S. Hammond and D. Umphress [77], provide a review of Test-Driven Development, in order to know the acceptance acquired in recent times and the efficiency provided in its application. The explored studies refer aspects such as external quality, code quality and productivity. Also, they refer that the extensions of the technique like: Acceptance Test-Driven Development, Behavior-Driven Development, and Agile Specification-Driven Development; bring new features into the software development that improve it. Finally, they mention that TDD is currently used but causes a slight confusion in the programmers at the moment of knowing if the development of each one of the features covered in the tests is correct. For this purpose, they propose the joint use with the Means-Ends Analysis Approach (MEAA) [77]; because it allows developers to execute functional test cases individually and incrementally, validating that each functionality has been successfully executed.

One of the techniques that has obtained a lot of relevance from the point of view of research and practice in recent years, is Behavior-Driven Development (BDD). A study presented in [78], in order to know the benefits BDD offers, analyzes tools that allow the execution of BDD for programming languages based on JVM (Java, Groovy or Scala), such as: Concordion, Spock, Cucumber, JBehave and easyb. The analysis shows that Spock and easyb focus on the level of unit tests, while JBehave, Concordion and Cucumber are more suitable for acceptance / integration tests. Only Concordion admits to some extent the creation of a specific omnipresent language for projects. JBehave, and Cucumber support predefined ubiquitous languages, while Spock and easyb have some important restrictions in this regard. All the frameworks analyzed support automated acceptance tests; however, Concordion, JBehave and Cucumber have more ways to define the scenarios. With the exception of easyb, the tools also provide the ability to use Gherkin. All the analyzed tools are suitable for BDD, but they differ in the definition of test scenarios; therefore, one or several must be select according to the needs presented by the project. In addition, there are specific independent tools, such as Mockito and EasyMock, that can be integrated into all analyzed frameworks allowing the creation of test mocks in automated unit tests.

After mentioning some analysis and systematic reviews, it is time to introduce studies that make use of the techniques within the business or experimental field.

With the aim of solving the typical problem of ensuring that a correct implementation of the requirements is made through the source code, R. Carvalho et al. proposes to

develop a module within an existing ERP system and integrate its development through BDD [79]. For this, ERP5, an Open Source ERP software, was selected because it works under a workflow engine that was taken as business processes and allowed to apply the technique in a simple way. The construction of a module within the system did not presented difficulties, since it was possible to relate the automated tests to existing flows using the Give-When-Then (GWT) constructors. The business process was separated, the GWT scenarios were defined, the code of each scenario was written within the tests, and the tests were executed; verifying that all of them pass satisfactorily. In this way, it was demonstrated that the BDD technique can be integrated into business applications, and unite the advantages of using BDD to track that all requirements were created correctly.

Now focusing on the financial field, where there are large annual losses due to unauthorized access and fraud in electronic commerce transactions; it is proposed to combine Problem Based Learning (PBL) and Acceptance Test-Driven Development (ATDD) to generate preventive solutions to cyber-attacks [80]. With the use of SCRUM and within a university work team of the Technological Institute of Aeronautics (Brazil), the development of a prototype of Real Time Integrated System for the Control of Unauthorized Access and Fraud Detection (SETRAIF) was executed. This system has four essential devices: the mobile transaction device (DMT) that provides reliable electronic transactions through mobile devices, a communication device in the cloud (DCN) that allows the secure exchange (encryption) of information between the DMT and the DCA; the access control device (DCA) uses information from the vendor and customer devices to validate or block access, and the fraud control device (DCF) that verifies the legitimacy of the transaction based on behavior and history geographical area of the client in order to reduce the risks of fraud for those involved. PBL intervened in the learning and communication within each team. The identification of the test strategy and, consequently, the test cases for the SETRAIF prototype were based on the agile test quadrants using ATDD. This interdisciplinary approach facilitates the exchange of knowledge and the specification of requirements, leading to the construction of quality software and improving communication among those involved.

Cloud Computing and Mobile Computing, due to their unique needs, have transformed the way applications are developed, orienting them to the micro services paradigm.

This creates uncertainty when reusing and maintaining each component; for which M. Rahman and J. Gao [81] propose an architecture of automated acceptance tests based on characteristics / scenarios of BDD to address all these concerns and maintain a continuous development. For the construction of the architecture, the following needs were considered: reuse of step implementations for test scenarios, differentiation of activities between developers and testers, ease of auditability, maintainability with respect to versions of the BDD framework (Cucumber, Behat, and Behave), and the complexity of integrating components of a micro service during its execution time. Thus, the architecture was developed within a single repository called ATT repository, instead of maintaining automated acceptance tests in multiple repositories of an application. This main repository is divided into 3 main levels: "Features" - that contains a subdirectory for each micro service with the BDD files necessary for its implementation; "Step-Implementation" - has implementations for specific steps and through one or several BDD micro services registered in a subdirectory within the characteristics directory; and "Library" – that contains the source code for the selected BDD framework. The construction of the architecture was successful, noting that one of the most efficient means to create robust software is the automated acceptance tests, and recommending the application of BDD in any situation.

An interesting case is presented in [82], where it intends to integrate TDD and BDD with Domain Driven Design (DDD), allowing enhancing the strengths and weaknesses of these techniques. For integration, the Apache Isis framework is used, which allows the development of Web applications or RESTful APIs in the JAVA programming language implementing DDD concepts. A test scenario based on a video rental system was carried out, where the implementation of tests with TDD used JUnit incorporated in the framework. These were created following the phases: red, green and refactoring, according to the literature. For the creation of tests following the BDD technique, Apache Isis incorporates Cucumber; allowing the creation of test scenarios in narrative language and the execution of these scenarios within classes. With this work, we can see that it is possible to integrate these techniques, promoting the increase of software quality and understanding among interested parties.

3.4. Main and common points

Based on the literary review and recent studies, some suggestions should be considered when applying these methodologies: Train all the team members continuously, incorporate the client throughout the life cycle of the software adequately, achieve a correct estimation of the work and, selecting the best tools to build and automate tests according to the development environment; especially if they are integrated with previous created software.

Considering the creation or replication of experiments and the experience of O. Dieste explained during the meeting via Skype made in February 2018; the following considerations are taken:

- The level of training applied to the experimental subjects, will depend on the knowledge they have about the techniques;
- The programming language Java is the best choice to use, because developers have instruction about during some stage of the student or professional life;
- The common tools are Eclipse as a programming IDE, JUnit for the construction of unit tests and Cucumber for the BDD application.
- The selection of the programming exercises will depend on the level of expertise of the subjects, but highlight the code katas: mars rover, bowling score, roman numerals and string calculator;
- The evaluation of the results obtained will depend on the objective of the experiment, executed under some metrics such as: external quality metric (QLTY), productivity metric (PROD), McCabe's cyclomatic complexity, Branch coverage, lines of code per man hour, others.

4. Empirical experiment

This section provides the description about the empirical experiment, with the aim of showing the impact in the quality of the software and in the developer productivity, when test based development techniques are applied.

The software industry, in recent years, has adopted agile methodologies in its daily tasks; and within this approach, there are techniques such as TDD, BDD, among others; which promise mainly to improve the quality of the software and the productivity of the programmers. As shown previously in Chapter 2, researchers have exposed Test-Driven Development to several scientific experiments with the aim of validating the advantages offered by their practice in the software development. Among them, this technique is compared with Test-Last Development (TLD) or with Incremental Test-Last (ITL), through the resolution of programming exercises (code katas).

4.1. Research methodology

The goal of this empirical experiment is to analyze the impact on the quality of software and on the developer productivity produced by applying test-based techniques in software development. This will be achieved through four related steps:

- Step 1: Teach a group of computer systems engineering students about: Software Testing, JUnit, Incremental Test-Last, Test-Driven Development and Behavior-Driven Development;
- Step 2: Provide a workshop about software development and testing techniques with the execution of several exercises (code katas);
- Step 3: Provide a challenge to the students so that they can apply the techniques in an autonomous way;
- Step 4: Analyze and evaluate the results obtained with the challenge, in order to show the impact in the quality of the software developed and productivity with the mentioned techniques.

4.1.1. Research questions

The experiment is focused on the following research questions with regard to three outcomes: external software quality (fulfillment of stakeholder requirements), internal software quality (way that the system has been constructed) and developer productivity (unit of product delivered – feature - by unit of time consumed in its development). External quality is based on functional correctness, specifically average percentage correctness [33], [75]. Internal code quality deals with the code quality in-terms of code complexity, test coverage, branch coverage, coupling and cohesion between objects [73]. Productivity is based on speed of production, and/or amount of functionality delivered per unit effort [33], [75].

- RQ1: Does TDD improve external code quality compared to ITL?
- RQ2: Does TDD improve internal code quality compared to ITL?
- RQ3: Does TDD improve productivity compared to ITL?
- RQ5: Does BDD improve external code quality compared to ITL?
- RQ5: Does BDD improve internal code quality compared to ITL?
- RQ6: Does BDD improve productivity compared to ITL?
- RQ7: Does BDD improve external code quality compared to TDD?
- RQ8: Does BDD improve internal code quality compared to TDD?
- RQ9: Does BDD improve productivity compared to TDD?

4.1.2. Hypothesis Formulation

Hypothesis testing is the base for the statistical analysis of an experiment [73]. The following hypotheses are stated for the above mentioned research questions:

External Code Quality for RQ1 - Null hypothesis is that the two studied techniques (TDD and ITL) present the same external code quality (H_0^1), with the alternate hypotheses that TDD (H_1^1) or ITL (H_2^1) presents greater external code quality.

$$H_0^1: ExtQlty_{TDD} = ExtQlty_{ITL}$$

$$H_1^1: ExtQlty_{TDD} > ExtQlty_{ITL}$$

$$H_2^1: ExtQlty_{TDD} < ExtQlty_{ITL}$$

Internal Code Quality for RQ2 - Null hypothesis is that the two studied techniques (TDD and ITL) present the same internal code quality (H_0^2), with the alternate hypotheses that TDD (H_1^2) or ITL (H_2^2) presents greater internal code quality.

$$H_0^2: IntQlty_{TDD} = IntQlty_{ITL}$$

$$H_1^2: IntQlty_{TDD} > IntQlty_{ITL}$$

$$H_2^2: IntQlty_{TDD} < IntQlty_{ITL}$$

Productivity for RQ3 - Null hypothesis is that the two studied techniques (TDD and ITL) present the same productivity (H_0^3), with the alternate hypotheses that TDD (H_1^3) or ITL (H_2^3) presents greater productivity.

$$H_0^3: Prod_{TDD} = Prod_{ITL}$$

$$H_1^3: Prod_{TDD} > Prod_{ITL}$$

$$H_2^3: Prod_{TDD} < Prod_{ITL}$$

External Code Quality for RQ4 - Null hypothesis is that the two studied techniques (BDD and ITL) present the same external code quality (H_0^4), with the alternate hypotheses that BDD (H_1^4) or ITL (H_2^4) presents greater external code quality.

$$H_0^4: ExtQlty_{BDD} = ExtQlty_{ITL}$$

$$H_1^4: ExtQlty_{BDD} > ExtQlty_{ITL}$$

$$H_2^4: ExtQlty_{BDD} < ExtQlty_{ITL}$$

Internal Code Quality for RQ5 - Null hypothesis is that the two studied techniques (BDD and ITL) present the same internal code quality (H_0^5), with the alternate hypotheses that BDD (H_1^5) or ITL (H_2^5) presents greater internal code quality.

$$H_0^5: IntQlty_{BDD} = IntQlty_{ITL}$$

$$H_1^5: IntQlty_{BDD} > IntQlty_{ITL}$$

$$H_2^5: IntQlty_{BDD} < IntQlty_{ITL}$$

Productivity for RQ6 - Null hypothesis is that the two studied techniques (BDD and ITL) present the same productivity (H_0^6), with the alternate hypotheses that BDD (H_1^6) or ITL (H_2^6) presents greater productivity.

$$H_0^6: Prod_{BDD} = Prod_{ITL}$$

$$H_1^6: Prod_{BDD} > Prod_{ITL}$$

$$H_2^6: Prod_{BDD} < Prod_{ITL}$$

External Code Quality for RQ7 - Null hypothesis is that the two studied techniques (BDD and TDD) present the same external code quality (H_0^7), with the alternate hypotheses that BDD (H_1^7) or TDD (H_2^7) gives greater external code quality.

$$H_0^7: ExtQlty_{BDD} = ExtQlty_{TDD}$$

$$H_1^7: ExtQlty_{BDD} > ExtQlty_{TDD}$$

$$H_2^7: ExtQlty_{BDD} < ExtQlty_{TDD}$$

Internal Code Quality for RQ8 - Null hypothesis is that the two studied techniques (BDD and TDD) present the same internal code quality (H_0^8), with the alternate hypotheses that BDD (H_1^8) or TDD (H_2^8) presents greater internal code quality.

$$H_0^8: IntQlty_{BDD} = IntQlty_{TDD}$$

$$H_1^8: IntQlty_{BDD} > IntQlty_{TDD}$$

$$H_2^8: IntQlty_{BDD} < IntQlty_{TDD}$$

Productivity for RQ9 - Null hypothesis is that the two studied techniques (BDD and TDD) present the same productivity (H_0^9), with the alternate hypotheses that BDD (H_1^9) or TDD (H_2^9) presents greater productivity.

$$H_0^9: Prod_{BDD} = Prod_{TDD}$$

$$H_1^9: Prod_{BDD} > Prod_{TDD}$$

$$H_2^9: Prod_{BDD} < Prod_{TDD}$$

4.1.3. Experiment description

A controlled experiment was made to 22 students of the Systems Engineering Degree of the Universidad Técnica del Norte (Ibarra - Ecuador). It consisted of three phases: Knowledge, Training, and Practice; and had a duration of approximately 30 hours.

In the initial phase, the **Knowledge phase**, concepts such as: Introduction to agile development, Testing, JUnit, Incremental Test-Last, Test-Driven Development,

Behavior-Driven Development, and Cucumber [83], were provided. In addition, at the end of the explanation of each of the techniques, a small example was used. It consisted of the development of a ***simple calculator***, that allows the calculus operations of adding and dividing. The application didn't have graphical interface to allow students to focus on the understanding of the execution of the technique.

In the **Training phase**, two code katas were developed with the students: Rock Paper Scissors (RPS) and Roman Numerals (RM). They were developed using the techniques chosen for the experiment (ITL, TDD and BDD). RPS is a traditional game involving two players making pre-defined hand gestures [84]. Each player gesture is played against the other, with a winner being decided based on the rules. RM is about converting Arabic numbers into their Roman numeral equivalents, and vice versa [85]. Additionally, an extra code kata (Numbers in Words) was proposed to be solved by the students (autonomously). The main goal of this exercise is to convert numbers into words [86].

In the third phase, the **Practice phase**, students developed completely autonomously, two code katas through the techniques learned: one FizzBuzz variant (FB) and String Calculator (SC). FB is a counting and number replacement game, where: any number that is divisible by three is replaced with the word 'fizz', any number divisible by five is replaced with the word 'buzz', any prime number is replaced with the word 'whiz', any number simultaneously divisible by three and five is replaced with 'fizz buzz', any prime number divisible by three is replaced with 'fizz whiz', and any prime number divisible by five is replaced with 'buzz whiz' [87], [88]. SC is a task to build a simple *string calculator* with a simple add method [89]. This method receives a string with some numbers separated by one or multiple delimiters, and returns the addition of all numbers. An estimation of four hours' duration was made to ensure the total resolution of each exercise.

It is important to emphasize that each code kata was evaluated with the ***function point metric***. This metric is considered the main tool for the functional measurement of software products and the processes involved in their development; allowing to quantify the size of a system in independent units of the programming language, methodologies, platforms and/or technologies used, called Function Points [90], [91]. In addition, it includes concepts that can be understood by both programmers and users.

The intention was to propose exercises of similar difficulty both in the training phase as in the practice phase. For example: RM and SC have approximately 24 point of function (see Appendix B).

4.1.4. Factors and metrics

The experiment was based upon two factors. The development approach level [33]: ITL, TDD or BDD, will be used as the main factor; while the tasks [33] corresponding to the development of code katas (FB and SC) will be used as a secondary factor. The effectiveness of the development approach will be studied under the perspective of the experiments [33], [73], [75].

The external quality metric (QLTY) represents the degree of agreement of the system with the functional requirements [33], [75]. The formula for calculating QLTY is defined as:

$$QLTY = \frac{\sum_{1}^{\#TUS} QLTY_i}{\#TUS} * 100$$

where $QLTY_i$ is the quality of *i*th user history implemented by the subject. $QLTY_i$ is defined as:

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

In turn, the number of user stories addressed (#TUS) is defined as:

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

where n is the number of user stories that make up the task. In both cases, it represents the number of passing JUnit assert statements in the set of tests associated with the *i*th user history. Consequently, a user history is considered addressed if it passes at least one of its JUnit assert statements. Accordingly, a user story is considered as tackled if at least one of its JUnit assert statements passes [75].

For example, supposing that a person tackles two user stories (#TUS = 2); this means there are two user stories for which at least one assert statement passes in the test suite. Let's assume the acceptance tests of the first tackled user story contain twelve assertions, out of which six are passing. The second user story's acceptance tests

contain nine assertions, and three are passing. The quality value of the first tackled user story (QLTY1) is 0.50, and the second user story has a quality value of 0.33 (QLTY2). Therefore, the QLTY measure for the subject is 41.5 percent, i.e, (QLTY = (0.50 + 0.33) / 2 * 100).

The productivity metric (PROD) represents the work done by the subjects, with the required quality, and within the specified time [75]. Its formula is defined as:

$$PROD = \frac{OUTPUT}{TIME}$$

OUTPUT symbolizes the percentage of passing JUnit assert statements found in the set of tests for a task.

$$OUTPUT = \frac{\#ASSERT(PASS)}{\#ASSERT(ALL)} * 100$$

TIME (minutes) is an estimate of the amount of work used in the resolution of a task, and is based on the time records (milliseconds) collected by the Integrated Development Environment (IDE).

$$TIME = \frac{t_{close} - t_{open}}{6000}$$

For example, a person implements a task with a total of 50 assert statements in a test suite. After running the acceptance test suite against the person's solution, 40 assert statements are passing. Then OUTPUT = (40 / 50) x 100 = 80%. Suppose that the solution was delivered in one hour and a half (i.e., TIME = 90 minutes). The person's PROD is therefore 0.89, denoting an assertion passing rate of 0.89 percent per minute.

Regarding the *internal quality* analysis, we used the metric referred in the experiment by Munir et al. [73], McCabe's cyclomatic complexity metric, that provides a quantitative measurement of the logical complexity of a software; that is, it indicates how a program can be difficult to test and maintain [73], [74]. The level of code complexity is presented in TABLE VIII.

TABLE VIII. McCabe's cyclomatic complexity level.

Complexity Number	Meaning
1-10	Structured and well written code High Testability Cost and Effort is less
11-20	Complex Code Medium Testability Cost and effort is Medium

21-50	Very complex Code Low Testability Cost and Effort are high
>50	Not at all testable Very high Cost and Effort

For instance, read the method to validate the type of triangle presented in Figure 8.

```
public String triangleValidator (int a, int b, int c){
    if (a <= 0 || b <= 0 || c <= 0){
        return "Invalid triangle";
    }
    if ((a > b + c || b > a + c || c > a + b)){
        return "Invalid triangle";
    }
    if (a == b && b == c){
        return "Equilateral triangle";
    }
    if (a == b || b == c || a == c){
        return "Isosceles triangle";
    }
    return "Scalene triangle";
}
```

Figure 8. Type of triangle validator method.

The logical complexity of the method is 12 (complex code, cost and effort is medium) and it was calculated in this form:

- Normal flow complexity is 1;
- if (a <= 0 || b <= 0 || c <= 0), complexity is 3;
- if ((a> b + c || b> a + c || c> a + b)), complexity is 3;
- if (a == b && b == c), complexity is 2;
- if (a == b || b == c || a == c), complexity is 3.

Furthermore, the Source Code Analyzer PMD was applied, to find common programming flaws like: unused variables, empty catch blocks, unnecessary object creation, and so forth [92]. Java PMD rules considered to evaluate each code kata resolution are part of the categories: best practices, code style and error prone.

4.1.5. Development Environment Operationalization

The development environment that the participants used included: Java 8 using the IntelliJ IDEA with 4 additional plugins: Cucumber, Activity Tracker, Metrics Reloaded and QAPlug. The Cucumber plugin will allow implementing the BDD technique in the resolution of the exercises. The Activity Tracker plugin is projected to track and record the activity of the IDE user, such as the time spent on tasks. McCabe's cyclomatic complexity metric will be applied with the use of the Metrics Reloaded plugin. In addition, the QAPlug plugin implements the PMD module to manage code quality.

4.1.6. Design

Opting for the modality where all the subjects go through all the experimental conditions, and having 3 types of interventions (ITL, TDD and BDD); the Latin Square design [93] (balanced crossover) was selected. In a Latin Square, each subject receives each intervention once [94]. Figure 9 shows the schematic representation of three-period Latin Square design selected. The rows are different groups and the columns represent different periods. Each group undergoes a sequence of three interventions over three periods (e.g., Group 1 receives Intervention ITL in Period 1, Intervention TDD in Period 2, and Intervention BDD in Period 3). The assignment of subjects to groups was random [95].

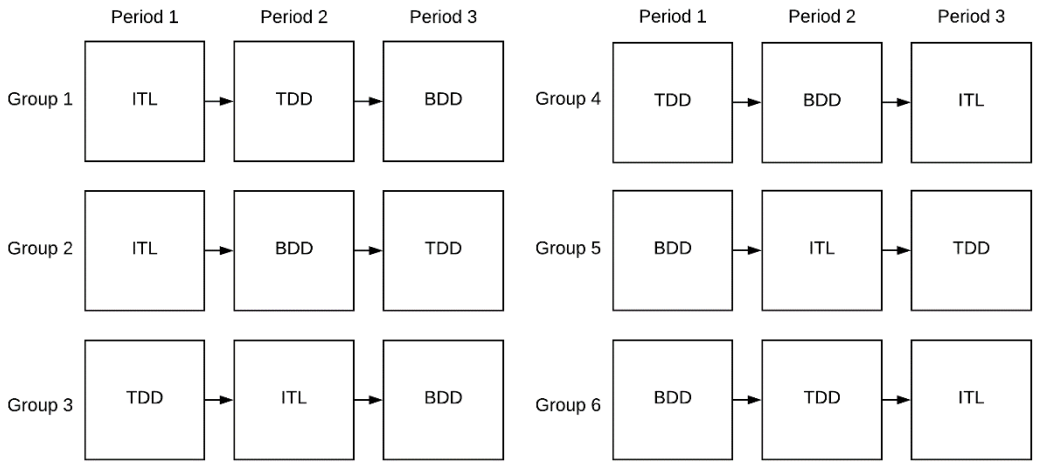


Figure 9. Balanced Latin Square Design.

4.2. Experiment´s execution and data collection

This section tries to present, in a summarized way, the process of execution of the experimentation and the data collection. The experiment was conducted over a workshop that occurred in the course of Software Engineering II (7th level). No one had knowledge about the techniques and their previous professional experience was non-existing, the expertise as programmers was based on the levels scaled into the professional career. Individuals have used procedural and object-oriented languages; and for knowledge imparted in classrooms, 100% have worked with JAVA. The data collection was done requesting students to submit the unit test files that covered all of the requirements of the challenges proposed and the additional plugins installed in IntelliJ IDEA.

4.2.1. Classes and materials

The knowledge taught as well as the material used in the classrooms were previously approved by the teacher MSc. Antonio Quiña, who offered his full support during the process. The organization of the workshop is presented in Figure 10, where the dates of execution of the experiment phases can be clearly differentiated; considering its development in the months of April and May respectively.

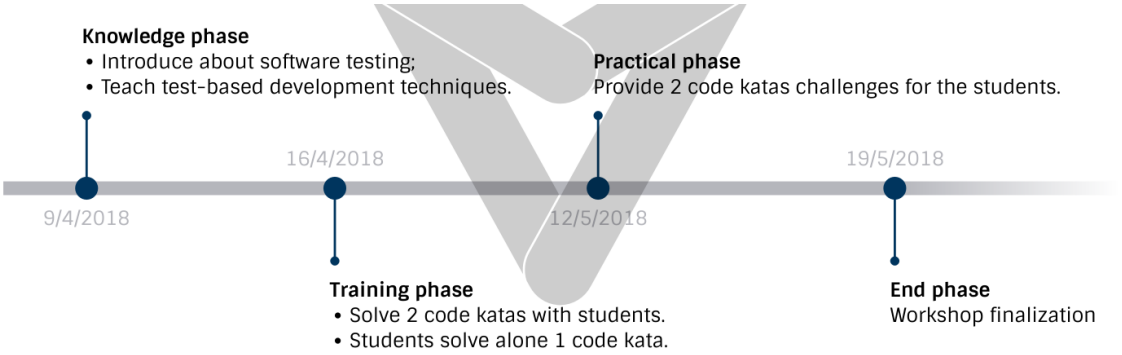


Figure 10. Workshop timeline

The materials used in the classroom were included into a document presented in **Appendixes C, D, E and F**; which has theory, examples and important links (references) that serve for the continuous learning of the students.

Each one of the code katas was explained by a slide deck, and the students downloaded a .zip file with a similar structure to the one presented in Figure 11.

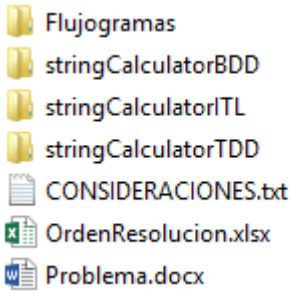


Figure 11. Code Katas ZIP content

- Flow folder (Flujogramas) - contains the images of the resolution flow of the TDD and BDD techniques, and the ITL practice;
- Project folders (e.g.: stringCalculatorBDD, stringCalculatorTDD, stringCalculatorITL) - projects configured for the execution of the programming exercise through ITL, TDD and BDD;
- Considerations file (CONSIDERACIONES.txt) - contains the general considerations of the execution of the exercise such as: use of the Activity Tracker plugin, not copying code made in one technique to another, respecting the requirements of the exercise, following the steps of the techniques, among others;
- Resolution order file (OrdenResolucion.xlsx) - contains the order of resolution of the exercise presented by the Latin Square where the individuals are organized by groups. This file was not presented in the training phase;
- Problem file - It has all the specifications referring to the problem, requirements, and possible test scenarios, among others.

In addition, the resolution of the exercises carried out in the classroom as those executed in the experimental phase, together with the .CSV file of the Activity Tracker plugin; they were attached to tasks of the virtual content management platform that supports the courses of the institution (moodle).

4.2.2. Unit Test classes data

The developer's productivity (PROD) metrics and, external quality (QLTY) are based on the number of assertions passed in software tests. To provide this information, some test classes were created in base on the requirements of each code kata.

TABLE IX and TABLE X show a summary of the assertions data presented in these classes, related to the FizzBuzz Kata and String Calculator Kata exercises respectively.

TABLE IX. FizzBuzz Kata test description.

Code Kata	FizzBuzz	**User Stories**	2	**Assertions**	22
Game logic - US 1					
Normal Numbers		Multiples of three		Multiples of five	
Number 1 returns "1". Number 4 returns "4". Number 8 returns "8".		Number 6 returns "Fizz". Number 9 returns "Fizz". Number 27 returns "Fizz".		Number 10 returns "Buzz". Number 25 returns "Buzz". Number 35 returns "Buzz".	
Primes		Multiples of three and five		Multiples of three and primes	
Number 2 returns "Whiz". Number 7 returns "Whiz". Number 11 returns "Whiz".		Number 15 returns "FizzBuzz". Number 45 returns "FizzBuzz". Number 75 returns "FizzBuzz".		Number 3 returns "FizzWhiz".	
Multiples of five and primes		Game exceptions			
Number 5 returns "BuzzWhiz".		Number 0 returns the message "Negatives numbers or Zero are not allowed: 0". Number -99 returns the message "Negatives numbers or Zero are not allowed: -99 ".			
Game sequence - US 2					
Range [20, 30] returns "Buzz Fizz 22 Whiz Fizz Buzz 26 Fizz 28 Whiz FizzBuzz". Range [100, 105] returns "Buzz Whiz Fizz Whiz 104 FizzBuzz". Range [395, 408] returns "Buzz Fizz Whiz 398 Fizz Buzz Whiz Fizz 403 404 FizzBuzz 406 407 Fizz".					

TABLE X. String Calculator Kata test description.

Code Kata	String Calculator	**User Stories**	4	**Assertions**	33
General rules - US 1					
Empty String		Basic values		Basic Sum	
String "" returns 0.		String "0" returns 0. String "1" returns 1. String "2" returns 2. String "8" returns 8.		String "1,2" returns 3. String "5,3,4" returns 12. String "3,6,15,18,46,33" returns 121.	
Long Sum				Other basic delimiter	

String "1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20" returns 210.	String "1\n2" returns 3. String "1\n2\n3" returns 6. String "1\n5,1\n7" returns 14.
Allow delimiters - US 2	
Simple Delimiters	Multiple Simple Delimiters
String "//;\n1;2" returns 3. String "//:\n1:2,7" returns 10. String "//%\n1%7" returns 8.	String "//[#][%]\n1#2%3" returns 6. String "//[=][_][&]\n120=200_105&25_10" returns 460.
Big Delimiters	Multiple Big Delimiters
String "//[***]\n8***2***9" returns 19. String "//[! -!]\N10!-!6!-!3" returns 19.	String "//[***][$$$]\n7***20$$$19" returns 46. String "//[***][%-!]\n128***11%-!100" returns 239.
Conditions of Calculator - US 3	
Exceptions	Greater than 1000 are not included
String "-10,12" returns the message "Negatives Not Allowed: -10". String "-3,2, -23,5" returns the message "Negatives Not Allowed: -3, -23".	String "100\n2\n1121\n20" returns 122. String "3,1000,1001,6,1234" returns 1009.
Check functionality - US 4	
String "3" returns 3. String "3,6,15,18,46,33" returns 121. String "3,6\n15" returns 24. String "//;\n9;6;15" returns 30. String "3,1000,1001,6,1234" returns 1009. String "//[=][_][&][!]\n280=20_35&125!999" returns 1459. String "//[#%?]\n28#%?20#%?999" returns 1047. String "//[?@!][%-!]\n128?@!2%-!99" returns 229.	

4.2.3. Activity Tracker plugin data

The Activity Tracker plugin tracks and records the IDE user (developer) activity, storing it in a CSV file. This file allowed verifying if the subject has executed the technique correctly. Additionally, a very simple program was created to read and summarize the exercise resolution done by the subject (activity in CSV file). For example, the development of an exercise through TDD is presented in Figure 12.

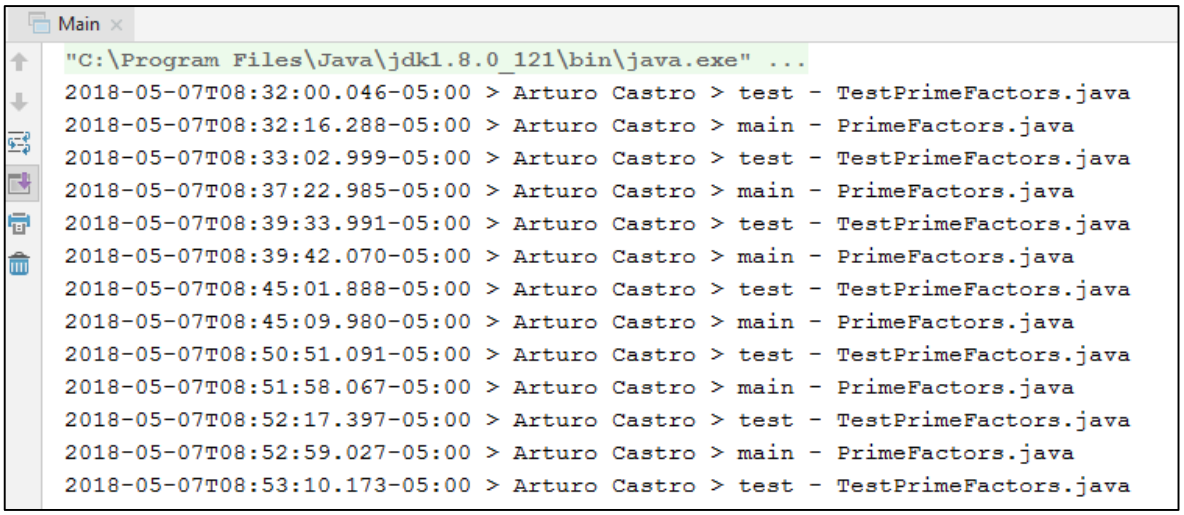


Figure 12.User activity captured from Activity Tracker plugin.

In addition, the plugin provides the extraction of the time invested in the project. This value is used within the productivity metric; and an example of this is present in Figure 13.

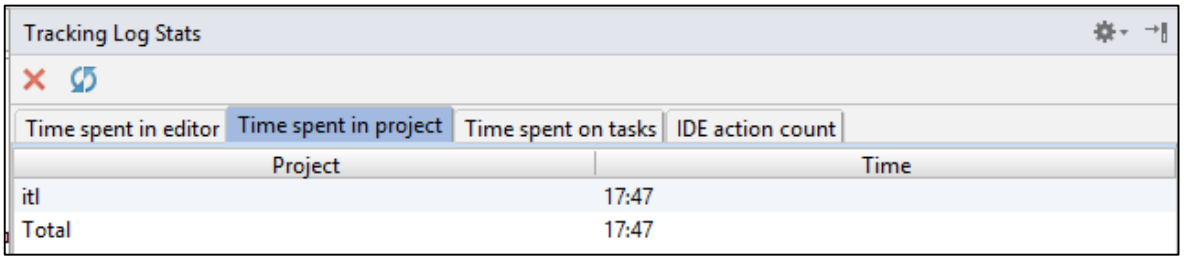


Figure 13. Time spent in project from Activity Tracker plugin.

4.2.4. Metrics Reloaded plugin data

In order to quantify the complexity of the software, the McCabe's cyclomatic complexity metric was used. It was applied in a simple form using the Metrics Reloaded plugin; which has various software metrics including the aforementioned.

A very simple example would be to have the resolution of the FizzBuzz Kata game presented in Figure 14.

```java
public class FizzBuzzWhiz {

    public String getResult(int number) {
        StringBuilder stringBuilder = new StringBuilder();
        if (isDivisible(number, divisor: 3)){stringBuilder.append("Fizz");}
        if (isDivisible(number, divisor: 5)){stringBuilder.append("Buzz");}
        if (isPrime(number)){stringBuilder.append("Whiz");}
        if (stringBuilder.length() > 0){return stringBuilder.toString();}
        return String.valueOf(number);
    }

    public boolean isDivisible(int number, int divisor) {
        return (number % divisor == 0);
    }

    public boolean isPrime(int number){
        if (number == 1)  {return false;}
        for (int count = 2; count < number; count++) {
            if (number % count == 0) { return false; }
        }
        return true;
    }
}
```

Figure 14. FizzBuzz Kata resolution

The execution of the plugin obtains the complexity value of the resolution. Figure 15 shows the result of the examination of the exercise, in this case it has a value of 10; meaning an structured and well written code (see TABLE VIII). This number represents the total of the addition of the complexity of each of the present methods, and is considered to the analysis.

Method metrics | Class metrics | Package metrics | Module metrics | Project metrics

method	ev(G)	iv(G)	v(G)
fizzbuzz.FizzBuzzWhiz.getResult(int	2	5	5
fizzbuzz.FizzBuzzWhiz.isDivisible(in	1	1	1
fizzbuzz.FizzBuzzWhiz.isPrime(int)	4	1	4
Total	**7**	**7**	**10**

Figure 15. FizzBuzz cycomatic analysis result

4.2.5. QAPlug plugin data

The analysis of the number of errors entered in the code is done by integrating PMD in IntelliJ IDEA using the QAPlug plugin. PMD allows evaluating the static quality of the code through established and configurable rules in XML format. For example,

taking into account the method to check if a number is prime of the resolution of FizzBuzz Kata (see Figure 14). If we execute it through the plugin, we will obtain the result of Figure 16, where a rule fails. That is, the total number of errors entered in the code is 1.

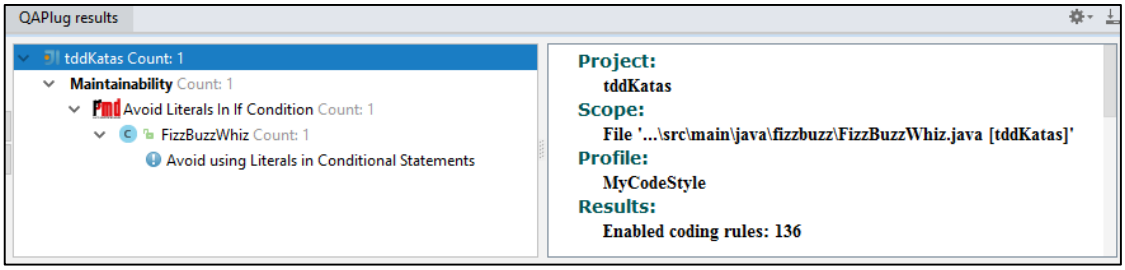


Figure 16. FizzBuzz PMD analysis result

This error belongs to the rule "AvoidLiteralsInIfCondition", which proposes to avoid the use of encoded literals in conditional statements [96]. The mitigation of this error is presented in Figure 17, where this value is declared as a static variable; improving the code's maintenance capacity.

```
private static final int ONE_ISNT_PRIME = 1;

public boolean isPrime(int number){
    if (number == ONE_ISNT_PRIME)  {return false;}
    for (int count = 2; count < number; count++) {
        if (number % count == 0) { return false; }
    }
    return true;
}
```

Figure 17. Prime numbers method changes

5. Data analysis and Results

This section contains the analysis of descriptive statistics for the data collected by metrics during the experimental phase. In addition, it responds to the hypotheses exposed in the previous chapter.

5.1. Frequency

Next, the description of grouping of values for each one of the analysis variables is presented, by frequency graphs.

5.1.1. External Quality

The data collected from the FizzBuzz code kata (see Figure 18) show that most subjects are in the fourth statistical range; This means that they have completed 75% to 100% of the exercise. Of the total sample, 90.91% of subjects reached this range when using ITL. On the contrary, when applying TDD or BDD, there was an increase between individuals of 4.55%.

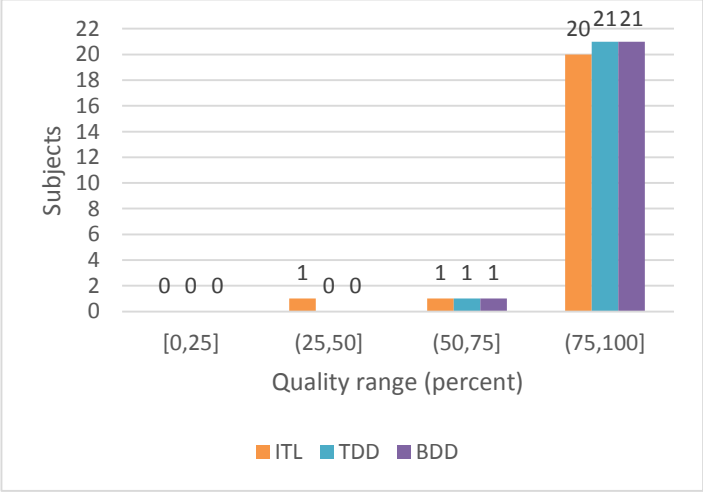


Figure 18. FizzBuzz: external quality data frequencies

Figure 19, presents a greater grouping of the subjects in the fourth statistical range, this means that they have completed 75% to 100% of the String Calculator exercise. Among from the sample group, 54.55% of subjects reached this range when using ITL. On the contrary, when applying TDD or BDD, an increase between individuals of 9.09% and 13.64% respectively was evidenced.

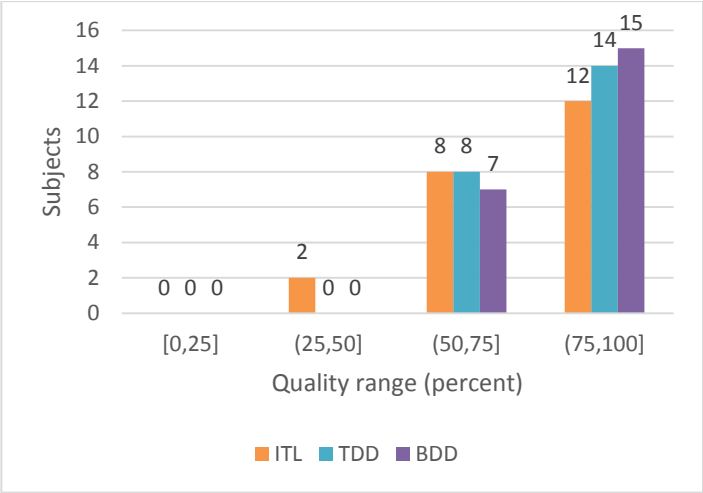


Figure 19. String Calculator: external quality data frequencies

5.1.2. Productivity

During the development of the FizzBuzz exercise (see Figure 20), 36.36% of subjects managed to obtain from 1% to 2% of productivity per minute applying ITL or TDD. When using BDD, half of the individuals (50%) improved their productivity entering the third statistical range (2,3).

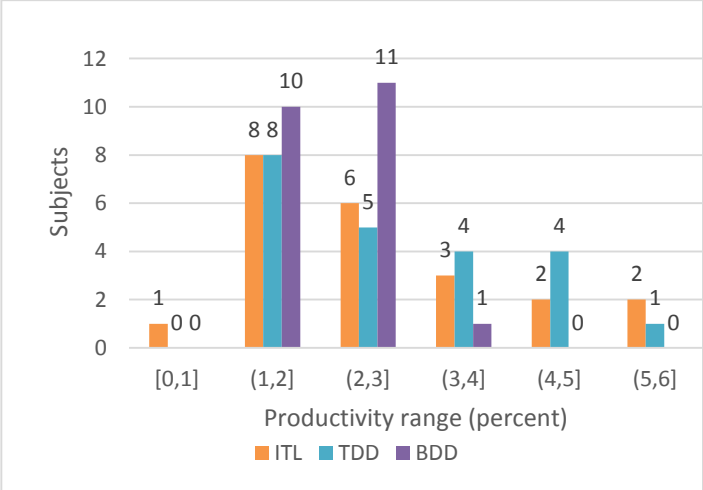


Figure 20. FizzBuzz: productivity per minute data frequencies

Figure 21, presents the productivity data for String Calculator; where the largest number of subjects are in the second statistical range, obtaining 1% to 2% of productivity per minute when developing the exercise. ITL and BDD obtained 54.55% of subjects. On the contrary, TDD shows 40.91% of individuals in this quadrant, presenting a decrease of 13.64% of people.

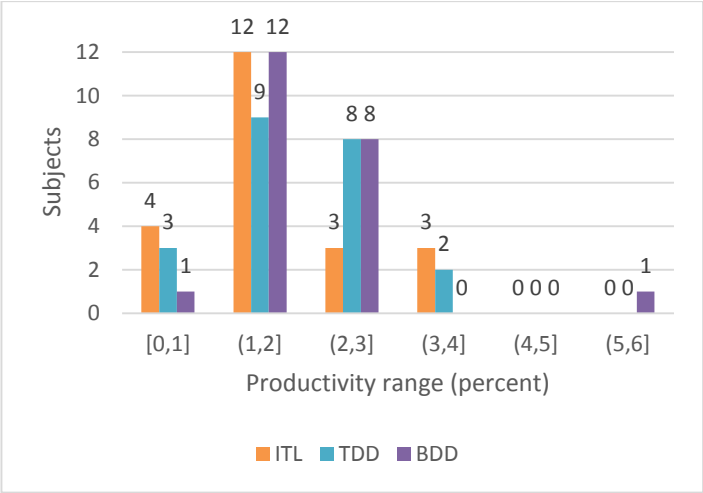


Figure 21. String Calculator: productivity per minute data frequencies

5.1.3. Internal Quality

A) McCabe's Cyclomatic complexity

The resolution of the code kata FizzBuzz (see Figure 22), shows that the majority of individuals are in the second statistical range; that is, they manage to build complex code software (present in TABLE VIII). TDD, stands out within the category with 71.43% of the sample. ITL, is in second place with 66.67% of subjects. The smallest number of people within this range was for BDD, which presents 57.14%.

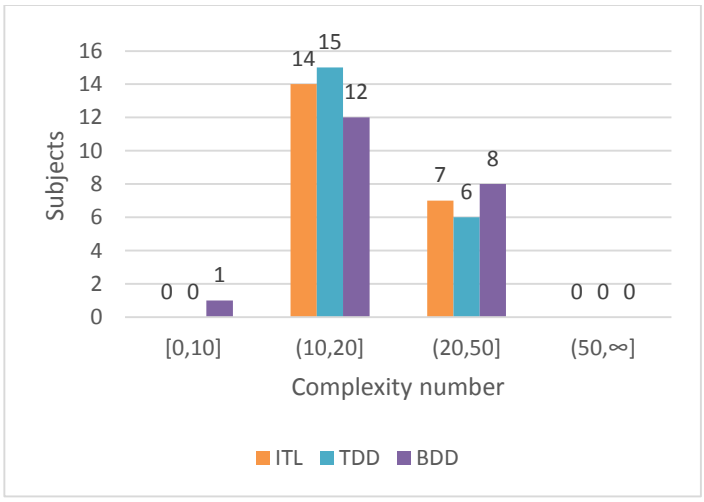


Figure 22. FizzBuzz: code complexity data frequencies

Figure 23 shows the String Calculator data regarding the complexity of the code; where it is observed that a large number of subjects in the first statistical rank; that is, they manage to build a software with a well-structured code (present in TABLE VIII).

TDD, stands out within the category with 68.18% of the sample. ITL and BDD, is in second place with 59.09% of subjects.

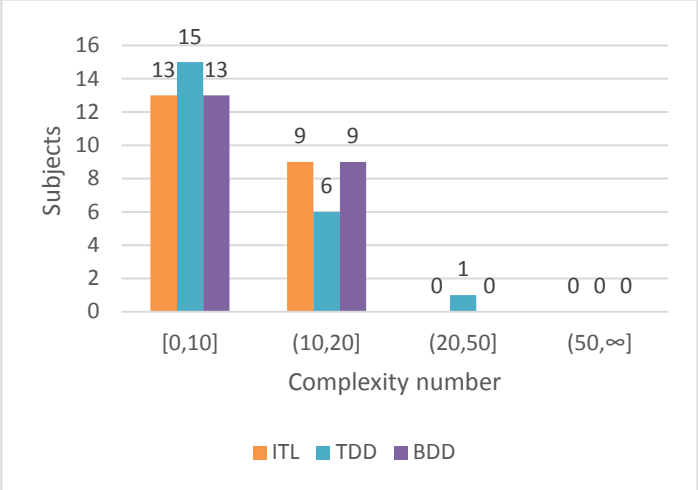


Figure 23. String Calculator: code complexity data frequencies

B) PMD

Most of the solutions for FizzBuzz, presented by the subjects (see Figure 24), are shown until a maximum of 3 errors entered in the code; locating them in the second statistical range. TDD and BDD, obtained 77.27% of individuals. ITL increased this figure reaching 81.82% of people.

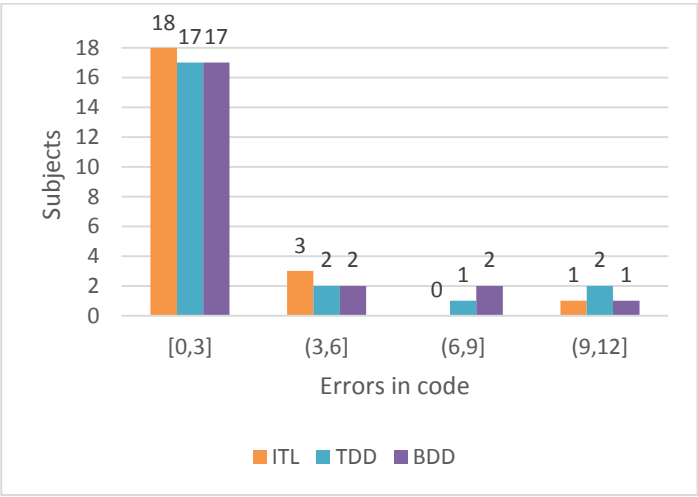


Figure 24. FizzBuzz: PMD analysis data frequencies

The data obtained from the analysis of errors by PMD for String Calculator, are presented in Figure 25. Many of the subjects' solutions show until a maximum of 3 errors introduced in the code; this being located in the second statistical range. ITL and BDD, obtained 59.09% of individuals. TDD increased this figure reaching at 72.73% of people.

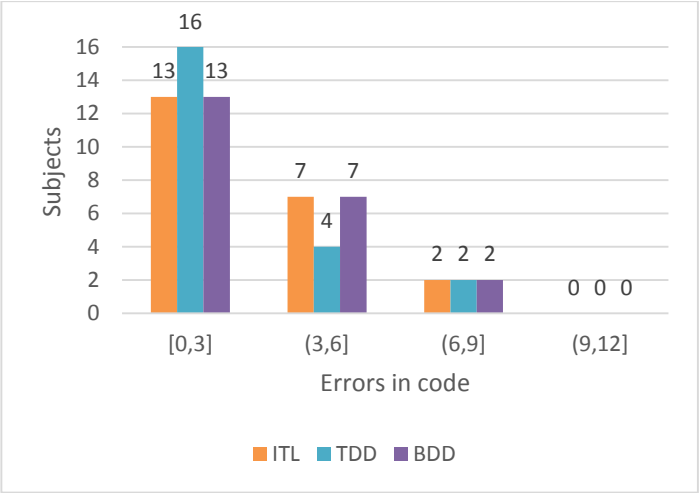


Figure 25. String Calculator: PMD analysis data frequencies

5.2. Symmetry and dispersion

Next, it´s observed the distribution of data and identify the possible outliers presented in each of the analysis variables, making use of box plots.

5.2.1. External Quality

Figure 26, represents the external quality of the FizzBuzz exercise and, indicates that the ITL and TDD data have a "very skewed" asymmetric distribution (negative bias). The range in which the data for ITL is displayed is [86.84%; 100%], for TDD of [89.47%; 100%] and, for BDD it is maintained at 100%; indicating a greater dispersion of the amounts for ITL compared with TDD and a grouping for BDD. We can also see the existence of extreme values for the 3 techniques; this being for ITL the values {73.68%; 31.58%}, for TDD 73.68% and, for BDD {73.68%; 94.74%}. In addition, the average external qualities of the groups of boxes are similar, but the values of some of the groups are more variable than the others.

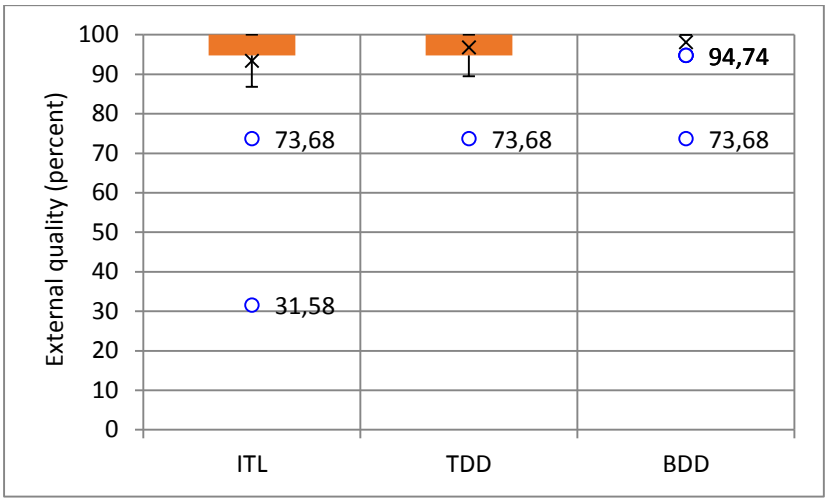


Figure 26. FizzBuzz: external quality box-plot

The external quality data of the String Calculator exercise can be found in Figure 27; and indicate that ITL, TDD and BDD have an asymmetric distribution with a negative bias. The range in which the data for ITL is displayed is [50%; 87.5%], and for both TDD and BDD it is [55.55%; 87.5%]; indicating a greater dispersion of the amounts for ITL. There is an extreme ITL value of 27.08%. In addition, the average extreme quality of all the groups is different.

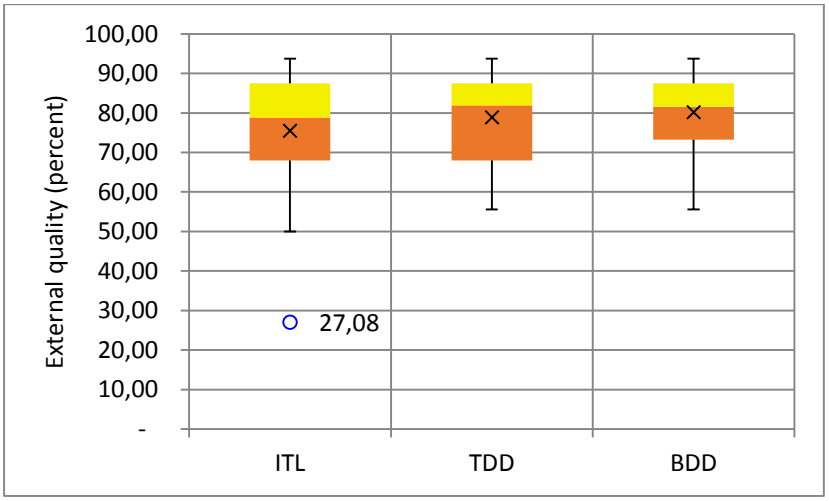


Figure 27. String Calculator: external quality box-plot

5.2.2. Productivity

The productivity chart per minute of the FizzBuzz code kata (see Figure 28), indicates that the ITL and TDD data have an asymmetric distribution with a positive bias; on the contrary, BDD has an asymmetric distribution with a negative bias. The range in which the data for ITL is displayed is [0.54%; 5.33%], for TDD of [1.06%; 5.24%]

and for BDD of [1.08%; 2.93%]; indicating a greater dispersion of the amounts for ITL and TDD compared with BDD. There is an atypical value in BDD of 3.27%. In addition, the average minute productivity of all the groups is different.

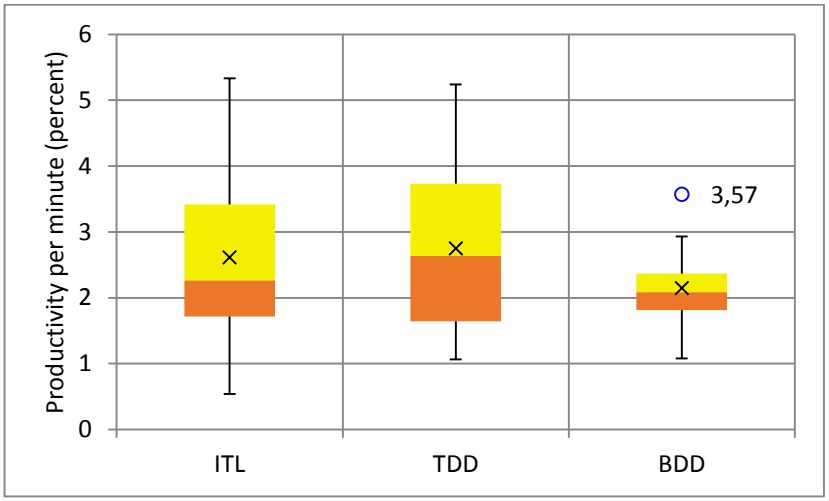


Figure 28. FizzBuzz: productivity per minute box-plot

The productivity per minute for String Calculator is interpreted in Figure 29. The data indicates that ITL and TDD have an asymmetric distribution with a positive bias; on the contrary, BDD has an asymmetric distribution with a negative bias. The range in which the data is displayed for ITL is [0.29%; 3.88%], for TDD of [0.73%; 3.83%] and for BDD of [0.72%; 2.84%]; indicating a greater dispersion of the amounts for ITL and TDD compared to BDD. The existence of an atypical value in BDD can also be appreciated, this being 5.38%. In addition, the average minute productivity of all the groups is different.

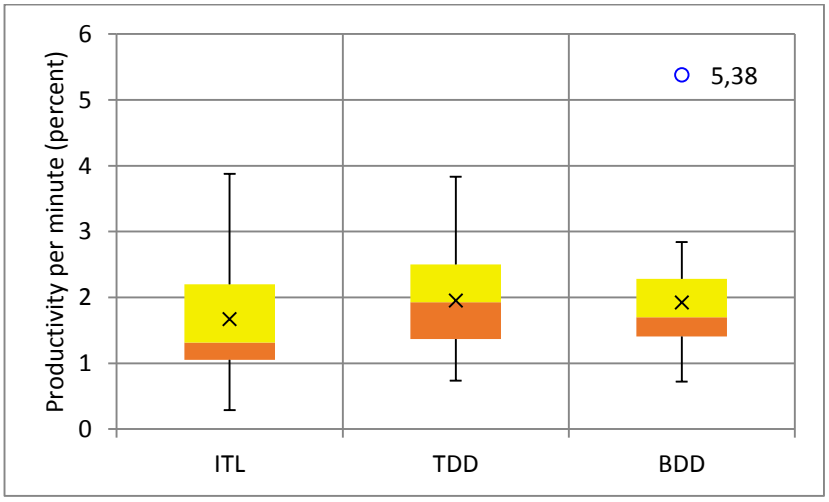


Figure 29. String Calculator: productivity per minute box-plot

5.2.3. Internal Quality

A) McCabe's Cyclomatic complexity

Figure 30, represents the level of code complexity for the FizzBuzz exercise, it indicates that the ITL and BDD data have an asymmetric distribution with a negative bias; but TDD presents a symmetric distribution. The range in which the data for ITL is displayed is [12; 25], for TDD is [12; 24], and for BDD is [8; 21]; indicating a greater dispersion of the amounts for BDD. In addition, the complexity of the average code of all the groups is different.

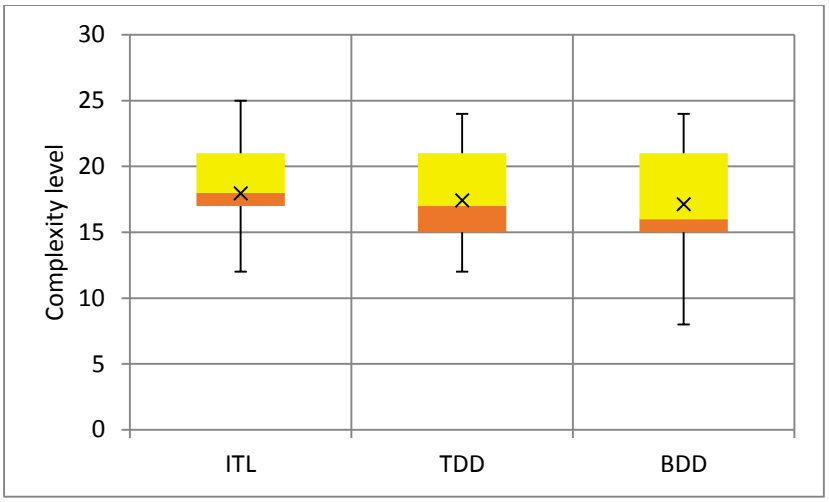


Figure 30. FizzBuzz: code complexity box-plot

The level of code complexity for the String Calulator exercise (see Figure 31), indicates that the ITL and BDD data have an asymmetric distribution with a positive bias; but TDD presents an asymmetric distribution with a negative bias. The range in which the data for ITL is displayed is [3; 19], for TDD is [4; 11], and for BDD is [4; 19]; indicating a smaller dispersion of the amounts for TDD. You can also see the existence of extreme values for 2 of the techniques; this being for ITL the value 20, and for TDD the values {18; 19; 21}. In addition, the complexity of the average code for ITL is different.

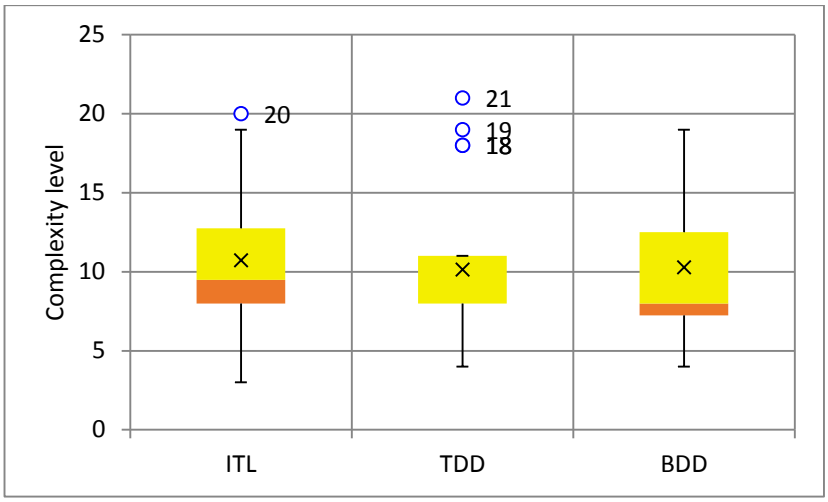


Figure 31. String Calculator: code complexity box-plot

B) PMD

The chart of the number of errors in code for the FizzBuzz kata (see Figure 32), indicates that the data of TDD and BDD have an asymmetric distribution with negative bias; but ITL presents an asymmetric distribution with a positive bias. The range in which the data for the three techniques is shown is [1; 3]; noting the same dispersion of data quantities. You can also see the existence of extreme values, being for ITL {4; 5; 12}, for TDD {5; 8; 11}, and for BDD {4; 5; 7; 8; 12}. In addition, the number of errors in average code are similar, but the values of some of the groups are more variable than the others.

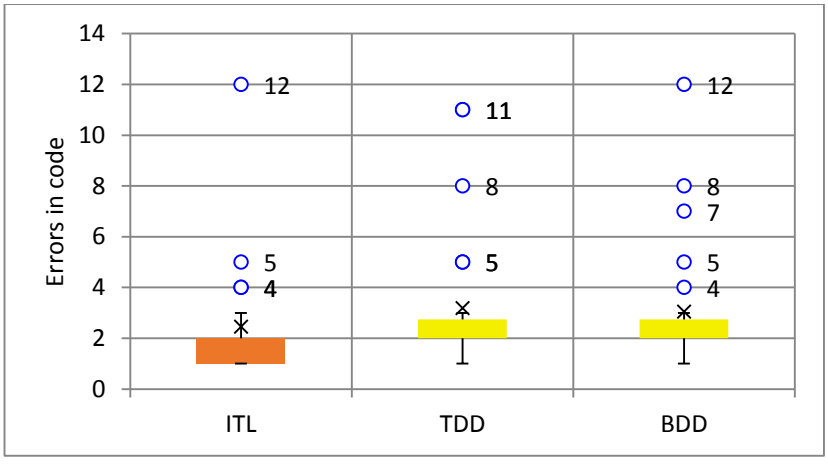


Figure 32. FizzBuzz: PMD analysis box-plot

Figure 33 presents the information of the number of errors in code for the String Calculator kata. The graph indicates that ITL has a symmetric distribution; TDD has an asymmetric distribution with a negative bias and BDD presents an asymmetric distribution with a positive bias. The range in which the data is displayed for ITL and

BDD is [0; 7], and for TDD is [0; 5]; indicating a smaller dispersion of the amounts for TDD. The existence of an extreme value for TDD can also be appreciated; being this 7. In addition, the number of errors in average code for the groups is different.

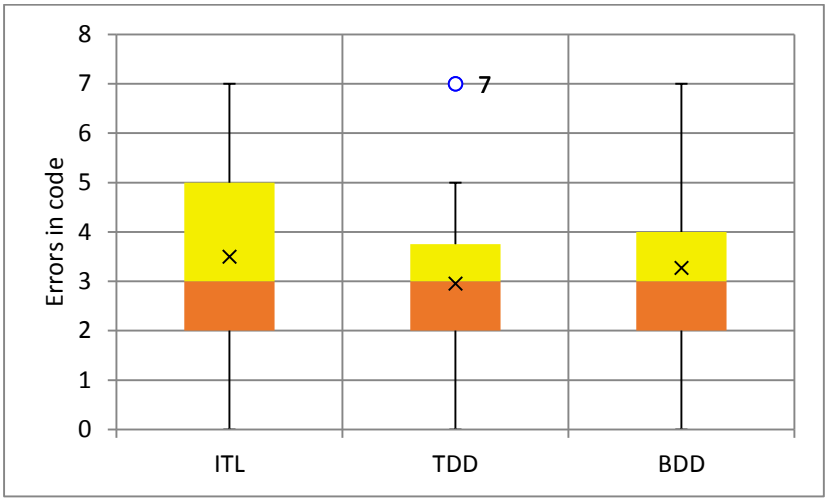


Figure 33.String Calculator: PMD analysis box-plot

5.3. Development trends

Next, the most relevant data of the development of the code katas FizzBuzz (FB) and String Calculator (SC) are described; which are detailed in TABLE XI and TABLE XII respectively.

TABLE XI. FizzBuzz metrics' data

Subject	Sex	Resolution Order	External Quality (percent)			Productivity (percent per minute)			McCabe's Cyclomatic complexity			PMD (number of errors in code)		
			ITL	TDD	BDD	ITL	TDD	BDD	ITL	TDD	BDD	ITL	TDD	BDD
S1	M	ITL → TDD → BDD	94,74	94,74	94,74	1,59	2,40	2,51	25	24	24	1	8	8
S2	M	ITL → TDD → BDD	94,74	89,47	94,74	1,51	1,70	2,07	17	15	15	12	11	12
S3	M	ITL → BDD → TDD	86,84	100	100	1,45	4,51	1,85	18	18	16	4	11	1
S4	M	TDD → ITL → BDD	100	100	100	1,66	1,63	2,18	15	15	15	4	5	7
S5	M	TDD → ITL → BDD	94,74	97,37	100	2,31	1,78	1,79	24	22	23	3	3	2
S6	M	ITL → BDD → TDD	94,74	94,74	94,74	1,88	2,10	1,63	23	22	23	2	2	2
S7	F	TDD → BDD → ITL	100	100	100	1,94	2,22	2,87	18	18	21	2	2	2
S8	M	TDD → ITL → BDD	100	94,74	100	4,00	1,45	2,88	21	20	21	2	2	4
S9	M	TDD → ITL → BDD	100	89,47	100	2,24	1,13	2,13	18	14	18	2	2	2
S10	M	BDD → ITL → TDD	100	100	100	3,53	3,84	1,70	12	12	12	2	2	2
S11	M	BDD → ITL → TDD	100	100	100	2,19	3,34	2,10	12	12	12	2	2	2
S12	M	TDD → BDD → ITL	73,68	73,68	73,68	1,02	1,06	1,08	12	12	12	2	2	2
S13	F	TDD → BDD → ITL	100	94,74	100	2,65	1,42	1,80	21	22	21	5	5	5
S14	M	BDD → TDD → ITL	100	100	100	3,09	4,01	2,38	21	21	21	1	1	1
S15	M	BDD → TDD → ITL	100	100	100	4,95	2,90	1,54	18	17	16	1	2	2
S16	M	BDD → ITL → TDD	94,74	100	100	1,95	3,07	1,92	17	17	16	1	1	3
S17	M	BDD → ITL → TDD	94,74	100	100	3,60	5,24	1,92	17	17	16	1	1	1
S18	M	TDD → BDD → ITL	100	100	100	5,33	1,61	2,11	12	12	12	2	2	2
S19	M	ITL → BDD → TDD	31,58	100	100	0,54	4,02	2,00	17	17	8	1	2	2
S20	F	BDD → TDD → ITL	100	100	100	5,08	3,41	2,31	21	21	21	1	1	1
S21	M	ITL → TDD → BDD	94,74	100	100	2,29	2,88	2,93	18	18	17	2	2	2
S22	M	ITL → TDD → BDD	100	100	100	2,75	4,82	3,57	15	15	15	1	1	2

In FB, regarding the external quality (see Figure 34), the subjects S2, S3, S5, S12 and S19 stand out. The distribution of techniques for S2 was ITL → TDD → BDD, reaching 94.74% for both Incremental Test-Last and Behavior-Driven Development, and 89.47% for Test-Driven Development. The second subject (S3) obtained values of 86.84% for ITL and 100% for BDD and TDD, following the order ITL → BDD → TDD. S5 executed the exercise with the sequence TDD → ITL → BDD, reaching quality ratios of 97.37%, 94.74% and 100%, in the order mentioned. The succession TDD → BDD → ITL, was signaled for S12, which had 73.68% quality in the 3 cases. The last subject (S19) solved the katas following the order ITL → BDD → TDD, reaching values of 31.58% in ITL, and 100% in TDD and BDD.

It could be said that the order of development does not directly influence the quality obtained by the subjects; but when obtaining the result 100% on a given exercise, it allows the subject to reach the same quantity of quality in the following solutions (S3 and S19). In addition, we believe that the level of understanding of the problem affects the results; for instance, it seems that S19 did not understand the problem at first, and S12 presented exactly the same solutions. Additionally, the decision to obtain different solutions for each technique exists, as in the case of S2.

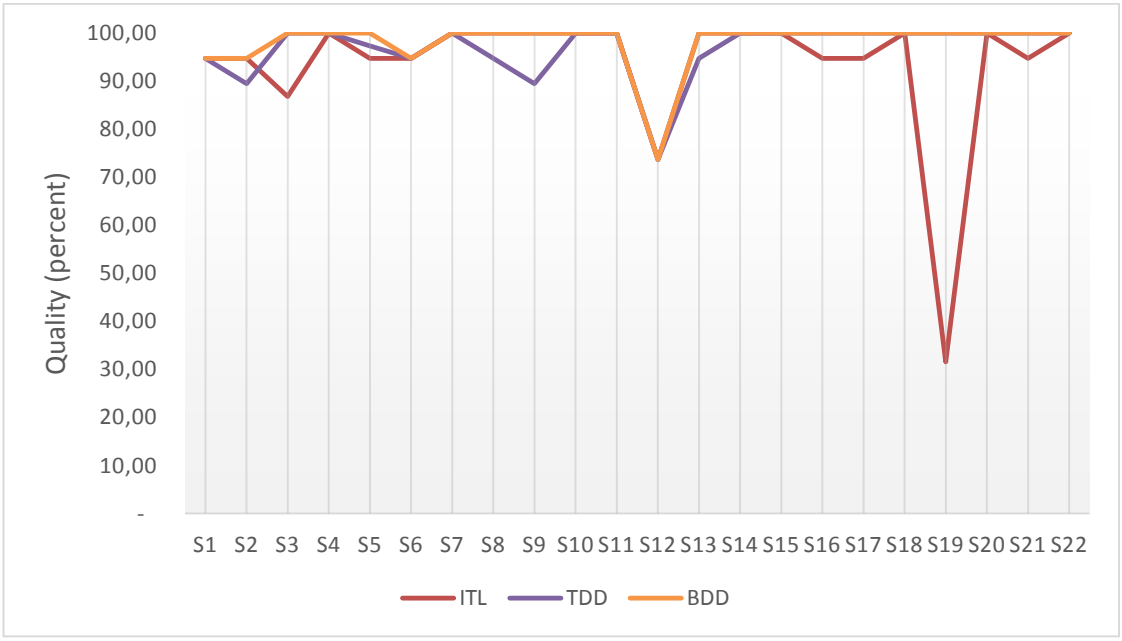


Figure 34. FizzBuzz: external quality

Figure 35, shows the development of FizzBuzz in terms of productivity, in which the subjects stand out: S3, S10, S12 and S18. ITL → BDD → TDD, was the sequence indicated for the first subject (S3); reaching a productivity increase of 4.51% for Test-

Driven Development versus 1.45% for Incremental Test-Last and 1.85% for Behavior-Driven Development. S10 with the execution order BDD → ITL → TDD, obtained values of 3.53% in ITL, 3.84% in TDD and 1.70% in BDD. The third subject (S12) reaches almost the same productivity value for the three techniques (1.02% in ITL, 1.06% in TDD and 1.08% in BDD); despite solving the code kata in the TDD → BDD → ITL order. S18 executed the exercise with the sequence TDD → BDD → ITL, reaching productivity proportions of 1.61%, 2.11% and 5.33%, in the distribution of mentioned techniques.

We believe that the order of the development interferes partially in the productivity per minute of the subjects. People tend to find a better solution each time they repeat the exercise, despite some techniques have more steps than others.

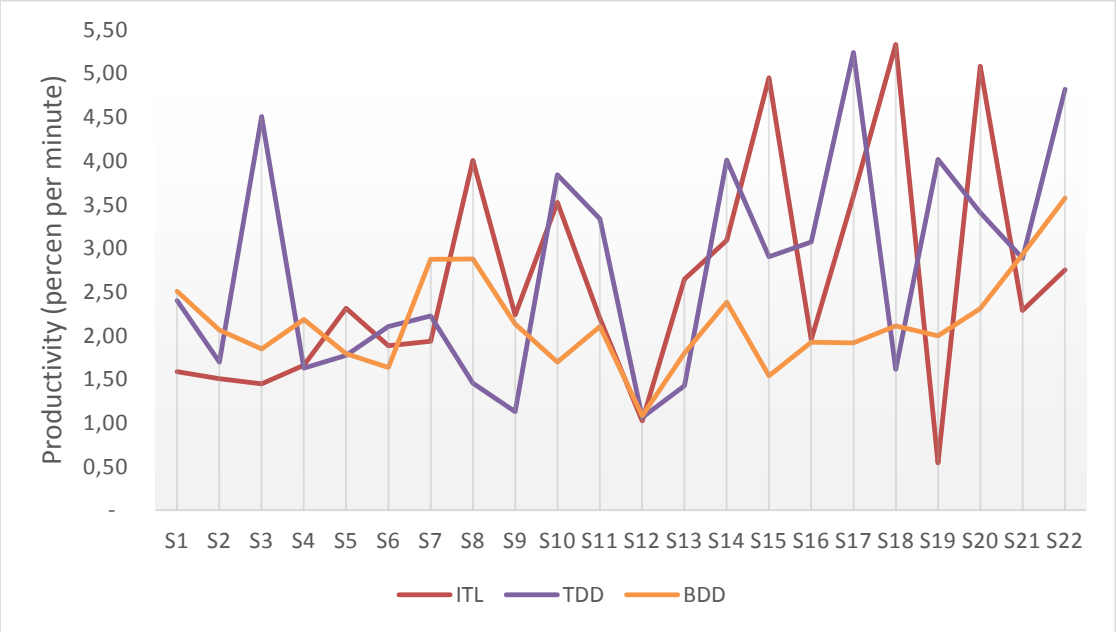


Figure 35. FizzBuzz: productivity per minute

The data extracted from the FizzBuzz code kata on the complexity of code is presented in Figure 36; highlighting the information of subjects S5, S7, S15 and S19. The distribution of the techniques for S5 was TDD → ITL → BDD, getting the complexity of 24 for Incremental Test-Last, 22 for Test-Driven Development, and 23 for Behavior-Driven Development. The second subject (S7) obtained values of 21 for BDD and of 18 for ITL and TDD, following the order TDD → BDD → ITL. S15 executed the exercise with the sequence BDD → TDD → ITL, reaching assessments 16, 17 and 18 of level of complexity in the order mentioned. The succession ITL →

BDD → TDD, was signaled for S19, which had complexity of 17 in ITL and TDD, and 8 in BDD.

Taking into consideration the data obtained, it is observed that the order of resolution does not affect the internal quality of code complexity; it is only dependent on the solution adopted by the developer.

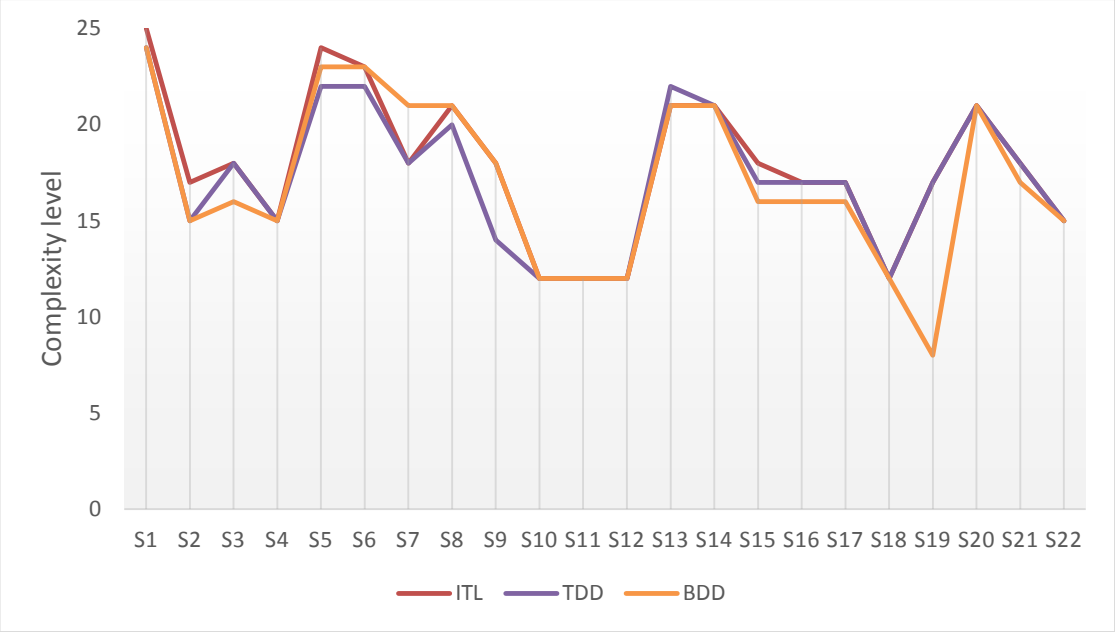


Figure 36. FizzBuzz: code complexity

The code error analysis provided by PMD for FizzBuzz (see Figure 37), highlights the subjects: S1, S3, S4, S11 and S12. ITL → TDD → BDD, it was the sequence indicated for the first subject (S1); reaching an increase of errors of 8 for Test-Driven Development and Behavior-Driven Development, compared to the only error introduced in Incremental Test-Last. S3 with order of execution ITL → TDD → BDD, obtained values of 4 in ITL, 11 in TDD and 1 in BDD. The third subject (S4) achieves very different values for the three techniques (4 in ITL, 5 in TDD and 7 in BDD); despite solving the code kata in the TDD → ITL → BDD way. S11 executed the exercise with the sequence BDD → ITL → TDD, reaching the same number of mistakes; that is, 2 values. The last subject (S12) solved the kata of the form TDD → BDD → ITL, reaching the same errata of 2 in the three cases.

Observing these cases, the order of resolution does not affect the internal quality in number of errors introduced in code; because these depend on the decisions made by the programmers when generating the solution.

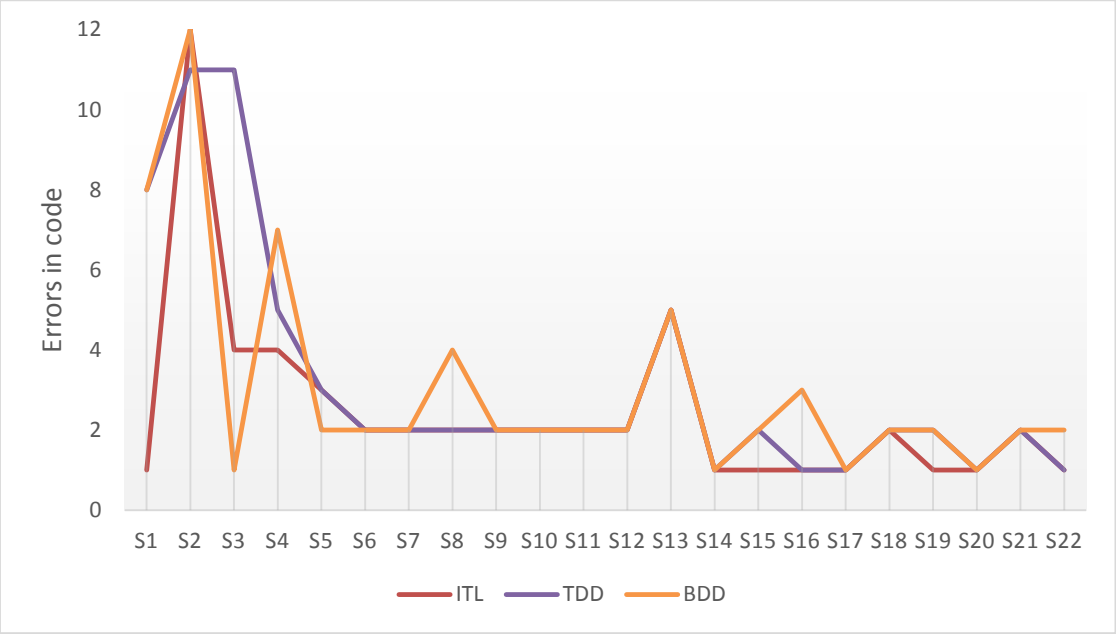


Figure 37. FizzBuzz: number of errors in code

Next, TABLE XII presents the most relevant data of the "String Calculator" exercise development.

TABLE XII. String Calculator metrics' data

Subject	Sex	Resolution Order	External Quality (percent)			Productivity (percent per minute)			McCabe's Cyclomatic complexity			PMD (number of errors in code)		
			ITL	TDD	BDD	ITL	TDD	BDD	ITL	TDD	BDD	ITL	TDD	BDD
S1	M	ITL → TDD → BDD	94,74	94,74	94,74	1,59	2,40	2,51	25	24	24	1	8	8
S2	M	ITL → TDD → BDD	94,74	89,47	94,74	1,51	1,70	2,07	17	15	15	12	11	12
S3	M	ITL → BDD → TDD	86,84	100	100	1,45	4,51	1,85	18	18	16	4	11	1
S4	M	TDD → ITL → BDD	100	100	100	1,66	1,63	2,18	15	15	15	4	5	7
S5	M	TDD → ITL → BDD	94,74	97,37	100	2,31	1,78	1,79	24	22	23	3	3	2
S6	M	ITL → BDD → TDD	94,74	94,74	94,74	1,88	2,10	1,63	23	22	23	2	2	2
S7	F	TDD → BDD → ITL	100	100	100	1,94	2,22	2,87	18	18	21	2	2	2
S8	M	TDD → ITL → BDD	100	94,74	100	4,00	1,45	2,88	21	20	21	2	2	4
S9	M	TDD → ITL → BDD	100	89,47	100	2,24	1,13	2,13	18	14	18	2	2	2
S10	M	BDD → ITL → TDD	100	100	100	3,53	3,84	1,70	12	12	12	2	2	2
S11	M	BDD → ITL → TDD	100	100	100	2,19	3,34	2,10	12	12	12	2	2	2
S12	M	TDD → BDD → ITL	73,68	73,68	73,68	1,02	1,06	1,08	12	12	12	2	2	2
S13	F	TDD → BDD → ITL	100	94,74	100	2,65	1,42	1,80	21	22	21	5	5	5
S14	M	BDD → TDD → ITL	100	100	100	3,09	4,01	2,38	21	21	21	1	1	1
S15	M	BDD → TDD → ITL	100	100	100	4,95	2,90	1,54	18	17	16	1	2	2
S16	M	BDD → ITL → TDD	94,74	100	100	1,95	3,07	1,92	17	17	16	1	1	3
S17	M	BDD → ITL → TDD	94,74	100	100	3,60	5,24	1,92	17	17	16	1	1	1
S18	M	TDD → BDD → ITL	100	100	100	5,33	1,61	2,11	12	12	12	2	2	2
S19	M	ITL → BDD → TDD	31,58	100	100	0,54	4,02	2,00	17	17	8	1	2	2
S20	F	BDD → TDD → ITL	100	100	100	5,08	3,41	2,31	21	21	21	1	1	1
S21	M	ITL → TDD → BDD	94,74	100	100	2,29	2,88	2,93	18	18	17	2	2	2
S22	M	ITL → TDD → BDD	100	100	100	2,75	4,82	3,57	15	15	15	1	1	2

In relation to the external quality of String Calculator (see Figure 38), subjects S2, S10, S15 and S21 stand out. The distribution of the techniques for S2 was BDD → ITL → TDD, reaching 85.42% for both Test-Driven Development and Behavior-Driven Development, and 73.61% for Incremental Test-Last. The second subject (S10) obtained values of 87.5% for ITL and TDD, and 93.75% for BDD; following the order BDD → ITL → TDD. S15 executed the exercise with the sequence ITL → TDD → BDD, reaching quality proportions of 27.08%, 61.11% and 73.26, in the order mentioned. The last subject (S12) solved the kata of the form TDD → BDD → ITL, reaching the same error of 2 in the three cases.

Just like the FizzBuzz analysis, we conclude that the order of development does not directly influence external quality, and these are due to the decisions made by each individual, upon developing their individual solution.

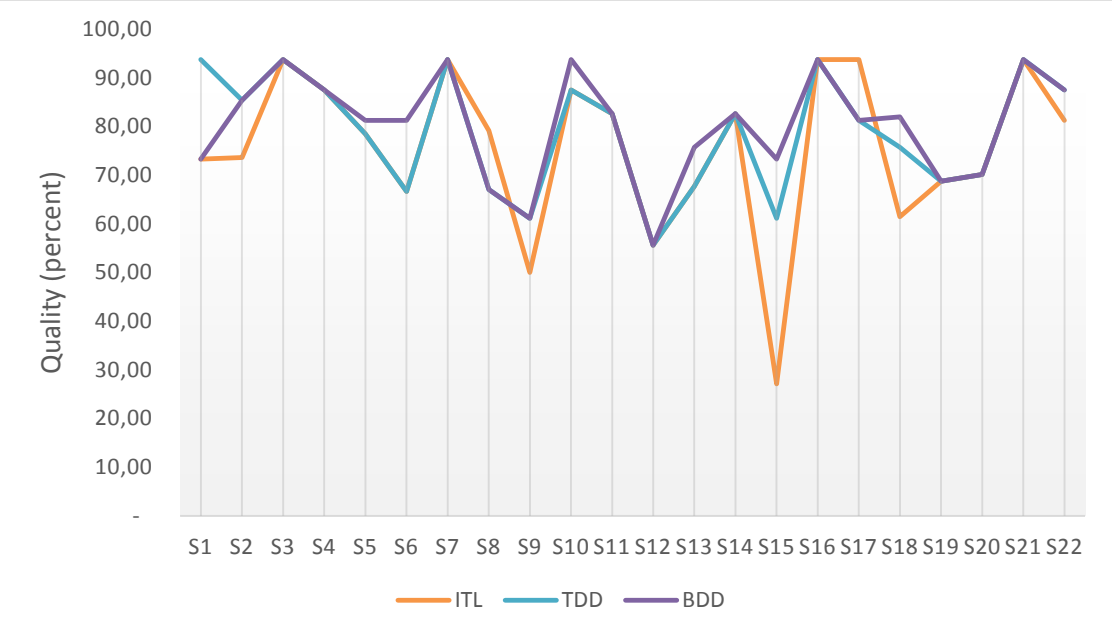


Figure 38. String Calculator: external quality

Figure 39, shows the performance of String Calculator in terms of productivity; highlighting the individuals: S2, S3, S15 and S17. BDD → ITL → TDD, was the sequence indicated for the first subject (S2); reaching a productivity increase of 5.38% for Behavior-Driven Development versus 1.91% for Test-Driven Development and 1.04% for Incremental Test-Last. S3 with order of execution ITL → TDD → BDD, obtained values of 2.7% in ITL, 2.91% in TDD and 2.02% in BDD. The third subject (S15) reaches an improvement in productivity value for the three techniques (0.29% in ITL, 0.89% in TDD and 1.36% in BDD); influencing its distribution of assigned

techniques (ITL → TDD → BDD). S17 executed the exercise with the sequence TDD → ITL → BDD, reaching productivity ratios of 2.56%, 3.88% and 2.56%, respectively.

As well as in the first code kata, it is observed that the order of development interferes partially in the productivity per minute of the subjects, but are mainly related to the decisions taken at the moment of solving the problem.

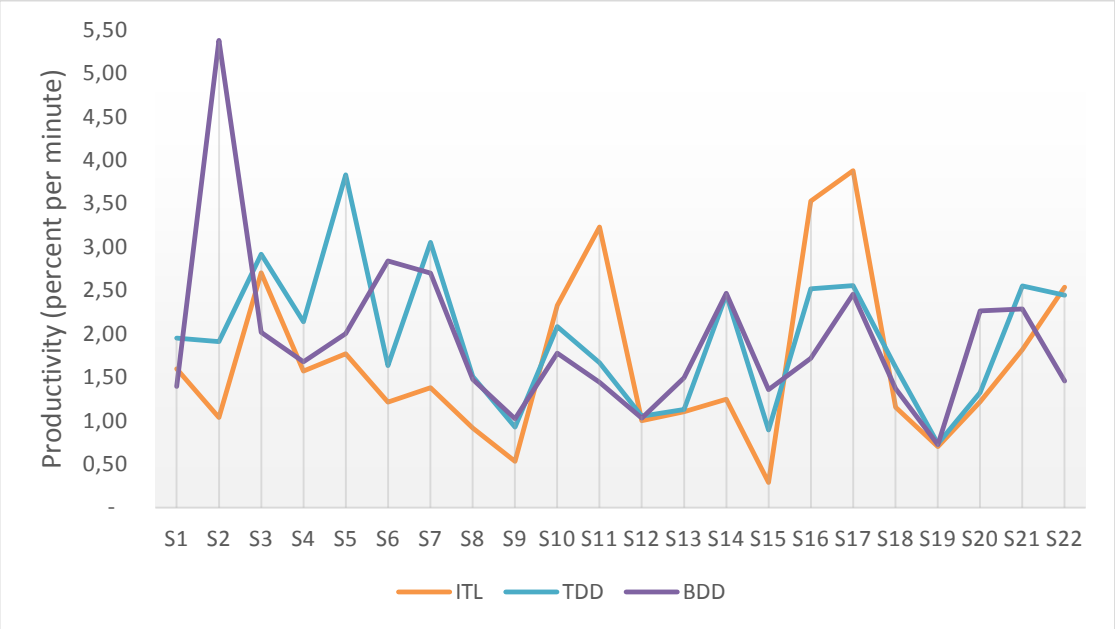


Figure 39. String Calculator: productivity per minute

The information of the subjects: S1, S6, S9 and S10 is highlighted within the extracted data about the code complexity (see Figure 40). The distribution of the techniques for S1 was BDD → TDD → ITL, that reached the complexity of 13 for Incremental Test-Last, 10 for Test-Driven Development, and 14 for Behavior-Driven Development. The second subject (S6) obtained values of 18 for ITL, and of 7 for ITL and TDD; following the order ITL → TDD → BDD. S9 executed the exercise with the sequence ITL → BDD → TDD, reaching ratings 3, 4 and 4 of level of complexity in the order mentioned. The sequence BDD → ITL → TDD, was signaled for S10, which had complexity of 20 in ITL, 21 in TDD, and 18 in BDD.

According to the analyzed data, there is the same similarity between the 2 code katas; noting that the order of resolution does not affect the internal quality in code complexity. The level of complexity is dependent on the solution adopted by the developer.

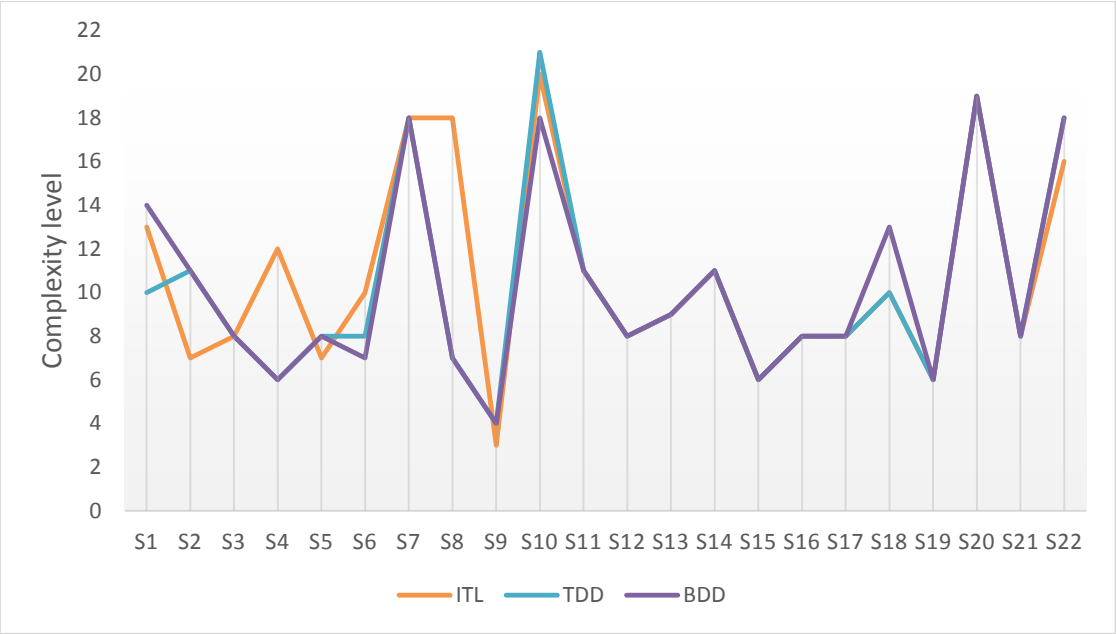


Figure 40. String Calculator: code complexity

The code error analysis provided by PMD for the String Calculator (see Figure 41), highlights the subjects: S1, S4, S7 and S9. The sequence indicated for the first subject (S1), BDD → TDD → ITL; allowed him to create solutions that have 3 errors in Incremental Test-Last, 2 errors in Test-Driven Development and 4 errors in Behavior-Driven Development. S4, with the execution order BDD → TDD → ITL, obtained values of 2 in TDD and BDD, and 4 in ITL. The third subject (S7), executed the exercise with the sequence TDD → BDD → ITL, reaching the same number of mistakes for ITL and BDD (6 values) and a decrease of errors in TDD (3 values). The last subject (S9) solved the kata of the form ITL → BDD → TDD, reaching erring only once in ITL.

Examining the data obtained and, like the first code kata, it is concluded that the order of resolution does not affect the internal quality in number of errors entered in code; because they depend mainly on the decisions made by the programmers when generating the solution.

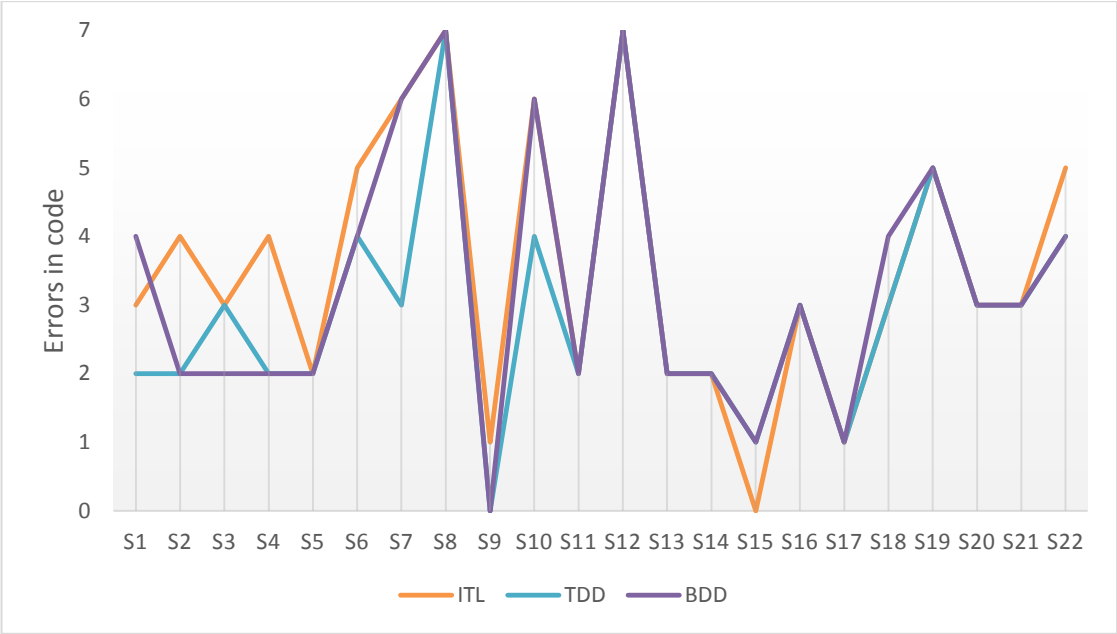


Figure 41. String Calculator: number of errors in code

5.4. Hypothesis evaluation

The analysis of errors written in the code by PMD (see TABLE XI and TABLE XII), displays diverse results due to the lack of experience that the subjects have when developing software; that is, they make mistakes involuntarily that can be considered minor newbie-programmer mistakes. Having noticed this, this will not be taken into consideration when responding to the planned hypothesis.

5.4.1. External Quality

There is improvement in external quality when the percentage provided by one technique is greater than the presented by another. In Figure 42 it can be seen that when applying TDD or BDD, a greater benefit is obtained compared to ITL. This allows to answer the questions RQ1, RQ4 and RQ7; discarding the incorrect hypotheses. For RQ1, hypotheses H_1^1 is accepted; because both in the code kata FizzBuzz (FB) and String Calculator (SC), TDD provides an improvement against ITL of 3.35% and 3.33% respectively. In relation to RQ4, H_1^4 is valid; because in both exercises there is an increase of 4.67% compared to ITL. Responding to RQ7, H_1^7 is admitted; since the BDD quality increase in FB is 1.32% and in SC it is 1.34%, against TDD.

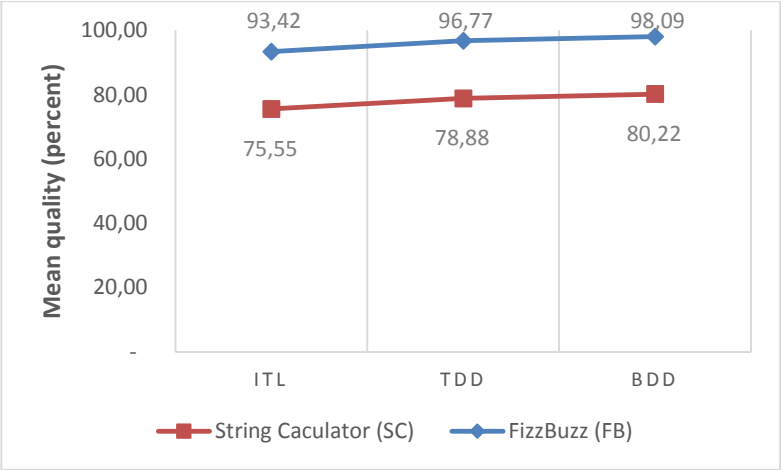


Figure 42. External quality response variable

5.4.2. Productivity

There is improvement in the productivity per minute of the developer when the percentage presented by one technique is higher when compared to the value provided by another. Figure 43 shows the increase or decrease in the productivity percentage when applying TDD or BDD to ITL. This allows responding to the questions RQ3, RQ6 and RQ9; discarding the incorrect hypotheses. For RQ3, hypotheses H_1^3 is accepted; because in the development of the code katas FizzBuzz (FB) and String Calculator (SC), TDD provides an improvement against ITL of 0.14% and 0.28% respectively. In relation to RQ6, H_1^6 s valid for SC, with an increase of 0.25% of BDD versus ITL; on the contrary, H_2^6 is appropriate for FB due to the decreasing of -0.47% in BDD versus ITL. Responding to RQ9, H_2^9 is admitted; because the productivity reduction of BDD in FB is -0.60% and in SC it is -0.03% against TDD.

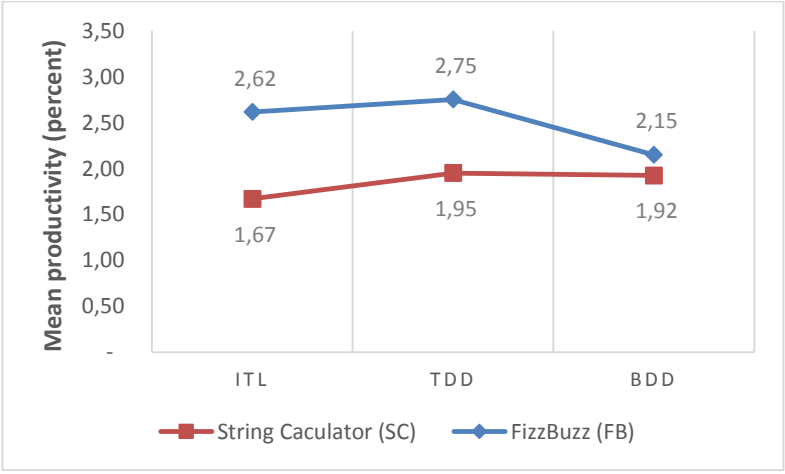


Figure 43. Productivity response variable

5.4.3. Internal Quality

There is improvement in the complexity of a program when the value presented by one technique is lower when compared to the value provided by another technique. Figure 44 shows the increase or decrease in the value of the McCabe's cyclomatic complexity in the application of ITL, TDD or BDD. This allows to answer the questions RQ2, RQ5 and RQ8; discarding the incorrect hypotheses. For RQ2, hypotheses H_1^2 is accepted; because in the codes kata FizzBuzz (FB) and String Calculator (SC), TDD provides an improvement against ITL of 0.50 and 0.59 respectively. Responding to RQ5, H_2^9 is admitted; since the level of complexity in BDD decreases a value of 0.77 in FB and 0.45 in SC, when compared with ITL. In relation to RQ8, H_2^8 is valid for SC with an increase of -0.14 of BDD versus TDD; on the contrary H_1^8 is appropriate for FB due to the decrease of 0.24 of BDD versus TDD.

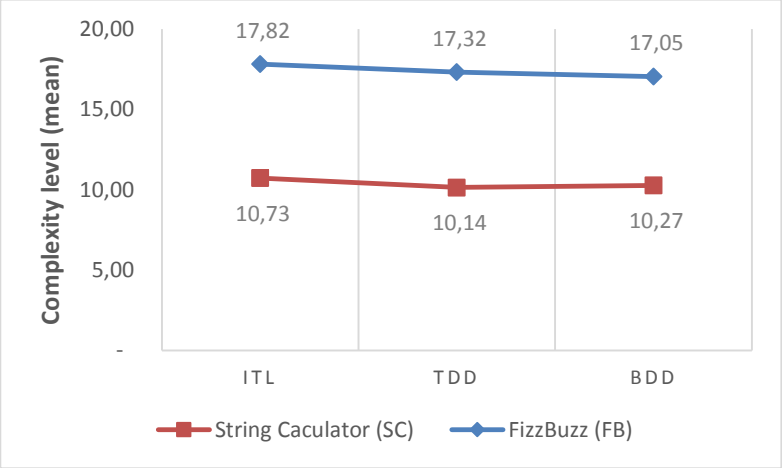


Figure 44. Program dificulty response variable

6. Conclusions and future work

This work allowed to validate the influence produced by the application of agile techniques: Test-Driven Development, Acceptance Test-Driven Development and Behavior-Driven Development, into the software life cycle; which provide an extended vision of what is going to be developed and guarantee a lower number of failures.

Both in conventional models and in agile, testing plays an important role in terms of software quality; because they help in the early detection of errors about requirements, design and functionality.

Test-based development techniques (TDD, ATDD and BDD) are distinct from traditional development, and provide benefits such as: improved collaboration and communication among stakeholders, the design is oriented to the client's needs, increase quality in the developed code, they reduce the number of errors, increment the productivity (shorter debugging time), and provide more flexibility in requirements changing; ensuring the quality of the product developed.

When comparing the three techniques it can be said that TDD is focused at the unit level (function, class, module, component, among others); ATDD works on the creation of acceptance criteria (user stories), ensuring the construction of the client's needs; and BDD is responsible for exploring, discovering and developing the desired behavior of software based on a common language where all stakeholders can understand. ATDD and BDD are distinct terminologies to stress some differences in the approach that leads to similar outcomes.

The wide variety of tools dedicated to testing that exist in the market allow the execution of these techniques with ease in any development environment and with any programming language.

The application of an intensive training on the techniques of software development based on tests (TDD and BDD), played an important role in the development of the experiment; providing greater compression of them by the subjects, and allowing the effects obtained from internal quality, external quality and productivity to be more evident.

The results of the experiment showed statistically significant differences in favor of TDD and BDD, validating the benefits that these agile techniques produce regarding the quality of the software (external and internal) and the productivity of the developers. However, a decrease in BDD productivity could be noticed in the resolution of the first code kata; maybe it happened because of the lack of software development experience that the programmers presented and / or the need for more training for this technique.

When comparing the information of the software development techniques based on tests, obtained from the experimentation, it can be said that BDD improves the external and internal quality. The use of Gherkin is probably the main factor so that occurs, because of this language allows a greater understanding of the requirements to be implemented. In addition, TDD presented better productivity values than BDD, because it presents fewer steps in its process probably.

The work carried out allowed to complete a publication in the Thirteenth International Conference on Software Engineering Advances, ICSEA 2018. Which relates the experiment carried out in order to know the impact produced by the development techniques based on tests.

Regarding the future work, the study could benefit in the following ways:

- Performing a replication of the experiment in another country, in order to analyze whether cultural issues have impact on the results;
- Apply the study in the industrial environment, in order to know if similar results are obtained. In this case it could be taken other exercises with a greater degree of difficulty so that they can be solved by the developers.

References

[1] Association of Modern Technologies Professionals, “Software Development Methodologies.” [Online]. Available: http://www.itinfo.am/eng/software-development-methodologies/. [Accessed: 04-Dec-2017].

[2] VersionOne Inc., “What is Agile? Learn About Agile Software Development.” [Online]. Available: https://www.versionone.com/agile-101/. [Accessed: 04-Dec-2017].

[3] J. Shore and S. Warden, *The Art of Agile Development*. O’Reilly Media, Inc, 2008.

[4] S. R. Meier, “Technology Portfolio Management for Project Managers 28 Feb 17 PMI Tysons Corner Chapter.”

[5] D. Graham, E. Van Veenendaal, I. Evans, and R. Black, *Foundations of software testing*, 2nd Editio. 2012.

[6] P. Consultores, “Introducción a las Pruebas de Software.” p. 11, 2012.

[7] B. B. Agarwal, S. P. Tayal, and M. (Mahesh) Gupta, *Software engineering and testing: An introduction*. Jones and Bartlett, 2010.

[8] R. Black and G. Rueda Sandoval, *Fundamentos de prueba de software*. RBCS, 2011.

[9] TechTarget, “What is white box (white box testing)? - Definition.” [Online]. Available: http://searchsoftwarequality.techtarget.com/definition/white-box. [Accessed: 10-Nov-2017].

[10] TechTarget, “What is black box (black box testing)? - Definition.” [Online]. Available: http://searchsoftwarequality.techtarget.com/definition/black-box. [Accessed: 10-Nov-2017].

[11] TechTarget, “What is gray box testing (gray box)? - Definition.” [Online]. Available: http://searchsoftwarequality.techtarget.com/definition/gray-box. [Accessed: 10-Nov-2017].

[12] R. M. Sharma, “Quantitative Analysis of Automation and Manual Testing,” *Int. J. Eng. Innov. Technol.*, vol. 4, no. 1, pp. 252–257, 2014.

[13] Agile Aliance, “Agile 101.” [Online]. Available: https://www.agilealliance.org/agile101/. [Accessed: 04-Dec-2017].

[14] VersionOne, “The 11th Annual State of Agile Report,” *Annual State of Agile*

Report, p. 17, 2017.

[15] A. Manfield, *Real Scrum and More*, Ilustrada. LULU Press, 2015.

[16] S. Millett, J. Blankenship, and M. Bussa, *Pro Agile .NET Development with SCRUM*. Apress, 2011.

[17] Agile Aliance, "The Agile Manifesto." [Online]. Available: https://www.agilealliance.org/agile101/the-agile-manifesto/.

[18] SmartSheet, "Agile Project Management 101: A Beginner´s Guide for Non-Project Managers," 2017.

[19] M. S. Palmquist, M. A. Lapham, S. Miller, T. Chick, and I. Ozkaya, "Parallel Worlds: Agile and Waterfall Differences and Similarities," *SEI, Carnegie Mellon Univ.*, no. October, pp. 1–101, 2013.

[20] C. Blé Jurado, *Diseño Ágil con TDD*. Lulu.com, 2010.

[21] M. Lotz, "Waterfall vs. Agile: Which Methodology is Right for Your Project?" [Online]. Available: https://www.seguetech.com/waterfall-vs-agile-methodology/. [Accessed: 02-Jul-2018].

[22] G. Ramakrishnan, "Introduction to Agile Software Development." [Online]. Available: https://www.slideshare.net/gramakri/introduction-to-agile-software-development-55679400. [Accessed: 02-Jul-2018].

[23] D. Morelos, "Traditional vs. Agile Software Development Method: Which One is Right for Your Project? - DZone Agile." [Online]. Available: https://dzone.com/articles/traditional-vs-agile-software-development-method-w. [Accessed: 02-Jul-2018].

[24] Eternal Sunshine, "From Traditional to Agile." [Online]. Available: https://eternalsunshineoftheismind.wordpress.com/2013/03/06/from-traditional-to-agile/. [Accessed: 02-Jul-2018].

[25] Agile Aliance, "What is Extreme Programming (XP)?" [Online]. Available: https://www.agilealliance.org/glossary/xp/. [Accessed: 08-Dec-2017].

[26] Agile Business Consortium, "What is DSDM." [Online]. Available: https://www.agilebusiness.org/what-is-dsdm. [Accessed: 08-Dec-2017].

[27] Agile Alliance, "What is a Kanban?" [Online]. Available: https://www.agilealliance.org/glossary/kanban. [Accessed: 08-Dec-2017].

[28] Agile Alliance, "What is Scrum?" [Online]. Available: https://www.agilealliance.org/glossary/scrum. [Accessed: 08-Dec-2017].

[29] Agile Alliance, "What is Test Driven Development (TDD)?" [Online].

Available: https://www.agilealliance.org/glossary/tdd/. [Accessed: 11-Nov-2017].

[30] R. Martinez, "TDD, una metodología para gobernarlos a todos," 2017. [Online]. Available: https://www.paradigmadigital.com/techbiz/tdd-una-metodologia-gobernarlos-todos/. [Accessed: 11-Nov-2017].

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

[32] G. Mclean Hall, *Adaptive Code via C#: Agile coding with design patterns and SOLID principles*. 2014.

[33] O. Dieste, E. R. Fonseca, G. Raura, and P. Rodríguez, "Efectividad del Test-Driven Development: Un Experimento Replicado," *Rev. Latinoam. Ing. Softw.*, vol. 3, no. 3, p. 141, 2015.

[34] TechTarget, "What is JUnit? - Definition." [Online]. Available: http://searchsoftwarequality.techtarget.com/definition/JUnit. [Accessed: 14-Nov-2017].

[35]. NET Foundation, "xUnit.net." [Online]. Available: https://xunit.github.io/. [Accessed: 14-Nov-2017].

[36] Typemock, "VB Unit Testing Tool." [Online]. Available: https://www.typemock.com/vb-unit-testing-tool. [Accessed: 14-Nov-2017].

[37] CUnit Project, "CUnit Home." [Online]. Available: http://cunit.sourceforge.net/. [Accessed: 14-Nov-2017].

[38] Gargoyle Software Inc., "HtmlUnit – Welcome to HtmlUnit." [Online]. Available: http://htmlunit.sourceforge.net/. [Accessed: 14-Nov-2017].

[39] Agile Alliance, "Acceptance Test Driven Development (ATDD)." [Online]. Available: https://www.agilealliance.org/glossary/atdd. [Accessed: 12-Nov-2017].

[40] Agile Alliance, "What are the Three Amigos in Agile?" [Online]. Available: https://www.agilealliance.org/glossary/three-amigos/. [Accessed: 12-Nov-2017].

[41] L. Koskela, *Test driven : practical TDD and acceptance TDD for Java developers*. Manning, 2008.

[42] R. Kumar, "Acceptance Test Driven Development." [Online]. Available: https://dzone.com/articles/acceptance-test-driven-development-closing-the-com. [Accessed: 29-Nov-2017].

[43] K. Pugh, *Lean-agile acceptance test-driven development better software*

through collaboration. .

[44] R. Quartel, “Acceptance Test-Driven Development: Are We Flogging a Dead Horse?,” 2013. [Online]. Available: https://www.solutionsiq.com/resource/blog-post/acceptance-test-driven-development-are-we-flogging-a-dead-horse/. [Accessed: 13-Nov-2017].

[45] Selenium Project, “Introduction — Selenium Documentation.” [Online]. Available: http://www.seleniumhq.org/docs/01_introducing_selenium.jsp. [Accessed: 14-Nov-2017].

[46] FitNesse.org, “FitNesse.UserGuide.” [Online]. Available: http://fitnesse.org/FitNesse.UserGuide. [Accessed: 14-Nov-2017].

[47] concordion.org, “Specification by Example | Concordion.” [Online]. Available: http://concordion.org/. [Accessed: 14-Nov-2017].

[48] robotframework.org, “Robot Framework.” [Online]. Available: http://robotframework.org/. [Accessed: 14-Nov-2017].

[49] J. F. Smart, *BDD In Action: Behavior Driven Development for the Whole Software Lifecycle*. Manning, 2014.

[50] Agile Alliance, “BDD: Learn about Behavior Driven Development.” [Online]. Available: https://www.agilealliance.org/glossary/bdd/. [Accessed: 16-Nov-2017].

[51] TechTarget, “What is behavior-driven development (BDD)? - Definition.” [Online]. Available: http://searchsoftwarequality.techtarget.com/definition/Behavior-driven-development-BDD. [Accessed: 16-Nov-2017].

[52] Enrique Amodeo, *Learning Behavior-driven Development with JavaScript*. Packt Publishing, 2015.

[53] Cucumber Limited, “Cucumber.” [Online]. Available: https://cucumber.io/. [Accessed: 17-Nov-2017].

[54] SpecFlow Organization, “SpecFlow - Binding Business Requirements to .NET Code.” [Online]. Available: http://specflow.org/. [Accessed: 17-Nov-2017].

[55] Behat Community, “Behat documentation.” [Online]. Available: http://behat.org/en/latest/guides.html. [Accessed: 17-Nov-2017].

[56] JBehave Community, “JBehave.” [Online]. Available: http://jbehave.org/. [Accessed: 17-Nov-2017].

[57] RSpec Community, “RSpec: Behaviour Driven Development for Ruby.”

[Online]. Available: http://rspec.info/. [Accessed: 17-Nov-2017].

[58] P. A. Vaca, C. Maldonado, C. Inchaurrondo, J. Peretti, M. S. Romero, and M. Bueno, "Test-Driven Development -Una aproximación para entender su utilidad en el proceso de desarrollo de Software," *Univ. Tecnológica Nac. Fac. Reg. Córdoba*, vol. 1–10, 2015.

[59] M. Gärtner, *ATDD by example: A practical guide to acceptance test-driven development*. 2012.

[60] D. North, "Introducing BDD | Dan North & Associates," 2006. [Online]. Available: https://dannorth.net/introducing-bdd/. [Accessed: 07-Dec-2017].

[61] S. Lakshmanan, "Test Approach and Comparisons between ATDD TDD and BDD." [Online]. Available: http://toolsqa.com/blogs/test-approach-and-comparisons-between-atdd-tdd-and-bdd/. [Accessed: 07-Dec-2017].

[62] K. Sureka, "Agile Development – Difference between TDD/ATDD/BDD." [Online]. Available: https://www.linkedin.com/pulse/agile-development-difference-between-tddatddbdd-komal-sureka. [Accessed: 08-Dec-2017].

[63] M. Kumar, "What is TDD, BDD & ATDD? – Assert Selenium." [Online]. Available: http://www.assertselenium.com/atdd/difference-between-tdd-bdd-atdd/. [Accessed: 08-Dec-2017].

[64] U. Eriksson, "What are TDD and ATDD? | ReQtest." [Online]. Available: https://reqtest.com/testing-blog/tdd-and-atdd-an-overview-of-the-two-popular-methods-2/. [Accessed: 09-Dec-2017].

[65] R. Wilcox, "BDD vs TDD: A Behavior-Driven Development Example | Toptal." [Online]. Available: https://www.toptal.com/freelance/your-boss-won-t-appreciate-tdd-try-bdd. [Accessed: 09-Dec-2017].

[66] T. A. Gamage, "Behavior Driven Development (BDD) & Software Testing in Agile Environments." [Online]. Available: https://medium.com/agile-vision/behavior-driven-development-bdd-software-testing-in-agile-environments-d5327c0f9e2d. [Accessed: 10-Dec-2017].

[67] Shilpa, "How The Testers Are Involved In TDD, BDD and ATDD Techniques — Software Testing Help." [Online]. Available: http://www.softwaretestinghelp.com/testers-in-tdd-bdd-atdd-techniques/. [Accessed: 05-Jan-2018].

[68] L. Keogh, "BDD: ATDD done well?" [Online]. Available: https://www.infoq.com/news/2011/02/BDD-ATDD. [Accessed: 05-Jan-2018].

[69] K. S. Khan, R. Kunz, J. Kleijnen, and G. Antes, “Five steps to conducting a systematic review,” *Jrsm*, vol. 96, no. 3, pp. 118–121, 2003.

[70] Venkat Moncompu, “Agile test automation: transition challenges and ways to overcome them,” *Pacific NW Softw. Qual. Conf.*, no. December, pp. 1–9, 2013.

[71] D. S. Janzen, “On the Influence of Test-Driven Development on Software Design,” pp. 0–7, 2006.

[72] D. Thomas, “CodeKata.” [Online]. Available: http://codekata.com/. [Accessed: 16-Jan-2018].

[73] H. Munir, K. Wnuk, K. Petersen, and M. Moayyed, “An experimental evaluation of test driven development vs. test-last development with industry professionals,” *Proc. 18th Int. Conf. Eval. Assess. Softw. Eng. - EASE ’14*, pp. 1–10, 2014.

[74] T. J. McCabe, “A Complexity Measure,” *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, 1976.

[75] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo, “A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?,” *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 597–614, 2017.

[76] P. A. Vaca *et al.*, “Test-Driven Development - Beneficios y Desafíos para el Desarrollo de Software .,” pp. 232–239, 2014.

[77] S. Hammond and D. Umphress, “Test driven development,” *Proc. 50th Annu. Southeast Reg. Conf. - ACM-SE ’12*, no. March 2012, p. 158, 2012.

[78] A. Okolnychyi and Konrad Fögen, “A Study of Tools for Behavior-Driven Development,” *Full-scale Softw. Eng. Trends Release Eng.*, no. 7, pp. 7–12, 2016.

[79] R. A. De Carvalho, F. Luiz, D. Carvalho, R. S. Manhães, and G. L. De Oliveira, “Implementing Behavior Driven Development in an Open Source ERP,” pp. 242–249, 2013.

[80] L. F. S. Hoffmann, L. E. G. de Vasconcelos, E. Lamas, A. M. da Cunha, and L. A. V. Dias, “Applying Acceptance Test Driven Development to a Problem Based Learning Academic Real-Time System,” *2014 11th Int. Conf. Inf. Technol. New Gener.*, pp. 3–8, 2014.

[81] M. Rahman and J. Gao, “A reusable automated acceptance testing architecture for microservices in behavior-driven development,” *Proc. - 9th IEEE Int. Symp. Serv. Syst. Eng. IEEE SOSE 2015*, vol. 30, pp. 321–325, 2015.

[82] E. C. S. Santos, D. M. Beder, and R. A. D. Penteado, “A Study of Test Techniques for Integration with Domain Driven Design,” *Proc. - 12th Int. Conf. Inf. Technol. New Gener. ITNG 2015*, no. 20, pp. 373–378, 2015.

[83] M. Wynne and A. Hellesoy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*, Pragmatic. 2012.

[84] Agile Katas, “Rock Paper Scissors Kata.” [Online]. Available: http://agilekatas.co.uk/katas/RockPaperScissors-Kata. [Accessed: 16-Feb-2018].

[85] Agile Katas, “Roman Numerals Kata.” [Online]. Available: http://agilekatas.co.uk/katas/RomanNumerals-Kata. [Accessed: 16-Feb-2018].

[86] Coding Dojo, “Kata Numbers In Words.” [Online]. Available: http://codingdojo.org/kata/NumbersInWords/. [Accessed: 01-Feb-2018].

[87] M. Whelan, “FizzBuzzWhiz Kata.” [Online]. Available: https://github.com/mwhelan/Katas/tree/master/Katas.FizzBuzzWhiz. [Accessed: 16-Feb-2018].

[88] Agile Katas, “Fizz Buzz Kata.” [Online]. Available: http://agilekatas.co.uk/katas/FizzBuzz-Kata. [Accessed: 16-Feb-2018].

[89] R. Osherove, “TDD Kata 1 - String Calculator.” [Online]. Available: http://osherove.com/tdd-kata-1/. [Accessed: 16-Feb-2018].

[90] Ifpug, “Function Point Counting Practices Manual,” *Group*, vol. on06/23/. 2010.

[91] F. Sánchez, “Medida del tamaño funcional de aplicaciones software,” *Univ. Castilla-La Mancha*, 1999.

[92] PMD Open Source Project, “PMD Source Code Analyzer.” [Online]. Available: https://pmd.github.io/. [Accessed: 19-Feb-2018].

[93] D. J. Saville and G. R. Wood, *Statistical Methods: The Geometric Approach*. Springer New York, 1991.

[94] R. Y. (Richard Y. Chin and B. Y. Lee, *Principles and practice of clinical trial medicine*. Elsevier/Academic Press, 2008.

[95] N. Hernández, “Diseño de Investigación.” [Online]. Available: http://www.aniorte-nic.net/apunt_metod_investigac4_5.htm. [Accessed: 01-Feb-2018].

[96] PMD Open Source Project, “Error Prone | PMD Source Code Analyzer.” [Online]. Available: https://pmd.github.io/pmd-

6.0.0/pmd_rules_java_errorprone.html. [Accessed: 04-Jun-2018].

[97] ISTQB, “ISTQB Glossary.” [Online]. Available: http://glossary.istqb.org/. [Accessed: 24-Oct-2017].

[98] Agile Alliance, “What is Refactoring?” [Online]. Available: https://www.agilealliance.org/glossary/refactoring. [Accessed: 06-Jan-2018].

[99] Techopedia, “What is Software Framework?” [Online]. Available: https://www.techopedia.com/definition/14384/software-framework. [Accessed: 06-Jan-2018].

[100] Techopedia, “What is a Programming Language?” [Online]. Available: https://www.techopedia.com/definition/24815/programming-language. [Accessed: 06-Jan-2018].

[101] TeachTarget, “What is paradigm?” [Online]. Available: http://whatis.techtarget.com/definition/paradigm. [Accessed: 06-Jan-2018].

[102] Scrum Org., “Scrum Glossary.” [Online]. Available: https://www.scrum.org/resources/scrum-glossary. [Accessed: 06-Jan-2018].

[103] Techopedia, “What is a Developer?” [Online]. Available: https://www.techopedia.com/definition/17095/developer. [Accessed: 06-Jan-2018].

[104] Techopedia, “What is a Software Architect?” [Online]. Available: https://www.techopedia.com/definition/31441/software-architect. [Accessed: 06-Jan-2018].

[105] Techopedia, “What is Business Systems Analyst?” [Online]. Available: https://www.techopedia.com/definition/18120/business-systems-analyst. [Accessed: 06-Jan-2018].

[106] Techopedia, “What is a Systems Analyst?” [Online]. Available: https://www.techopedia.com/definition/4816/systems-analyst. [Accessed: 06-Jan-2018].

This page was intentionally left blank

Appendix

Appendix A: Paper

(see appxA_WIP_ExperimentalEvaluation.pdf)

An Experimental Evaluation of ITL, TDD and BDD

Abstract— Agile development embodies a distancing from traditional approaches, allowing an iterative development that easily adapts and proposes solutions to changing requirements of the clients. For this reason, the industry has recently adopted the use of its practices and techniques, e.g., Test-Driven Development (TDD), Behavior-Driven Development (BDD), amongst others. These techniques promise to improve the software quality and the productivity of the programmers; therefore, several experiments, especially regarding TDD, have been carried out within the academy and in industry. These show variant results (some of them with positive effects and others not so much). The main goal of this work is to verify the impact made by the TDD and BDD techniques in software development, analyzing their main promises regarding quality and productivity. We aim to conduct the experience in the academy, with a group of students from the Systems Engineering Degree of the Universidad Técnica del Norte, Ecuador. The students will receive training and appropriate education to improve knowledge about it, and we aspire to achieve interesting results concerning both quality and productivity. The challenge that it is also desirable, is to replicate the experiment in the industry or other adequate contexts.

Keywords—Empirical research; ITL; TDD; BDD; Software Engineering; productivity; code quality; Incremental Test-Last; Test-Driven Development; Behavior-Driven Development.

I. INTRODUCTION

In software development, quality is probably the most important aspect [1]. The industry in this area is well aware of this, because users prefer products that provide a satisfying and productive experience. However, this kind of products are difficult to build. To do this, teams make use of software development methodologies such as: traditional or agile; that allow to plan and control the process of creating a software [2]. Agile methods have been very popular in the industry [1]; because in contrast to traditional methods, the first use an iterative approach that propose to respond quickly to the changing needs of the client [2][3], improve the quality and increase the productivity of programmers [4]. A question arises: Do agile practices (such as Test-Driven Development (TDD) or Behavior-Driven Development (BDD)) help increase product quality and developer productivity? In this context, we intend to run a workshop and a controlled experiment that will answer that question.

The document is structured as follows: Section II introduces software testing and the techniques used in the experiment, Section III provides a summary of the related work, Section IV defines the goals, Section V contains the design of the study. Finally, the expected results and the conclusion and future work are presented in Section VI and VII, respectively.

Appendix B: Function Point Analysis

An example of the function point analysis with the "Roman Numerals" and "String Calculator" code katas is present bellow. For the analysis of all code katas, please see *appxB_FunctionPoint.xlsx.*

A) Roman Numerals code kata function point analysis

Puntos de Función Sin Ajustar										
Descripción	Complejidad									Total
	Baja			Media			Alta			
Entradas (EI)	2	X	3	0	X	4	0	X	6	6
Salidas (EO)	2	X	4	0	X	5	0	X	7	8
Consultas (EQ)	0	X	3	0	X	4	0	X	6	0
Archivos Lógicos (ILF)	1	X	7	1	X	10	0	X	15	17
Interfaces de Programa (EIF)	0	X	5	0	X	7	0	X	10	0
PFSA										31

Figure 45. Roman Numerals unadjusted FPs

Puntos de Función Ajustados (PF)		
N° de Factor	Factor de Ajuste	Valor 0....5
1	Comunicación de Datos	0
2	Proceso Distribuido	0
3	Objetivos de Rendimiento	0
4	Configuracion de Explotacion Compartida	0
5	Tasas de Transacciones	0
6	Entrada de Datos en Línea	0
7	Eficiencia con el usuario Final	2
8	Actualizaciones en Línea	0
9	Lógica de Proceso Interno Compleja	3
10	Reusabilidad de Codigo	4
11	Conversion de Instalacion Contempladas	0
12	Facilidad de Operación	0
13	Instalaciones Multiples	0
14	Facilidad de Cambios	4
Ajuste de Complejidad Técnica (ACT)		13
Complejidad del Proceso Ajustada (ACTA)		0,78
Total de Puntos de Fucion Ajustados (PFA)		24,18

Figure 46. Roman Numerals adjusted FPs

B) String Calculator code kata function point analysis

Puntos de Función Sin Ajustar										
Descripción	Complejidad									Total
	Baja			Media			Alta			
Entradas (EI)	1	x	3	0	x	4	0	x	6	3
Salidas (EO)	2	x	4	0	x	5	0	x	7	8
Consultas (EQ)	0	x	3	0	x	4	0	x	6	0
Archivos Lógicos (ILF)	0	x	7	2	x	10	0	x	15	20
Interfaces de Programa (EIF)	0	x	5	0	x	7	0	x	10	0
PFSA										31

Figure 47. String Calculator unadjusted FPs

Puntos de Función Ajustados (PF)		
N° de Factor	Factor de Ajuste	Valor 0....5
1	Comunicación de Datos	0
2	Proceso Distribuido	0
3	Objetivos de Rendimiento	0
4	Configuracion de Explotacion Compartida	0
5	Tasas de Transacciones	0
6	Entrada de Datos en Línea	0
7	Eficiencia con el usuario Final	2
8	Actualizaciones en Línea	0
9	Lógica de Proceso Interno Compleja	4
10	Reusabilidad de Codigo	4
11	Conversion de Instalacion Contempladas	0
12	Facilidad de Operación	0
13	Instalaciones Multiples	0
14	Facilidad de Cambios	4
Ajuste de Complejidad Técnica (ACT)		14
Complejidad del Proceso Ajustada (ACTA)		0,79
Total de Puntos de Fucion Ajustados (PFA)		24,49

Figure 48. String Calculator adjusted FPs

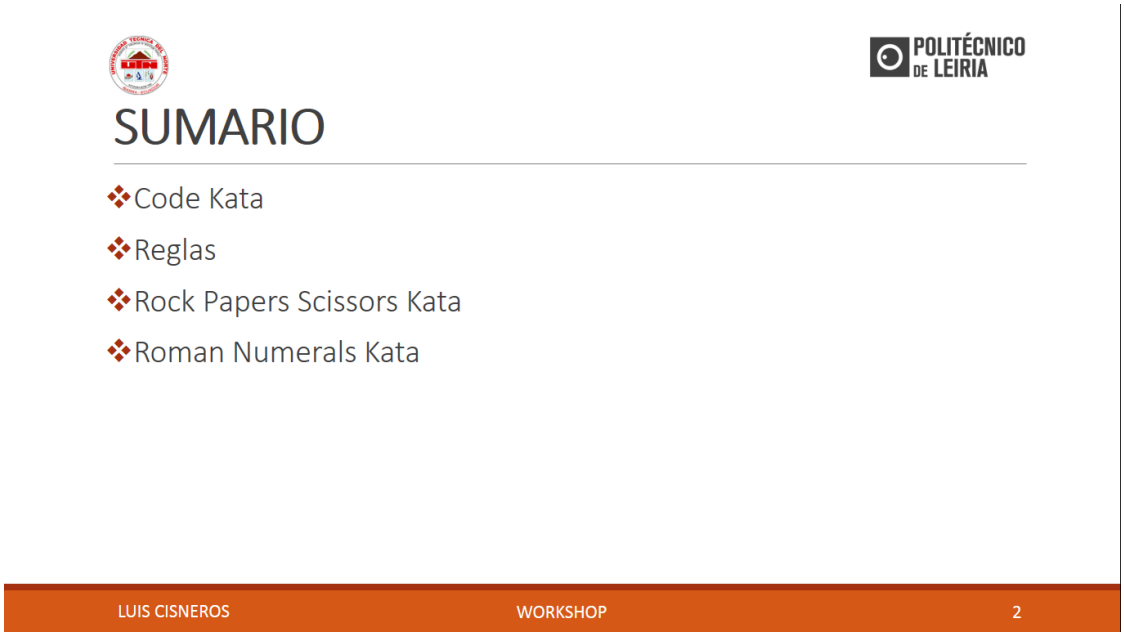
Appendix C: Workshop Content

(see appxC_ContenidoWorkshop.docx)

A ppendix D: Code Katas Exercises - slides

(see appxD_PresentacionEjemplos,pptx)

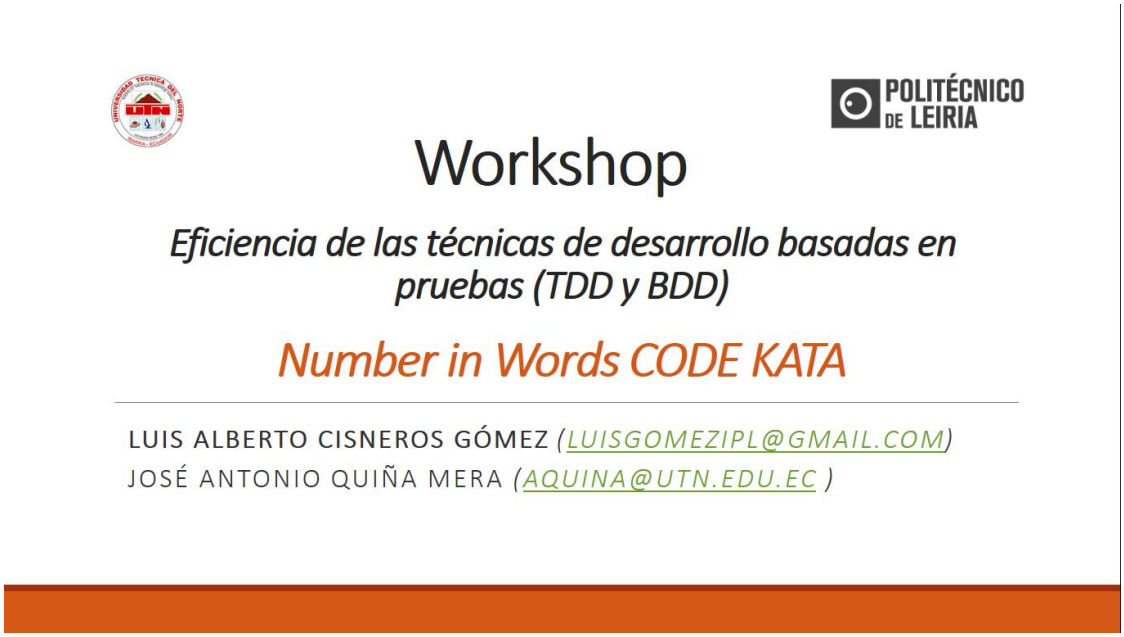
Slide 1



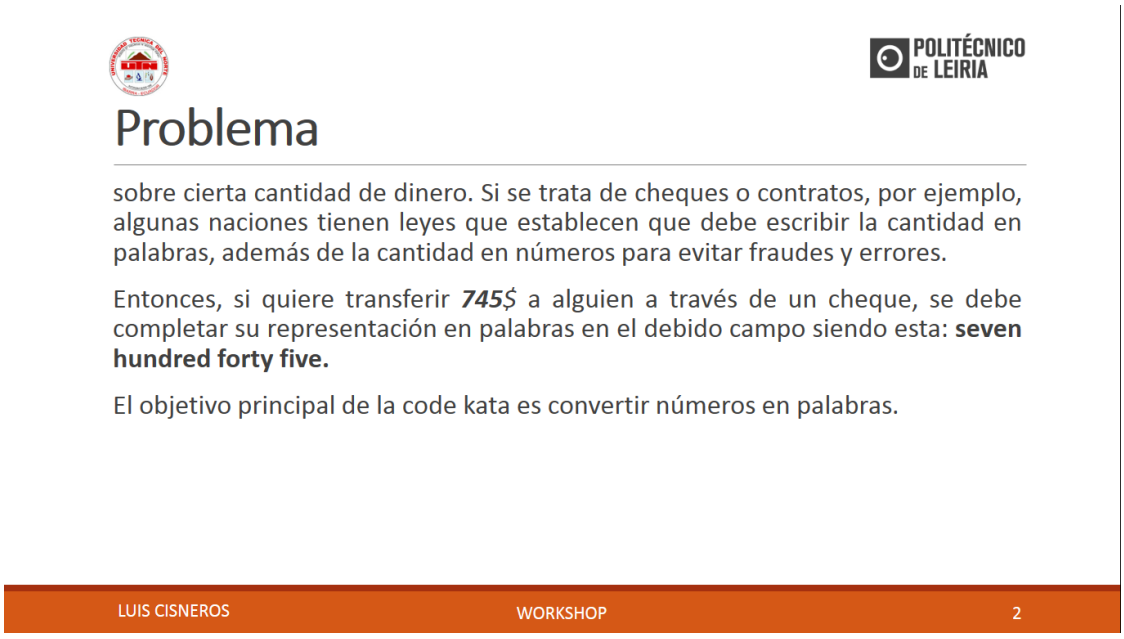
Slide 2

Appendix E: Code Katas Problems - slides

(see appxE_NumbersInWords,pptx)



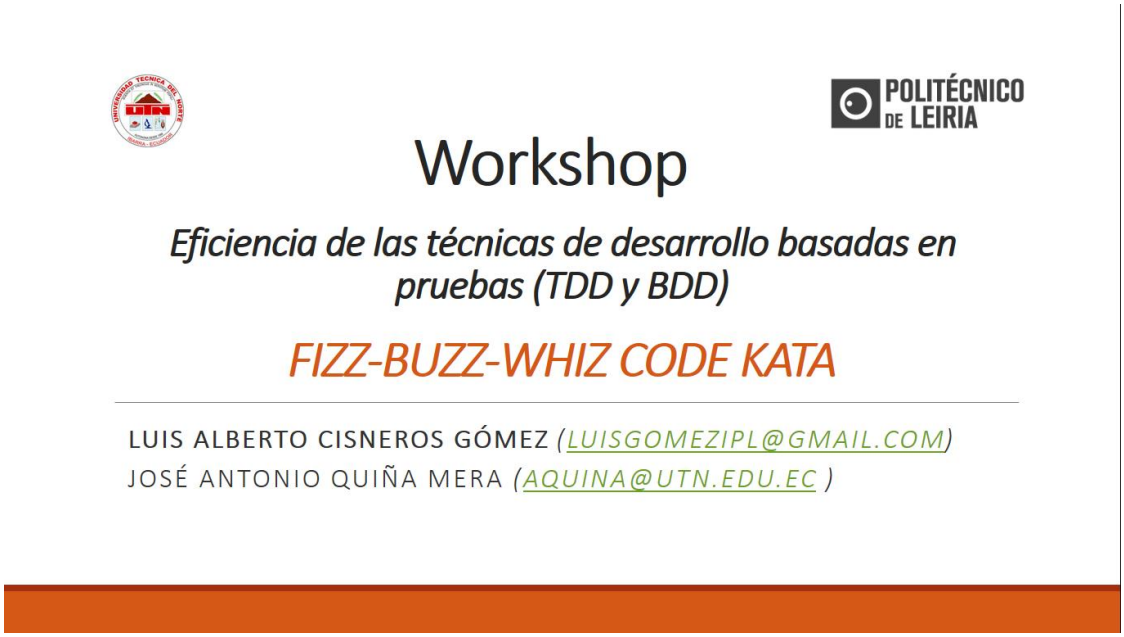
Slide 1



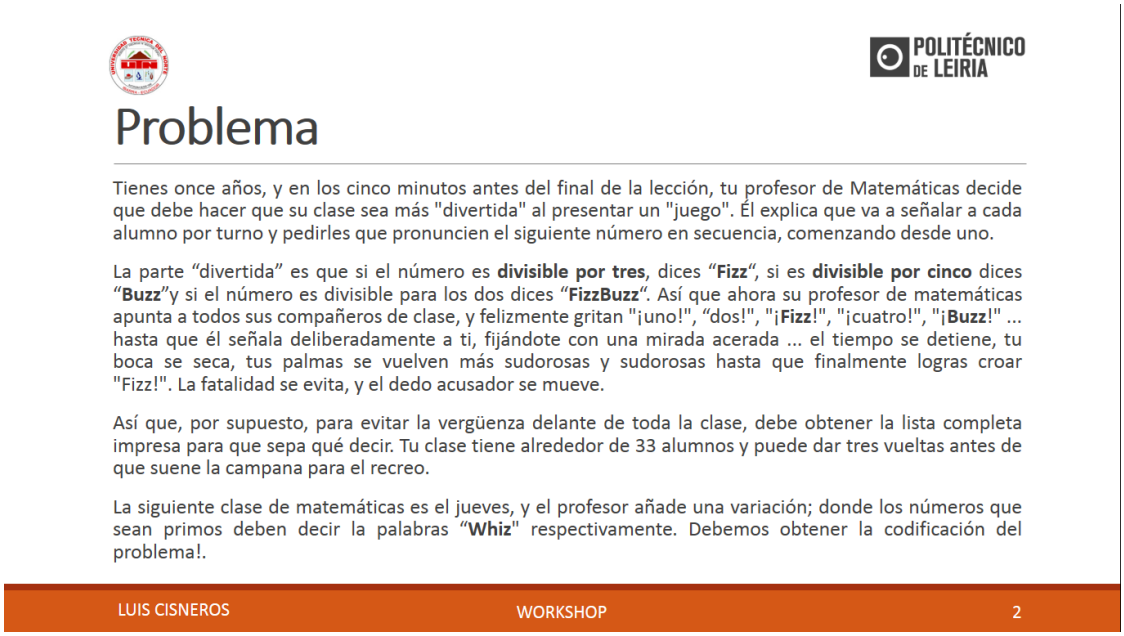
Slide 2

Appendix F: FizzBuzz slides

(see appxF_FizzBuzz,pptx)



Slide 1

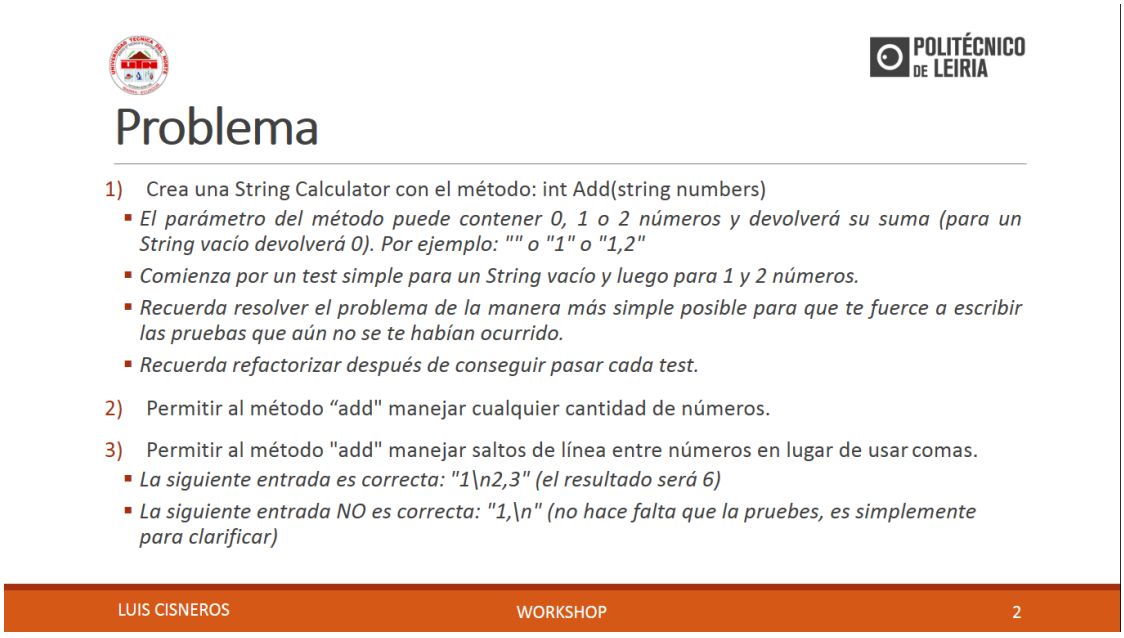


Slide 2

Appendix G: String Calculator slides

(see appxG_StringCalculator,pptx)

Slide 1



Slide 2

This page was intentionally left blank

Glossary

Software:	Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system [97]**,**
Software quality:	The totality of functionality and features of a software product that bear on its ability to satisfy stated or implied needs [97],
Software lifecycle:	The period of time that begins when a software product is conceived and ends when the software is no longer available for use [97],
Requirement:	A condition or capability needed by a user to solve a problem or achieve an objective that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document [97],
Functionality:	The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions [97],
Functional requirement:	A requirement that specifies a function that a component or system must perform [97],
Non-functional requirement:	A requirement that does not relate to functionality, but to attributes such as reliability, efficiency, usability, maintainability and portability [97],
Risk:	A factor that could result in future negative consequences [97],
Bug/defect:	A flaw in a component or system that can cause the component or system to fail to perform its required function, e,g,, an incorrect statement or data definition [97],
Error:	A human action that produces an incorrect result [97],
Failure:	Deviation of the component or system from its expected delivery, service or result [97],
Refactoring:	Refactoring consists of improving the internal structure of an existing program's source code, while preserving its external behavior [98],
Framework:	A software framework is a concrete or conceptual platform

	where common code with generic functionality can be selectively specialized or overridden by developers or users [99],
Programming language:	A programming language is a computer language engineered to create a standard form of commands; these commands can be interpreted into a code understood by a machine [100],
Paradigm:	A paradigm is a pattern or an example of something, Thomas Kuhn uses the word to mean the model that scientists hold about a particular area of knowledge [101],
Feature:	An attribute of a component or system specified or implied by requirements documentation (for example reliability, usability or design constraints) [97],
Test plan:	A document describing the scope, approach, resources and schedule of intended test activities [97],
Product Owner:	The role in Scrum accountable for maximizing the value of a product, primarily by incrementally managing and expressing business and functional expectations for a product to the Development Team(s) [102],
Developer:	A developer is a person that builds and create software and applications; they write, debug and execute the source code of a software application [103],
Software architect:	A software architect is a developer who is responsible for the high-level design and strategic planning of new software products [104],
Business analyst:	A business systems analyst is a person whose job it is to apply business goals to the IT system of an enterprise [105],
Domain expert:	A domain expert is a person with special knowledge or skills in a particular area of endeavor,
Systems analyst:	A systems analyst is an IT professional who works on a high level in an organization to ensure that systems, infrastructures and computer systems are functioning as effectively and efficiently as possible [106],
Tester:	A skilled professional who is involved in the testing of a

component or system [97],

Client: Person, company or organization that specifies the requirements of the system,

Quality assurance: Part of quality management focused on providing confidence that quality requirements will be fulfilled [97],

Reliability: The ability of the software product to perform its required functions under stated conditions for a specified period of time, or for a specified number of operations,

Usability: Extent to which a software product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use [97],

Efficiency: Resources expended in relation to the extent with which users achieve specified goals [97],

Maintainability: The ease with which a software product can be modified to correct defects, modified to meet new requirements, modified to make future maintenance easier, or adapted to a changed environment [97],

Portability: The ease with which the software product can be transferred from one hardware or software environment to another,

Bottom-up testing: An incremental approach to integration testing where the lowest level components are tested first, and then used to facilitate the testing of higher level components [97],

Smoke testing: A subset of all defined/planned test cases that cover the main functionality of a component or system, to ascertaining that the most crucial functions of a program work, but not bothering with finer details [97],

Top-down testing: An incremental approach to integration testing where the component at the top of the component hierarchy is tested first, with lower level components being simulated by stubs [97],

Regression testing: Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made [97],

Acceptance criteria:	The exit criteria that a component or system must satisfy in order to be accepted by a user, customer, or other authorized entity [97],
User acceptance testing:	Acceptance testing carried out by future users in a (simulated) operational environment focusing on user requirements and needs [97],
Operational acceptance testing:	Operational testing in the acceptance test phase, typically performed in a (simulated) operational environment by operations and/or systems administration staff focusing on operational aspects [97],
User story:	A high-level user or business requirement commonly used in Agile software development, typically consisting of one sentence in the everyday or business language capturing what functionality a user needs and the reason behind this, any non-functional criteria, and also includes acceptance criteria [97],