Continuous Delivery: Aplicação na Glintt

FRANCISCO JOSÉ DA FONSECA RAMALHO

Outubro de 2018

Continuous Delivery: Application at Glintt

Francisco José da Fonseca Ramalho

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Computer Systems

Supervisor: Dr. Alexandre Manuel Tavares Bragança
Co-Supervisor: Engenheiro Joel Campos

Porto, October 14, 2018

Dedicatory

To my parents, my brother, my grandma and my girlfriend that always believed in me, more than myself.

Abstract

Nowadays, Continuous Integration and Continuous Delivery are practices that the organizations that develop software implement to facilitate and improve their daily work. These continuous practices allow organizations to perform more reliable and constant releases of new features and products.

In this dissertation, it will be documented the application of continuous practices in a real case, at Glintt Healthcare Solutions organization. A case study of methods, approaches and technologies will be made to understand the better way to implement those practices and take the most possible advantages of that process.

It is expected that this project decreases or completely solves the problems that originated it, contributing to improve the quality of the products and increase the costumers' trust in the organization.

Keywords: Continuous, integration, delivery, automation, testing

Resumo

Atualmente, os conceitos de Continuous Integration e Continuous Delivery estão relacionados com práticas que qualquer organização que desenvolve software implementa de modo a melhorar o seu trabalho diário. Estas práticas permitem às organizações entregar novos produtos e funcionalidades mais confiáveis e com maior frequência.

Nesta dissertação é documentada a aplicação dessas práticas num caso real, na organização Glintt Healthcare Solutions. Irá ser realizado um caso de estudo de métodos, abordagens e tecnologias relacionadas com essas práticas, tendo como objetivo perceber a melhor maneira de implementá-las e tirar o melhor proveito da aplicação desses processos.

É esperado que a realização deste projeto consiga diminuir ou colmatar os problemas verificados na organização e que lhe deram origem. Assim, espera-se que o projeto contribua para melhorar a qualidade dos produtos da organização e para melhorar a confiança dos clientes na mesma.

Acknowledgement

First, I'd like to thank Instituto Superior de Engenharia do Porto (ISEP) and all people who taught me a lot in the last five years. A special thanks to my supervisor Dr. Alexandre Bragança for the guidance given along the implementation of this solution and writing of this dissertation.

To Glintt-HS, I thank the opportunity to design and implement a solution related to a subject that I am very interested on. Special thanks to my supervisor Joel Campos for all the advices and the time given to me for implementing this project. Also, thanks to my team's colleagues that contributed to the existence of this solution and discussed various points about the best approaches and methods.

There is also so many people who went along with me at ISEP in the last years that it would be hard to enumerate them all; I thank all of them for all the moments we spent together. Special thanks to Paulo Pereira e Carlos Silvas, two of my best friends and colleagues at ISEP and Glintt-HS. Without them, this Master of Science would be very much harder and working in team wouldn't be so easy.

Finally, special thanks to my family - my parents, my brother and my grandma - for all the support and all the moral values and ideas given in my entire life. Also, a big thanks to my girlfriend who always been there for me in the toughest times and always encouraged me to keep fighting to overcome all the struggles.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

AMD	Asynchronous Module Definition.
BDD	Behavior-Driven Development.
CD	Continuous Delivery.
CI	Continuous Integration.
CLI	Command-Line Interface.
CSS	Cascading Style Sheets.
CSV	Comma-Separated Values.
DLL	Dynamic-Link Library.
DSRM	Design Science Research Methodology.
FEI	Front End of Innovation.
FFE	Fuzzy Front End.
GUI	Graphical User Interface.
HTML	HyperText Markup Language.
IDE	Integrated Development Environment.
IS	Information systems.
JS	JavaScript.
JSON	JavaScript Object Notation.
NCD	New Concept Development.
OS	Operating System.
TAP	Test Anything Protocol.
TDD	Test-Driven Development.
TFS	Team Foundation Server.
TS	TypeScript.
UI	User Interface.
VC	Value for the customer.
VCS	Version Control System.
VP	Value Proposition.

XML	Extensible Markup Language.
XP	Extreme Programming.

Chapter 1

Introduction

This dissertation focus on a pratical approach of Continuous Integration (CI) and Continuous Delivery (CD), applied in a real organization. In this first chapter, it is described the problem, the goals and the preconized approach, concluding with the current document structure.

1.1 Context

This document presents a solution implemented by Glintt Healthcare Solutions (Glintt-HS) [1], a company that belongs to the Global Intelligent Technologies (Glintt) [2] group. This group represents a Portuguese multinational, listed on the Lisbon stock exchange (Euronext Lisbon), and its mission is to "contribute with technological innovation to improve the levels of health and well-being in the global society in which we live." (Glintt – Global Intelligent Technologies 2018). Glintt-HS covers the Healthcare market, both the public and private sector, and has more than 20 years of experience in that market. Nowadays, more than 200 hospitals and clinics provide their services using Glintt solutions.These solutions are not only used in Portugal, but also in the United Kingdom, in Angola and in Brasil.

In Glintt-HS there was a problem related to the release of a stable version of a product (i.e., the version that is posteriorly installed in the costumers' environment). Regarding branch control, there is only a development branch and the production branch (version 17 branch for example) that is used to release a new product version to an internal production-like test environment (similar to the costumer environment) and later effectively to the costumer. With only the existence of that two branches, sometimes errors occurred when a new patch (version of the application) was deployed to the internal test environment, destabilizing it. The errors detected could take days to resolve and affect the release of that version to the costumer. A team of testers manually perform acceptance tests in that environment and only when no errors are detected, the patch is approved and closed and can be installed on the costumers' environment.

Trying to solve those problems, an intermediate environment and branch at the organization's repository were created. This new environment (called CI environment) has the objective to be the first environment where testers can approve new developments or bug corrections made primarily in the development branch. Only when these developments or bug corrections are approved in the new environment, their correspondents code changes can be merged to the 17 branch and deployed to the production-like environment. This will allow to keep a better control of the changes that go to the 17 branch and to maintain a more stable patch

[1] https://www.glintt.com/pt/o-que-fazemos/mercados/healthcare/Paginas/Home.aspx

[2] https://www.glintt.com/

environment. Also, only when a developer finishes its new development or bug fix can merge the changes to the intermediate branch, contributing to a stable CI environment as well.

To automate the integration and release of the application to the new CI environment, processes of Continuous Integration and Continuous Delivery were studied and implemented. This dissertation's project will focus on the CI/CD process of one specific project called *Atendimento*. This project's dimension and purpose is approached in chapter 3, more specifically in section 3.1.

1.2 Problem

The current CI/CD process that exists is really premature and continues to have its issues. The first issue is that the CI build is only triggered one time a day during the night; according to the good principles of CI, the best practice is to trigger a build automatically after a developer does a change to the source code central repository, to detect bugs of his code as soon as possible. With the practice followed by Glintt-HS, many changes made during the day are gathered in the same build executed during the night. If errors are detected, this means that the developer will only review his changes in the next day, when he could have done it minutes after his changes were made to the repository (if the build were triggered automatically with its commit/check-in). Furthermore, the errors detected can be not very specific sometimes and its harder to know who's changes originated that error, as changes by many developers are integrated in the same build.

Another big gap in the existing process is the lack of automated tests. In the current process, after the compilation of the source code, the resulted binaries are deployed to the CI environment (site that contains the CI version of the application). Due to these premature processes, sometimes the application has errors in runtime in that environment. The deployment with errors could be avoided with the execution of some tests in the previous stages. Adding the fact that this process doesn't have a rollback process, solving these runtime errors can take hours or even days to solve. As new changes made to CI environment are only merged to production branch when they are approved in that environment by a testers team, the problems referred can also influence the production version of the application by delaying scheduled releases of new features and, consequently, decreasing costumers' trust on the organization.

1.3 Goals

As mentioned above, the current process of CI at Glintt-HS is summarized in a daily integration and deployment of the new changes integrated to the internal test CI environment. With this new solution, it is intended that automated tests and a robust process of CD are implemented. As there is a current process that it is pretended to be evolved to a new one, there are some questions that can be used to evaluate the work here documented. Is this new solution a viable way to overcome the current problems that exist at Glintt-HS? Does the final result make up for the resources and effort spent on the design and implementation of this solution? Does this new solution makes work easier to the people involved and make them more satisfied with the process?

Considering the questions stated above, the following goals are established for this project:

- Plan, design and implement a robust architecture of CD at Glintt-HS to decrease or solve the mentioned problems;

- Better process than the current existing one, satisfying the principles and good practices of Continuous Integration and Continuous Delivery;
- Implementation of a good testing strategy to solve many of the existing problems;
- Addition of test automation and rollback process to make the application more reliable.

The achievement of the points presented above lead to a more stable CI environment and also increases the satisfaction of the interested parts (testers, developers and managers). Lastly, it is expected that the solution documented indirectly improve the release of new production versions. It is expected that a more trusty and stable CI environment will lead to a more secure production environment and with features released to production faster.

1.4 Value Analysis

As stated before, CI and CD are composed by practices that help organizations to improve their release processes. The solution here documented predicts the automation of processes in Glintt-HS, contributing to have a more stable CI environment and, consequently, a more secure merge of changes to the production-like environment. These new resulted conditions will facilitate the work of people involved in these processes (testers, developers and managers), by automating the integration of code, tests and the deployment of the application to the CI environment. Furthermore, the new process will increase product quality and reliability, adding value to Glinnt-HS proposition to its customers.

The value analysis of the concerned solution will be detailed in section 2.4 of chapter 2.

1.5 Preconized Approach

Every good solution to a problem, including in the area of Computer Systems, relies on a well structured research process. Normally, two methodologies are used in that research process to define the correct path to obtain knowledge about the issue concerned: behavioral science and design science. The first one "(...) is usually associated with theories creation", while the second "(...) is associated with the creation and evaluation of artifacts." (Al-Jallad 2012).

This dissertation presents a research focused on solving a real problem in the organizational context of Glintt-HS, so it is possible to conclude that this work follows the design science methodology. Design Science Research Methodology (DSRM) is a well known and accepted framework to implement design science research in Information systems (IS) research. This methodology will be used on the development of this dissertation's work and is is composed by six steps (Peffers et al. 2007):

1. Problem identification and motivation: definition of the problem and justification of the value of a solution. This justification allows to accept the results and to know researcher's understanding of the problem. The problem concerned in this dissertation is presented in this first chapter in section 1.2 and its value analysis and justification is also presented in this chapter and detailed in chapter 2;
2. Define the objectives for a solution: definition of the objectives of the solution from the problem concerned. These objectives can be quantitative or qualitative and require knowledge of the state of the problem and other solutions. The goals of this dissertation's solution are presented in section 1.3 of this chapter;

3. Design and development: involves the design of the artifact. This step includes the identification of functionality and architecture, then it is developed the designed artifact. The design of the solution is discussed in chapter 4 of this document. Its implementation is described in chapter 5.
4. Demonstration: prove that the artifact can solve the problem or parts of it. This demonstration is usually accomplished through experimentation, simulation, case study or similar activities, that in this project were performed at Glintt-HS. In the solution presented in this document, its value analysis can provide some clarifications on how it can solve the problem concerned. Furthermore, the next point (evaluation) is important to give more strength to that value analysis;
5. Evaluation: analyze the efficiency of the solution to solve the problem. The objectives are compared to the observed results in the demonstration activity. After evaluation, the researcher can choose to go back to a previous activity, if the results didn't match the stated objectives, or to advance to the next activity. The evaluation of the this document's solution is presented in chapter 6;
6. Communication: Communicate to other researchers and audiences the problem, the artifact, its design and its results demonstrated. This phase will be composed by the presentation of the work developed to an evaluation committee.

1.6 Document Structure

This document structure starts with the introduction chapter, where it is presented to the reader the conceptualization of the subject, the problem that this project will try to solve and the stated goals to achieve. A brief description of the value analysis and the approach to solve the presented problem conclude this chapter.

In chapter 2, this document presents the State of the Art, where is more detailed this project's theme and related concepts. First, the theoretical concepts are detailed and then their correspondent technological background is analyzed. In the same chapter, methods and approaches to implement a CI and CD process are described, and the value analysis of the current project is studied and detailed.

Chapter 3 shows an analysis of the solution, contextualizing with the current state of CI and CD at Glintt-HS first and then realizing a comparison and evaluation of the methods and approaches addressed in the previous chapter.

In chapter 4, the design of the documented solution is presented. The first section of the chapter describes the methods, approaches and technologies chosen to implement the solution and the next section presents its high-level diagrams.

The next chapter, chapter 5, contains all the information about the implementation process, presenting all the strategies used to develop the necessary tests and explaining the Team Foundation Server (TFS) and Jenkins workflows.

In chapter 6, it is presented the factors that will be used to evaluate this solution, including test hypotheses, the evaluation methodologies and the statistic tests to evaluate each factor. This chapter's goal is to evaluate results of the solution and write some conclusions for each factor analyzed.

The last chapter (chapter 7) presents the conclusions of all work developed in this project.

Chapter 2

State of the Art

In the chapter of the State of the Art is presented an overview of the project, being discussed important concepts and technologies related to the problem in the Theoretical Background and Technological Background, respectively. Also, it presents the known Methods and Approaches to the main issue reported in this dissertation and the Value Analysis of the project.

2.1 Theoretical Background

Continuous Integration and Continuous Delivery are practices that nowadays every company that develops software tries to implement because those practices facilitate the integration, testing and publishing of the daily work. “Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day” (Fowler 2006a). Every time a developer makes changes to the application, in other words, pushes new code to the central repository that contains the application, an automated build is triggered and tests are also executed. This triggered process allows to detect integration problems and to verify if the new code pushed breaks any part of the application. It also makes easier to detect the error when a test fails and allows to build cohesive applications more quickly and with less integration issues.

Continuous Delivery is an approach that allows to "(...) build software in such a way that the software can be released to production at any time." (Fowler 2013). Continuous Delivery depends directly on the implementation of the Continuous Integration process, by building the applications and running tests to detect errors. Only when the application is built and tested with success, it is possible to deploy the executables to production. Continuous Delivery is sometimes assumed has having the same meaning as Continuous Deployment, but these concepts are not the same. Continuous Deployment is a practice where every change made by a developer goes through the building and testing process and it is automatically pushed into production. With Continuous Delivery, the deployment to production is a choice dependent on the preferences defined by the team’s members. In other words, with Continuous Delivery not every changes to the code result in a release to production.

In the previous paragraphs, it is realized that Continuous Integration, Continuous Delivery and Continuous Deployment are concepts related to each other. The focus of Continuous Integration is on testing automation to detect if new changes don’t break the application and its previous functionalities. Continuous Delivery is an extension of Continuous Integration, complementing it with release automation that can be done daily, weekly, fortnightly, or whenever is required. These releases are usually triggered by a click on a button. In Continuous Deployment there is no human intervention, which means that if all tests end with success a new change is released to production. This process accelerates the feedback of customers and take pressure off the team because there isn't a scheduled day to release anymore (Pittet 2018).

To implement these practices usually a deployment pipeline is configured. This pipeline is also known as build pipeline or staged build and it is composed by several stages that combine the building and testing of the application and finishes with changes being committed to production. So, the main goal of this pipeline is to enable the automated process of releasing new changes to production. A typical pipeline includes three main stages: one responsible for the build automation and continuous integration; another stage for the automated tests; and the final one for the deployment automation (Phillips 2014). In the first stage, the application is compiled in order to create the executables that will be used in the next stages. The new functionalities developed are built, integrated with the rest of the application and even unit testing is performed at this phase. In the test automation stage, more complex tests are performed to realize if the application meets its requirements, whether they are functional, security or performance requirements. This stage may take a long time to run, depending on the application's size and complexity of the tests. The final stage of deployment can also be staged, with the new changes being monitored in a subset of the production environment before being completely released. The automation of the deployment allows the delivery of new functionality to users within minutes, and as the quality of the software is verified in the previous stages, this one is a reliable and a low-risk phase.

Every change made to the repository of the application should trigger a build that must be monitored by the developer that made that change. If it breaks the application, the developer must fix the issues until every commits he has done pass the integration build. There are two ways to trigger that build: using a manual build or using a continuous integration server (Fowler 2006a). The first approach is similar to the build that the developer does on its local machine before committing his changes to the repository. After committing, he uses the integration machine to check out the head of the mainline. The head of the mainline points to a specific commit in the branch that the developer is working, and realizing the check out makes it point to the most recent commit. After that, he triggers manually the integration build. In the second approach, the continuous integration server is waiting for commits made to the repository. When a commit is made, the CI server automatically checks out the head of the mainline, triggers a build and notifies the result of the build to the developer who made the commit. This developer can only conclude his commit is done when the build finishes successfully, either it is triggered manually or automatically. None of the approaches is necessarily the best one. The better approach depends on the team's and projects preferences and requirements.

An important part of the CI process is to include automated tests to detect bugs as soon as possible. Today, Test-Driven Development (TDD) and Extreme Programming (XP) have contributed a lot to the implementation of self-testing code and many people implement and appreciate it. Nevertheless, TDD and XP are not necessary to implement self-testing code. Both approaches defend that it is supposed to write tests before writing the code, but this is not mandatory for the purposes of CI. (Fowler 2006a).

Self-testing code is achieved with a group of automated tests that can check almost the all code base for bugs and these tests are normally executed through a command and are self-checking. A build is self-testing if the failure of any test causes the build failure.

As Martin Fowler refers, TDD allowed the rise of XUnit family of tools that are pretty good for self-testing code (Fowler 2006a). This family provides a variety of testing frameworks and it is a derivation of JUnit. These tools were started by Kent Beck that built a framework to organize and run unit tests. It allowed developers to easily define their own tests and to run them after every change made to the code (Fowler 2006b). The X in XUnit represents a programming language. So, JUnit is a testing framework for Java, CUnit is for C, CppUnit is for C++, NUnit is for .NET and there are many more.

XUnit tools are a good group of frameworks to implement self-testing code, but there are other tools that focused more on end-to-end testing and can be also used, like Selenium, Sahi, Watir and many more. These tools are used for Web Application testing, automating browsers behavior. This allows to simulate user actions on applications and compare the results of that actions with the expected results. If one comparison of results fail, is is expected that the all suite of tests results in failure as the build that triggered them.

Testing tools are discussed again in the next section, relating the best tool with the type of test to implement on the CI/CD process.

Statistically speaking, these methodologies have shown some great results when implemented in various organizations. In Figure 2.1, it is shown the results of a study made by ThoughtWorks and Puppet Labs in 2014, comparing high-performing organizations that implemented Continuous Integration and Delivery methodologies with its peers that use traditional methods.

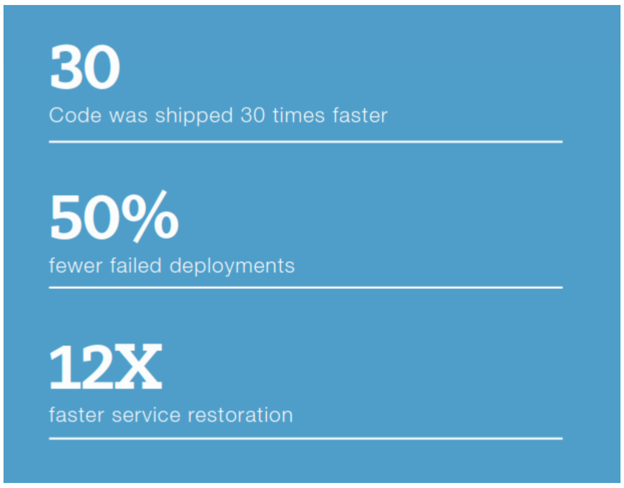


FIGURE 2.1: Statistics comparing organizations that use Continuous Integration and Delivery with peers using traditional methods (Gesso 2015)

In the results of the study, it can be concluded that the releases made to production suffered a huge improvement for the organizations that implement CI and CD methodologies. The code changes were released 30 times faster, the number of failed deployments was reduced in 50% and the service restoration was 12 times faster!

With similar outstanding results, Hewlett Packard made an internal study (A Practical Approach to Large-Scale Agile Development), as shown in Figure 2.2. This study proved that the implementation of CI and CD methodologies resulted in 40% reduction in development costs, 140% increase in programs under development, 78% reduction in development cost per program and increased the time available for innovation (5x more time).

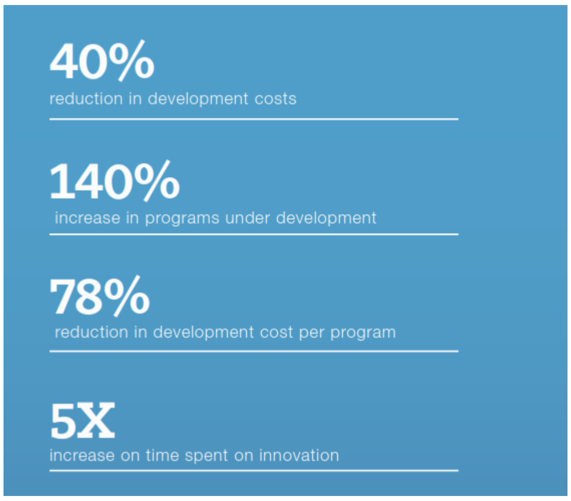


FIGURE 2.2: Study at Hewlett Packard using Continuous Integration and Delivery Methodologies (Gesso 2015)

2.2 Methods and Approaches

This section starts with the presentation of some good practices to implement a testing strategy and a delivery pipeline, based on the words of Jez Humble and David Farley. Next, some distinct processes of implementing the all process of CI and CD are discussed to understand the best way to construct the solution.

2.2.1 Good Practices for a Testing Strategy and a Delivery Pipeline

In their book "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation", Jez Humble and David Farley write about how they think testing should be implemented in real-life situations. They write about how a test strategy should be implemented in "New projects", "Midprojects" and "Legacy systems". The project approached in this dissertation is a Midproject because there is already a project with many features and with people developing even more new changes to the code. Also, as this project doesn't have automated tests, it is also a Legacy System. Next it is presented the strategy to implement tests in a project with these types.

In a midproject, "(...) the best way to introduce automated testing is to begin with the most common, important, and high-value use cases of the application." (Jez Humble 2011). To identify those use cases it is required to the costumer who will define the most value functionalities to the business. To cover these high-value functionalities it is recommended to automate happy path tests. "In general, for each story or requirement there is a single canonical path through the application in terms of the actions that the user will perform. This is known as the happy path." (Jez Humble 2011).

As the automation of the happy path is a priority, it will be necessary to perform more manual testing to check if the system is working as it should. If a function is tested manually more than a couple of times, and is not likely to change, the test should be automated. On the contrary, if someone is spending a lot of time fixing specific tests, that means the functionality is changing and it is required to talk to the costumer and the development team. If the functionality is indeed changing it is possible to ignore it while it is not finished.

Regarding Legacy Systems, "the first priority when dealing with such a system is to create an automated build process if one doesn't exist, and then create an automated functional test scaffolding around it." (Jez Humble 2011). To create this set of automated tests it is easier if documentation exists or if people who worked on the legacy system are available. Sometimes, the managers of the project consider these automation of tests a low-value activity, so it is necessary to identify the high-value actions of the system by talking to its users.

Talking to the system's users will allow to a set of automated tests that cover its core functionalities. This will protect the legacy functions and later more tests can be added to cover new behavior. For the legacy system, these tests are usually composed by smoke tests.

As part of this dissertation work, it is important to know how to implement a Deployment Pipeline. In their book, Jez Humble and David Farley present the following steps (Jez Humble 2011):

1. "Model your value stream and create a walking skeleton":

   First, it is necessary to map the value stream that goes from check-in to release. If the project is already running, this task is fast, it is just needed to talk to everyone involved in the process. If the project is new, the value stream must be the most appropriated. To define it, it is possible to check other project with similar characteristics or define minimum steps: "(...) a commit stage to build your application and run basic metrics and unit tests, a stage to run acceptance tests, and a third stage to deploy your application to a production-like environment (...)" (Jez Humble 2011).

2. "Automate the build and deployment process";

   In this step, the build process must start to compile the source code and produce binaries. This process must be triggered every time someone commits a change to the code. After that, the deployment is automated. For that, it is needed a machine that will be production-like and it is where user acceptance tests are executed. This should include the implementation of a deployment test to check if the application was correctly deployed.

3. "Automate unit tests and code analysis";

   This step is required to run unit tests, code analysis and ultimately a selection of acceptance and integration tests. Unit tests are faster to run because they don't connect to the filesystem or database, so they should be executed after building the application. With more complexity of the application, more unit and component tests are required. If this commit stage gets over five minutes, it is recommended to split it into suites that run in parallel. This parallelism can be achieved using a CI Server.

4. "Automate acceptance tests";

   "The acceptance test phase of your pipeline can reuse the script you use to deploy to your testing environment. The only difference is that after the smoke tests are run, the acceptance test framework needs to be started up, and the reports it generates should be collected at the end of the test run for analysis." (Jez Humble 2011). This stage is composed by functional and nonfunctional tests.

5. "Automate releases".

   The last goal in this process is to automate releases. Not all pipelines need to automatically deploy the application to production, as discussed before in the difference between Continuous Delivery and Continuous Deployment. This choice depends on the environment of the project, its requirements and business needs.

2.2.2 Build a CI and CD Solution

A process of CI/CD does not have a general solution and neither all its theoretical steps are required to be implemented. Therefore, each organization can mount its process in a different way, having in mind its business requirements and some restrictions/limitations that can exist.

A simple CI process is expected to accomplish the basic aspects defined in section 2.1. It should trigger an automated build each time a change is made to the source control repository and it should compile the sources into binaries that can be used to deploy the application. In a basic process, the next step could be the deployment of the application because, like stated, we have the necessary binaries. This is in general the current process that exists in Glintt-HS, but usually the build is not triggered automatically after a change, it is scheduled for some hour of the day. After the publication of the application is done, a team of testers is responsible for manually test it, trying to find bugs and communicate it to the development team. The current process of Glintt-HS will be more detailed in the chapter 3.

This process described above is very premature, not having implemented automated tests that would give more consistency to the process. The compilation of the sources of the code is not enough to understand that all modules of the application are fulfilling their tasks. Many times the application is published and errors emerge on runtime. Adding test steps to the process, these would be executed between the compile of the sources and the deployment of the application; considering that there are tests that need to be performed in a production-like environment, a deployment can be done before the final one. The automated tests that can be implemented and included in the CI process are mentioned in section 2.1.

After successfully integrating tests in the CI/CD process, there are not a lot alternatives in play, except for the deployment part. As discussed in section 2.1, the CI process can originate Continuous Delivery or Continuous Deployment. The main difference between the two concepts is in the automation of the deployment. In the first concept approach, the deployment is done according to a click triggered by a person, while in the second case, the deployment is automatic. As stated before, none of this approaches is seen as the right one; the use one or another depends on the organization's choices and environment.

Having as example the case of Glintt-HS, the automation of tests does not mean that manual tests should not be done by another team after the deployment of the application. Automated tests may not cover parts of the application that a human eye can. Manual and automated tests complete each other and allow to have more confidence in the application.

2.3 Technological Background

In this section, some tools that allow the implementation of CI and CD methodologies are presented. First, it is presented some CI servers that can be used: it is discussed some trends on those technologies, showing the top ones used on the last years and described in more detail some tools important for this project and that are already in use in the organization, and one or two alternatives to those tools. Then, it is presented some technologies used to implement tests that can be executed in the CI process. These technologies are also important for this project's work.

2.3.1 CI Servers

In the last years, many Continuous Integration servers have emerged, including open source tools. In 2014, as shown in Figure 2.3, Jenkins[1] was clearly on top of the CI servers used by developers and organizations. Amongst the respondents of the inquiry, more than half of them used Jenkins as their CI server, approximately 70%. None of the other tools referred by developers were used by more than 10% of them. Hudson[2] which once was considered a rival of Jenkins, appears in the third position with only 8% usage. Bamboo[3] appeared as the second most used tool (9%) and TeamCity[4] as the fourth (7,5%). These two tools are used by some developers probably because of the popularity of Atlassian's and JetBrains' tools for developers. Travis CI[5] and CruiseControl[6] represent a smaller percentage of the market, appearing as 2% and 1%, respectively. The rest of the respondents (2,5%) use other CI servers (White 2014).

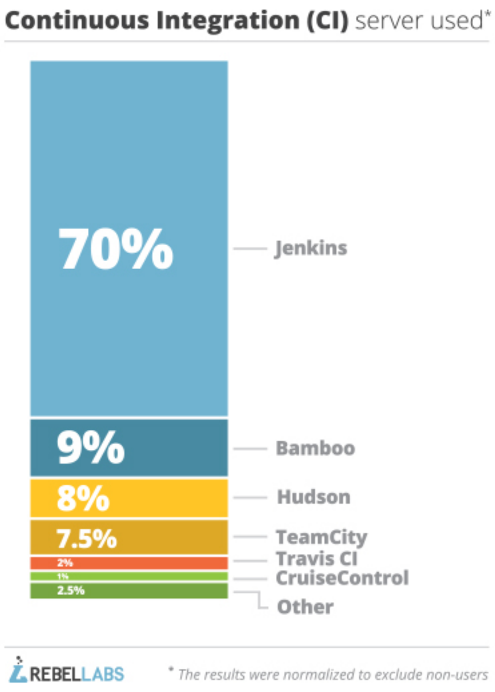


FIGURE 2.3: Continuous Integration servers used (2014) (White 2014)

Today, Jenkins continue to be one of the top CI Servers used by developers and organizations and some of the other tools referred above continue to be on the top eight of CI Servers. Considering a lot of sources in the Web about CI tools, here is the top 8 of Continuous Integration (Pecanac 2016):

- Jenkins;
- TeamCity;

[1]https://jenkins.io/
[2]http://hudson-ci.org/
[3]https://www.atlassian.com/software/bamboo
[4]https://www.jetbrains.com/teamcity/
[5]https://travis-ci.org/
[6]http://cruisecontrol.sourceforge.net/

- Travis CI;
- Go CD;
- Bamboo;
- GitLab CI;
- CircleCI;
- Codeship.

In the list presented, it can be observed that Jenkins, Bamboo, TeamCity and Travis CI remain in the top of CI tools, as in the study of 2014 described above. Go CD is the old Cruise Control that was referred in the study of 2014, being the new and fresh CI tool of the ThoughtWorks company. Comparing this list with Figure 2.3, it is observed that Hudson is no longer on the top of CI tools. Jenkins emerged as a fork of Hudson after a dispute with Oracle (that bought the Sun Microsystems that developed Hudson). Hudson continued to be developed by Oracle before being donated to the Eclipse Foundation, but never had the ability to compete with Jenkins and lost the race against the other tools as well.

Next, it is described in more detail two tools of the list presented above that can be used in the implementation of the solution to the problem discussed: Jenkins and TeamCity. Also, there is a description of Microsoft TFS[7]. This tool is currently used in the organization to keep its source code under control and in the premature CI process that exists. It will be an important part of the CI process to obtain the project's source code and some tests that will exist as well, and it can also be used to improve that process.

Jenkins

"Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software." (Jenkins 2018) As "the leading open source automation server", Jenkins offer the following values and advantages (Jenkins 2018):

- Continuous Integration and Continuous Delivery: Jenkins can be used as a simple CI Server or used as a Continuous Delivery hub for any project;
- Easy installation: it is a Java-based program, making available packages for Windows, Mac OS X and other Unix-like operating systems;
- Easy configuration: it is configured through its web interface that includes error checking and built-in help;
- Plugins: provides hundreds of plugins that allow the integration with almost every tool of CI and CD;
- Extensible: it can be extended through its plugin architecture;
- Distributed: it distributes work across various machines, allowing to implement builds, tests and deployment in multiple machines faster.

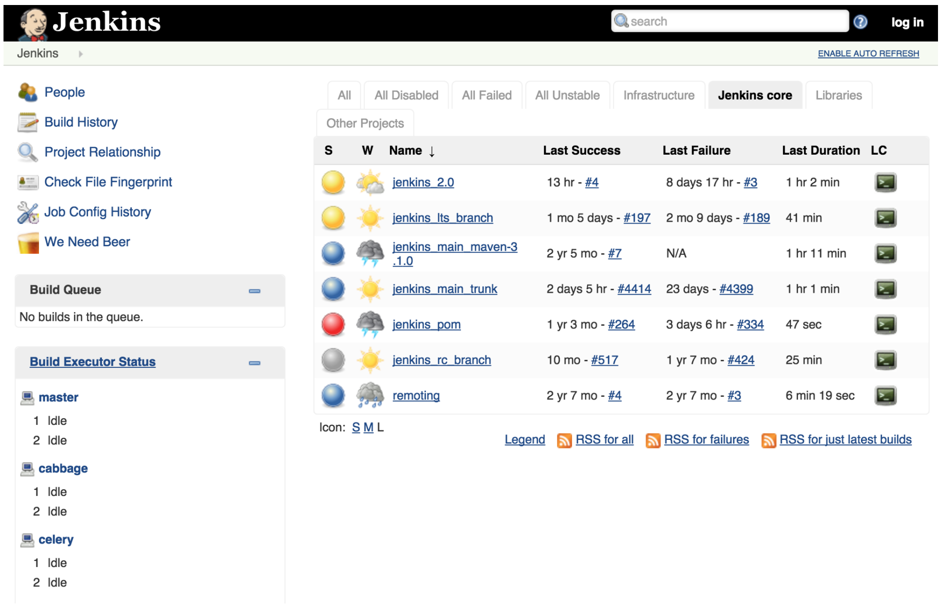
Jenkins was created in 2004 by Kohsuke Kawaguchi while working at Sun Microsystems, at that time the project was named Hudson. As stated before, due to disputes between Oracle and project collaborators, Jenkins was forked from Hudson in 2011. The collaborators that

[7]https://www.visualstudio.com/tfs/

started Hudson (including Kawaguchi) moved to Jenkins and Oracle continued developing Hudson, until Eclipse took over it. Today, Kohsuke Kawaguchi is the CTO at CloudBees and this company is an important participant in the Jenkins project, having employees exclusively working for this project. Hudson has been announced as obsolete by Eclipse in February 2017, also referring that "Jenkins is the replacement for it." (Eclipse 2017)

This CI Server is commonly used for "building snapshot and release artifacts for your application, deployment of the released artifact with custom scripts, Continuous integration pipeline support for establishing software development life cycle work flow for your application and support for scheduled builds and automation test execution" (Easy 2018). Furthermore, it can be associated to a source control server, trigger builds by commit, polling and periodically, can run shell and bash scripts, ANT and Maven targets and can publish results, archive artifacts and send email notifications.

In Figure 2.4 it is presented an overview of Jenkins' interface.



Build Queue

No builds in the queue.

Build Executor Status

master
1 Idle
2 Idle

cabbage
1 Idle
2 Idle

celery
1 Idle
2 Idle

S	W	Name ↓	Last Success	Last Failure	Last Duration	LC
🟡	⛅	jenkins_2.0	13 hr - #4	8 days 17 hr - #3	1 hr 2 min	>_
🟡	☀	jenkins_lts_branch	1 mo 5 days - #197	2 mo 9 days - #189	41 min	>_
🔵	⛈	jenkins_main_maven-3.1.0	2 yr 5 mo - #7	N/A	1 hr 11 min	>_
🔵	☀	jenkins_main_trunk	2 days 5 hr - #4414	23 days - #4399	1 hr 1 min	>_
🔴	⛈	jenkins_pom	1 yr 3 mo - #264	3 days 6 hr - #334	47 sec	>_
⚪	☀	jenkins_rc_branch	10 mo - #517	1 yr 7 mo - #424	25 min	>_
🔵	🌧	remoting	2 yr 7 mo - #4	2 yr 7 mo - #3	6 min 19 sec	>_

Icon: S M L

Legend RSS for all RSS for failures RSS for just latest builds

FIGURE 2.4: Jenkins User Interface (UI) (CircleCI 2018)

TeamCity

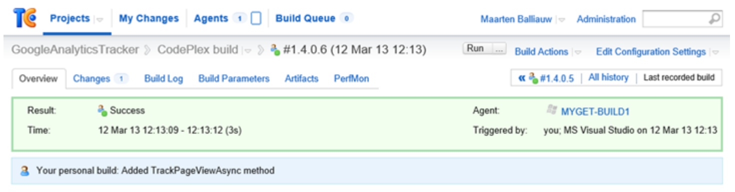
"TeamCity is a user-friendly continuous integration (CI) server for developers and build engineers free of charge with the Professional Server License and easy to set up!" (Alexandrova 2016). As Jenkins, it is a Java-based program, created by the JetBrains company. This tool offers the following features (JetBrains 2018):

- Technology Awareness: TeamCity provides a real good support to many tools. For example, in the case of Visual Studio projects it "provides automatic detection of tool versions, testing frameworks support, code coverage, static code analysis, and more" (JetBrains 2018), without the installation of any plugins;
- Key Integrations: it provides integration with key tools as the version/source control, issue tracker, build tool and package repository;
- Cloud Integrations: TeamCity implements cloud computing, by having its build agents running in Amazon EC2, Microsoft Azure, and VMware vSphere;

- Continuous Integration: it allows to implement CI, providing many features to construct build and test methodologies;
- Configuration: this tool allows to reuse settings and configurations, having in mind the principle to avoid duplication of settings as developers avoid duplication of code;
- Build History: with TeamCity it is possible to maintain an history of builds, changes and failures. That allows anyone to run history builds, see statistics and test history reports and add builds to favorites;
- Build Infrastructure: Team City is like "a conveyor belt of changes from developers and a bunch of testers taking the changes, verifying them and complementing these changes with verification results." (JetBrains 2018). In this case, testers are replaced with Build Agents;
- Code Quality Tracking: it helps developers and organizations achieving better code quality, having tools of code analysis and inspection integrated. Code quality can be a build failure condition;
- Version Control System (VCS) Interoperability: integration of version control systems is very good. any project code source can fetched by TeamCity;
- Extensibility and Customization: with TeamCity anyone can customize interact and extend their server. It is possible to interact via REST API, customize build scripts and create plugins for TeamCity;
- System Maintenance: TeamCity helps developers to maintain their CI Server, reporting about Disk Usage, Build Time, and Server Health;
- User Management: as TeamCity is used by many teams and organizations, it provides user management that includes user roles, user groups, user authentication and logs with all user actions.

Having in mind the features above, with TeamCity it is possible to "run parallel builds simultaneously on different platforms and environments, optimize the code integration cycle and be sure you never get broken code in the repository, review on-the-fly test results reporting with intelligent tests re-ordering, run code coverage and duplicates finder for Java and .NET and customize statistics on build duration, success rate, code quality, and custom metrics" (Alexandrova 2016).

In Figure 2.5 it is presented an overview of TeamCity's Web UI.



GoogleAnalyticsTracker › CodePlex build › #1.4.0.6 (12 Mar 13 12:13)

Result: Success
Time: 12 Mar 13 12:13:09 - 12:13:12 (3s)
Agent: MYGET-BUILD1
Triggered by: you; MS Visual Studio on 12 Mar 13 12:13

Your personal build: Added TrackPageViewAsync method

FIGURE 2.5: TeamCity UI (Balliauw 2013)

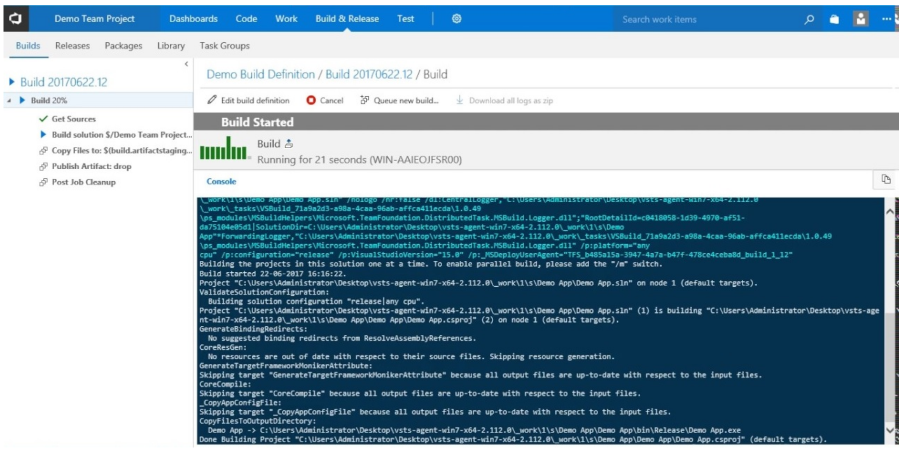
Team Foundation Server

"Team Foundation Server provides a set of collaborative software development tools that integrate with your existing IDE or editor thus enabling your cross-functional team to work effectively on software projects of all sizes." (Microsoft 2018). It is a Microsoft product that provides the following features (Microsoft 2018):

- Version control: TFS provide unlimited private repositories to store projects' code. It allows the administrators of the server to manage permissions and policies for those repositories;
- Tools for Agile teams: it allows to manage the agile processes of teams and organizations, using backlogs and Kanban boards. This makes easier the implementation of agile methodologies as Scrum and Kanban. Work items from the backlogs or boards can be linked to code and builds, making easier to follow the process;
- Continuous integration: TFS provides tools to construct CI builds to trace errors as soon as possible, to automate and track deployments (release management), to maintain code quality with its testing tools and to deliver software faster with code and module reuse (package management);
- Java support: the developers can use any IDE of their choice - Eclipse, IntelliJ, Android Studio, Visual Studio Code and more - and use any build tool - Ant, Maven and Gradle. Also, they can implement CI and CD with TFS features or integrating with systems like Jenkins. TFS allows to develop software for any platform and mobile languages, including C++, PHP, Python, Go, Swift;
- Integration: it is possible any tool or third-party service with TFS, using REST API and OAuth 2.0. It is also easy to integrate other services and tools from Microsoft.

With TFS and specially with Team Foundation Build process, it is possible to define a build process (automated build to compile and test code), customize that build process (run scripts before and after compilation, create personalized templates for the build process and use environment variables), run, monitor, and manage your builds, and diagnose build problems (checking logs created by the build).

In Figure 2.6 there is an overview of TFS's Web UI.



```
\_work\_tasks\VSBuild_71a9a2d3-a98a-4caa-96ab-affca411ecda\1.0.49
\ps_modules\MSBuildHelpers\Microsoft.TeamFoundation.DistributedTask.MSBuild.Logger.dll";"RootDetailId=c0418058-1d39-4970-af51-
da75104e05d1|SolutionDir=C:\Users\Administrator\Desktop\vsts-agent-win7-x64-2.112.0\_work\1\s\Demo
App"*ForwardingLogger,"C:\Users\Administrator\Desktop\vsts-agent-win7-x64-2.112.0\_work\_tasks\VSBuild_71a9a2d3-a98a-4caa-96ab-affca411ecda\1.0.49
\ps_modules\MSBuildHelpers\Microsoft.TeamFoundation.DistributedTask.MSBuild.Logger.dll" /p:platform="any
cpu" /p:configuration="release" /p:VisualStudioVersion="15.0" /p:_MSDeployUserAgent="TFS_b485a15a-3947-4a7a-b47f-478ce4ceba8d_build_1_12"
Building the projects in this solution one at a time. To enable parallel build, please add the "/m" switch.
Build started 22-06-2017 16:16:22.
Project "C:\Users\Administrator\Desktop\vsts-agent-win7-x64-2.112.0\_work\1\s\Demo App\Demo App.sln" on node 1 (default targets).
ValidateSolutionConfiguration:
  Building solution configuration "release|any cpu".
Project "C:\Users\Administrator\Desktop\vsts-agent-win7-x64-2.112.0\_work\1\s\Demo App\Demo App.sln" (1) is building "C:\Users\Administrator\Desktop\vsts-age
nt-win7-x64-2.112.0\_work\1\s\Demo App\Demo App\Demo App.csproj" (2) on node 1 (default targets).
GenerateBindingRedirects:
  No suggested binding redirects from ResolveAssemblyReferences.
CoreResGen:
  No resources are out of date with respect to their source files. Skipping resource generation.
GenerateTargetFrameworkMonikerAttribute:
Skipping target "GenerateTargetFrameworkMonikerAttribute" because all output files are up-to-date with respect to the input files.
CoreCompile:
Skipping target "CoreCompile" because all output files are up-to-date with respect to the input files.
_CopyAppConfigFile:
Skipping target "_CopyAppConfigFile" because all output files are up-to-date with respect to the input files.
CopyFilesToOutputDirectory:
  Demo App -> C:\Users\Administrator\Desktop\vsts-agent-win7-x64-2.112.0\_work\1\s\Demo App\Demo App\bin\Release\Demo App.exe
Done Building Project "C:\Users\Administrator\Desktop\vsts-agent-win7-x64-2.112.0\_work\1\s\Demo App\Demo App\Demo App.csproj" (default targets).
```

Figure 2.6: TFS Build UI

2.3.2 Testing Tools

One of the big problems in Glintt-HS and in its premature CI and CD process is the lack of automated tests of any kind. In this section, it is discussed the types of tests that the project concerned needs and some tools and methods used to implement each of those tests. The study of these methods are important to define the approach to follow on the implementation of the solution to the problem presented in this dissertation. The approach to implement is discussed in chapter 3.

Nowadays in Glintt-HS, the absence of tests sometimes result in deployments with errors that are consequence of bad integration with other modules and services developed by other teams, unexpected results of features developed or even corrupted binaries. So, in the automated build process it is important to add and configure automated tests. There are some kind of automated tests that can be considered:

- Unit tests;
- Integration tests;
- Deployment tests;
- Functional acceptance tests;
- Nonfunctional acceptance tests.

For these types of tests there are many tools available to implement them, and the best tool depends on the type of project concerned and the requirements related to their application. Furthermore, contextualizing with CI and CD processes, there are different approaches to integrate these tests in the automated build process. Next, it is presented some tools and approaches for each type of test. In the end of the chapter, it is presented some good practices to a testing strategy and to start and maintain a Delivery Pipeline.

2.3.3 Unit tests

"A unit test is a way of testing a unit - the smallest piece of code that can be logically isolated in a system. In most programming languages, that is a function, a subroutine, a method or property. The isolated part of the definition is important. In his book "Working Effectively with Legacy Code", author Michael Feathers states that such tests are not unit tests when they rely on external systems: “If it talks to the database, it talks across the network, it touches the file system, it requires system configuration, or it can’t be run at the same time as any other test." (SmartBear 2018)".

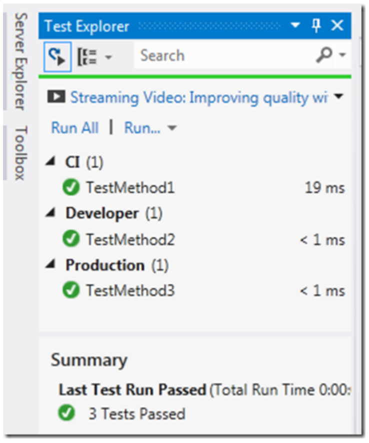
The project that this dissertation focus on is a Web application implemented in C#, so the next sections focus on tools to test .NET applications. Also, TypeScript (TS)[8] is very used, so it is presented some alternatives to test TS as well. In terms of tools to implement unit testing there are some mature alternatives and that are used by many developers. For example, MSTest[9], NUnit[10] and xUnit.NET[11] for c# and Jest[12], Mocha[13] and Tape[14] for TS/JavaScript (JS)[15] are a part of that list of tools and each one of them is approached next. Tape was chosen to be discussed because it was a little bit tested in Glintt-HS too.

[8]https://www.typescriptlang.org/
[9]https://msdn.microsoft.com/en-us/library/ms182489.aspx
[10]http://nunit.org
[11]https://xunit.github.io/
[12]https://facebook.github.io/jest/
[13]https://mochajs.org/
[14]https://github.com/substack/tape
[15]https://www.javascript.com/

MSTest, NUnit and xUnit.NET

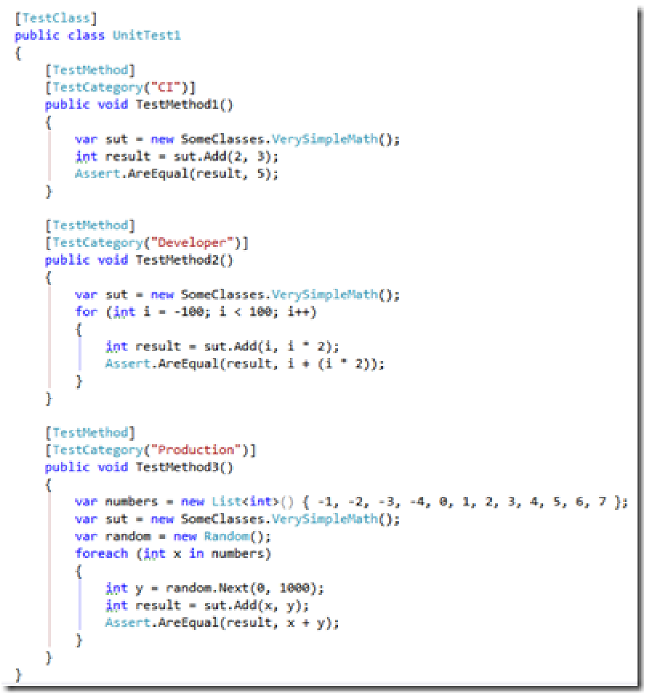
MSTest is a tool created by Microsoft to run unit tests for .NET applications. In practice, it was created as a command line tool and nowadays it is also referred as the Visual Studio Unit Testing Framework. Today, MSTest is integrated with Visual Studio, that means that anyone that install Visual Studio has automatically access to the MSTest tool in its Integrated Development Environment (IDE).

As MSTest comes integrated with Visual Studio, it is easy to create a new test project because it is just needed to use the test project template. So, to develop tests for a project it is good practice to create a test project, that in Visual Studio is possible with the "New project" option and then finding the test project type. After writing tests it is possible to execute them and its results appear in the IDE as shown in Figure 2.7. In Figure 2.8 is presented an example of the structure of a MSTest test.



```
◢ CI (1)
   ✓ TestMethod1        19 ms
◢ Developer (1)
   ✓ TestMethod2        < 1 ms
◢ Production (1)
   ✓ TestMethod3        < 1 ms

Summary
 Last Test Run Passed (Total Run Time 0:00:
 ✓ 3 Tests Passed
```

FIGURE 2.7: Test Explorer in Visual Studio (Sandstrom 2012)



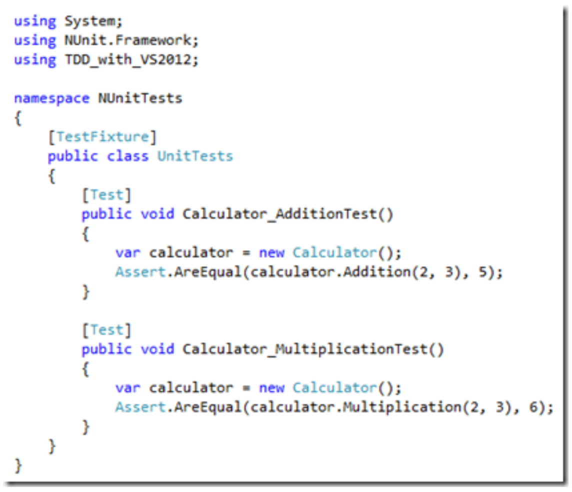
```
[TestClass]
public class UnitTest1
{
    [TestMethod]
    [TestCategory("CI")]
    public void TestMethod1()
    {
        var sut = new SomeClasses.VerySimpleMath();
        int result = sut.Add(2, 3);
        Assert.AreEqual(result, 5);
    }

    [TestMethod]
    [TestCategory("Developer")]
    public void TestMethod2()
    {
        var sut = new SomeClasses.VerySimpleMath();
        for (int i = -100; i < 100; i++)
        {
            int result = sut.Add(i, i * 2);
            Assert.AreEqual(result, i + (i * 2));
        }
    }

    [TestMethod]
    [TestCategory("Production")]
    public void TestMethod3()
    {
        var numbers = new List<int>() { -1, -2, -3, -4, 0, 1, 2, 3, 4, 5, 6, 7 };
        var sut = new SomeClasses.VerySimpleMath();
        var random = new Random();
        foreach (int x in numbers)
        {
            int y = random.Next(0, 1000);
            int result = sut.Add(x, y);
            Assert.AreEqual(result, x + y);
        }
    }
}
```

FIGURE 2.8: MSTest test structure (Sandstrom 2012)

"NUnit is a unit-testing framework for all .Net languages. Initially ported from JUnit, the current production release, version 3, has been completely rewritten with many new features and support for a wide range of .NET platforms." (Charlie Poole 2017). This framework is an open source software, now released under the MIT license. This license allows the use of NUnit without restrictions in free or paid applications.

Recently, NUnit joined the .NET Foundation that helps providing guidance and support to the NUnit project. The last version of the tool was developed by Charlie Poole, Rob Prouse, Simone Busoli, Neil Colvin and other community members that also helped in the previous versions. NUnit can be integrated with Visual Studio, installing the correspondent Visual Studio extension or the NUnit test adapter NuGet[16] package. In Visual Studio the presentation of test result is similar to Figure 2.7. The structure of the tests in NUnit is presented in Figure 2.9.

[16]https://www.nuget.org/



```
using System;
using NUnit.Framework;
using TDD_with_VS2012;

namespace NUnitTests
{
    [TestFixture]
    public class UnitTests
    {
        [Test]
        public void Calculator_AdditionTest()
        {
            var calculator = new Calculator();
            Assert.AreEqual(calculator.Addition(2, 3), 5);
        }

        [Test]
        public void Calculator_MultiplicationTest()
        {
            var calculator = new Calculator();
            Assert.AreEqual(calculator.Multiplication(2, 3), 6);
        }
    }
}
```

FIGURE 2.9: NUnit test structure (Oliveira 2012)

"xUnit.net is a free, open source, community-focused unit testing tool for the .NET Framework. Written by the original inventor of NUnit v2, xUnit.net is the latest technology for unit testing C#, F#, VB.NET and other .NET languages." (. Foundation 2017a). This tool is part of the .NET Foundation and it is licensed under Apache 2.

This tool divides tests "into “facts” and “theories” to distinguish between “always true” and “true for the right data”" (Dietrich 2017). Like NUnit, xUnit.Net can be integrated with Visual Studio. Previously, it was possible to install a Visual Studio extension to use xUnit.NET in that IDE. Today, the xUnit.NET Visual Studio Runner is only available via NuGet package. The presentation of results in Visual Studio is equal to what is shown in Figure 2.7 and in Listing 2.1 it is presented the structure of the tests in xUnit.NET.

```
using Xunit;

namespace MyFirstUnitTests
{
    public class Class1
    {
        [Fact]
        public void PassingTest()
        {
            Assert.Equal(4, Add(2, 2));
        }

        [Fact]
        public void FailingTest()
        {
            Assert.Equal(5, Add(2, 2));
        }

        int Add(int x, int y)
        {
            return x + y;
        }
    }
}
```

LISTING 2.1: xUnit.NET test structure (. Foundation 2017b)

In Table 2.1 it is presented the pros and cons of the three tools referred, serving as a comparison for those tools (Dietrich 2017).

TABLE 2.1: MSTest vs. NUnit vs. xUnit.NET.

	MSTest	**NUnit**	**xUnit.NET**
Pros	▪ Integrated in Visual Studios installation, easy access in the IDE ▪ Easy to create a new test project (File -> New Project) ▪ Code coverage without installing any other tools	▪ Interoperability - integrates with non-Microsoft build platforms and custom test runners ▪ Fast testing running ▪ Test annotations - allows specification of multiple inputs to a test	▪ Extremely intuitive terminology ▪ Excellent extensibility ▪ Commitment, responsiveness, and evangelism (by the authors)
Cons	▪ Performance ▪ Interoperability - integrates with Microsoft-/Visual Studio tools only	▪ Harder integration with Visual Studio comparing to MSTest - extra work, installing various tools	▪ Few or insufficient documentation ▪ Requires a different way of thinking about some things and more learning (results in a small core of users)

Jest, Mocha and Tape

"Jest is used by Facebook to test all JS code including React applications. One of Jest's philosophies is to provide an integrated "zero-configuration" experience. We observed that when engineers are provided with ready-to-use tools, they end up writing more tests, which in turn results in more stable and healthy code bases." (Facebook 2018).

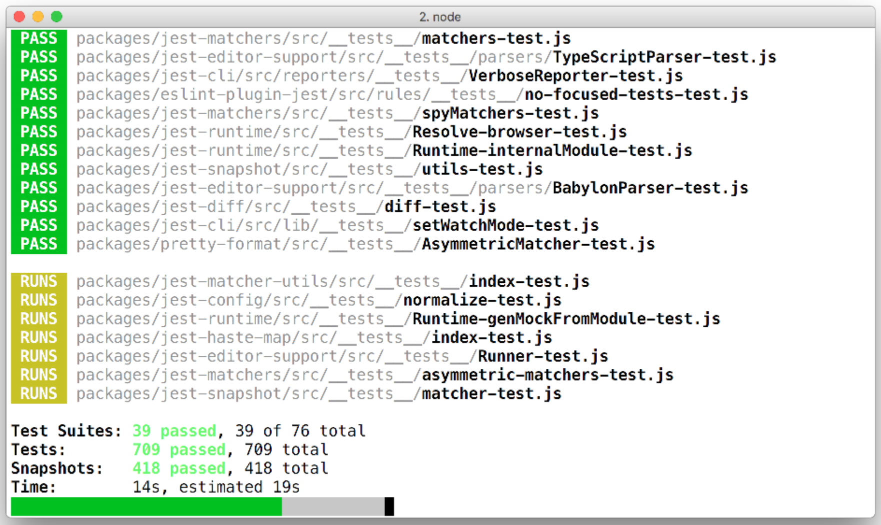
This tool provides the following characteristics (Facebook 2018):

- Fast and sandboxed: with Jest, test runs are parallelized to achieve maximum performance. In each test, sandboxed test files and global state are reseted, not allowing conflicts between different tests;
- Built-in code coverage reports: without any other setup or libraries, it is possible to generate code coverage reports using the option –coverage. Code coverage information includes untested files;
- Zero configuration: when using React or React Native projects, Jest is configured automatically. Just create a "__tests__" folder or files with .spec.js or .test.js extension and they will be executed by Jest;
- Powerful mocking library: Jest contains a powerful mocking library for modules and functions;
- Works with TypeScript: it accepts compile-to-JavaScript languages and integrates with Babel (JS compiler) and TS through ts-jest.

This JS test tool is used by many teams to test web applications, node.js services, mobile apps, and APIs. For example, it is used by Facebook, Instagram, Twitter, Spotify, Oculus and many others. In Listing 2.2 and in Figure 2.10, it is presented an example of a Jest test and an example of test results output, respectively.

```
const add = require('./add');
describe('add', () => {
  it('should add two numbers', () =>
    {
    expect(add(1, 2)).toBe(3);
  });
});
```

LISTING 2.2: Jest test example (Facebook 2018)



```
2. node
PASS  packages/jest-matchers/src/__tests__/matchers-test.js
PASS  packages/jest-editor-support/src/__tests__/parsers/TypeScriptParser-test.js
PASS  packages/jest-cli/src/reporters/__tests__/VerboseReporter-test.js
PASS  packages/eslint-plugin-jest/src/rules/__tests__/no-focused-tests-test.js
PASS  packages/jest-matchers/src/__tests__/spyMatchers-test.js
PASS  packages/jest-runtime/src/__tests__/Resolve-browser-test.js
PASS  packages/jest-runtime/src/__tests__/Runtime-internalModule-test.js
PASS  packages/jest-snapshot/src/__tests__/utils-test.js
PASS  packages/jest-editor-support/src/__tests__/parsers/BabylonParser-test.js
PASS  packages/jest-diff/src/__tests__/diff-test.js
PASS  packages/jest-cli/src/lib/__tests__/setWatchMode-test.js
PASS  packages/pretty-format/src/__tests__/AsymmetricMatcher-test.js

RUNS  packages/jest-matcher-utils/src/__tests__/index-test.js
RUNS  packages/jest-config/src/__tests__/normalize-test.js
RUNS  packages/jest-runtime/src/__tests__/Runtime-genMockFromModule-test.js
RUNS  packages/jest-haste-map/src/__tests__/index-test.js
RUNS  packages/jest-editor-support/src/__tests__/Runner-test.js
RUNS  packages/jest-matchers/src/__tests__/asymmetric-matchers-test.js
RUNS  packages/jest-snapshot/src/__tests__/matcher-test.js

Test Suites: 39 passed, 39 of 76 total
Tests:       709 passed, 709 total
Snapshots:   418 passed, 418 total
Time:        14s, estimated 19s
```

FIGURE 2.10: Jest test results (Facebook 2018)

"Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing simple and fun. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases." (MochaJS 2018).

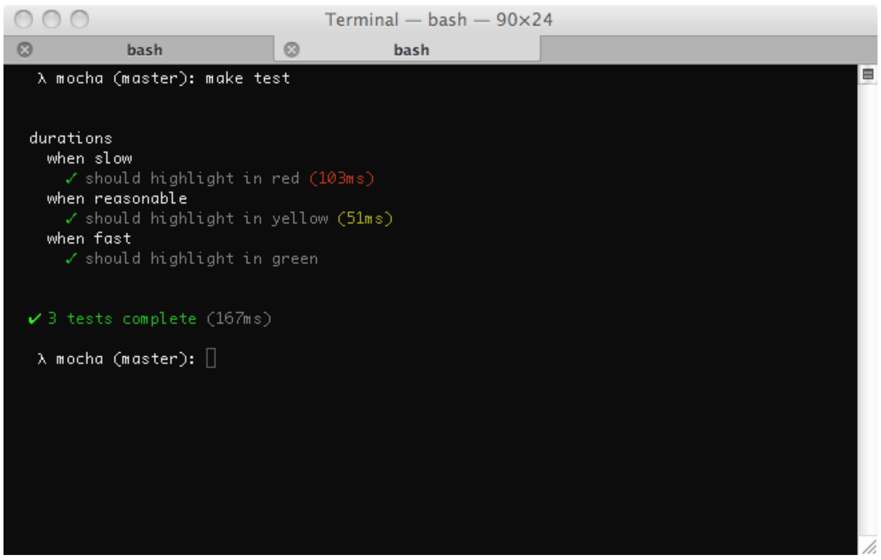
The most important features of this tool are (MochaJS 2018):

- Browser support: Mocha runs in the browser, having functions that only work on it;
- Test coverage reporting: it provides real-time code coverage report trough the continuous testing tool Wallaby.js;
- JS API for running tests;
- Use any assertion library you want: it is possible to use libraries as should.js, expect.js, chai, better-assert and unexpected;
- Simple async support, including promises: test asynchronous code is very simple, it is just needed to invoke a callback when the test is complete. By adding a callback, Mocha will know that it should wait for this function to be called to complete the test;
- Highlights slow tests: in the test report it shows the slower, reasonable and faster tests.

In Listing 2.3 and in Figure 2.11, it is presented an example of a Mocha test and an example of test results output, respectively.

```
var assert = require('assert');
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the
    value is not present', function()
     {
       assert.equal([1,2,3].indexOf(4)
    , -1);
    });
  });
});
```

LISTING 2.3: Mocha test example (MochaJS 2018)



```
 λ mocha (master): make test


durations
  when slow
    ✓ should highlight in red (103ms)
  when reasonable
    ✓ should highlight in yellow (51ms)
  when fast
    ✓ should highlight in green


✔ 3 tests complete (167ms)

 λ mocha (master):
```

FIGURE 2.11: Mocha test results (MochaJS 2018)

Tape is a "tap-producing test harness for node and browsers" (Halliday 2018) and it is released under the MIT License. It is a simple Test Anything Protocol (TAP) library. TAP "is a simple text-based interface between testing modules in a test harness" (TestAnything 2018), used to present test results.

In its core, it contains the following features (Halliday 2018):

- Pretty reporters: the default TAP output is good, but it is possible to customize the output with another npm[17] modules alongside Tape;
- Uncaught exceptions: these exceptions by default are not intercepted and Tape will crash. It is possible to change this, by using tape-catch to report exceptions as TAP errors;
- CoffeeScript support: it is possible to write Tape tests in CoffeeScript;
- Promise support: using blue-tape npm package, it is possible to test promises. If a promise is returned it will be checked for errors and if there are errors the test fails;

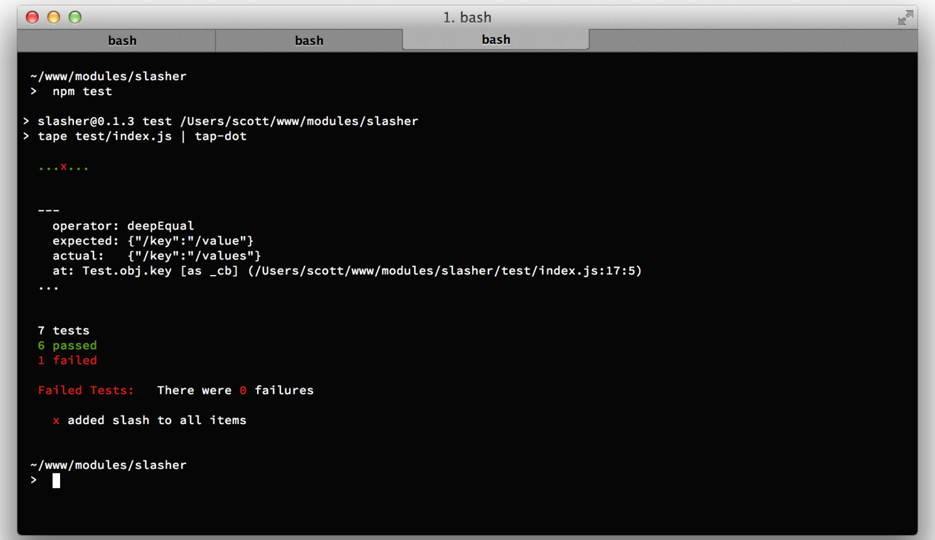
[17]https://www.npmjs.com/

- ES6 support: it includes a package that allows to run test suites that include ESNext/Harmony features, utilizing Babel.

In Listing 2.4 and in Figure 2.12, it is presented an example of a Mocha test and an example of test results output, respectively.

```
var test = require('tape');

test('timing test', function (t) {
    t.plan(2);

    t.equal(typeof Date.now, '
   function');
    var start = Date.now();

    setTimeout(function () {
        t.equal(Date.now() - start,
   100);
    }, 100);
});
```

LISTING 2.4: Tape test example (Halliday 2018)



```
~/www/modules/slasher
>  npm test

> slasher@0.1.3 test /Users/scott/www/modules/slasher
> tape test/index.js | tap-dot

  ...x...


  ---
    operator: deepEqual
    expected: {"/key":"/value"}
    actual:   {"/key":"/values"}
    at: Test.obj.key [as _cb] (/Users/scott/www/modules/slasher/test/index.js:17:5)
  ...


  7 tests
  6 passed
  1 failed

  Failed Tests:   There were 0 failures

    x added slash to all items


~/www/modules/slasher
>
```

FIGURE 2.12: Tape test results (Corgan 2016)

In Table 2.2 it is presented the pros and cons of Jest, Mocha and Tape (Harding 2017).

TABLE 2.2: Jest vs. Mocha vs. Tape

	Jest	Mocha	Tape
Pros	▪ Although it was primarily used to test React apps, it can be integrated with other applications ▪ Snapshot testing: good to check if the application's UI doesn't unexpectedly change between releases ▪ It has a wide API (not required to include additional libraries)	▪ Extensibility ▪ Different ways of configuration ▪ Flexibility in its assertions, spies and mocks ▪ Test structure as globals: not required to include or require it in every file	▪ Simple(gives everything you need and nothing more) ▪ Does not support globals (does not pollute the global environment) ▪ Typescript/CoffeeScript/ES6 support
Cons	▪ Globals populating environment	▪ Learn the assertion library each person chooses is required ▪ Plugins needed might require to include globals in every file (inconsistency with fourth advantage	▪ No setup/teardown methods (like beforeEach() and afterEach()): difficult if developers want to crate nested test suites ▪ Not advisable to build complex testing environments

2.3.4 Integration tests

"Integration tests determine if independently developed units of software work correctly when they are connected to each other." (Fowler 2018). So, in these tests many modules are combined together into the the entire system or sub-systems. "Looking at it from a more 2010s perspective, these conflated two different things: testing that separately developed modules worked together properly; test that a system of multiple modules worked as expected."(Fowler 2018). These tests are normally slower than unit tests because they require more setup and perform actions related to input/output, to connecting with databases, the filesystem or other systems.

The project discussed in this dissertation needs integration tests implemented in C#. Usually, the tools used for various programming languages to develop unit tests are also used to implement integration tests. Therefore, the tools referred in section 2.3.3 can be used to implement integration tests.

The tools that can be used to implement these tests are MSTest, NUnit and xUnit.NET as referred in the section above. The pros and cons of these tools are equal if they are used for unit tests or integration tests. So, there is enough information to see which tool is better to develop integration tests too.

2.3.5 Deployment tests

"The smoke test, or deployment test, is probably the most important test to write once you have a unit test suite up and running—indeed, it's arguably even more important. It gives you the confidence that your application actually runs. If it doesn't run, your smoke test should be able to give you some basic diagnostics as to whether your application is down because something it depends on is not working." (Jez Humble 2011). So, the objective of this test is to check if the deployment worked correctly, by checking if the application is correctly installer, configured, that it is responding and it is able to contact the services it requires.

To perform a smoke test, it is easier to haver a script that does it automatically. This test "(...) could be as simple as launching the application and checking to make sure that the main screen comes up with the expected content." (Jez Humble 2011). It "(...) should also check that any services your application depends on are up and running—such as a database, messaging bus, or external service." (Jez Humble 2011). Having the previous words in mind, a smoke test can be composed by a limited number of functional acceptance tests to check the core features of the application. This should not cover all the functional acceptance tests created, making the smoke test faster and providing feedback to developers about the last deployment as soon as possible. The functional acceptance tests and the tools to implement them are discussed in the next section 2.3.6, and they can be used in the deployment tests (at least the ones that cover main features).

To implement deployment tests, Stephen Haunts and other developers created a tool, the Smoke Tester[18], that helps developers to create a sequence of test steps to be executed. This tool has a graphical interface to make that process easier, where it is also possible to execute the tests. It also has integrated a command line test runner that allows to script the test execution. This tool is extensible, so anyone can easily add their custom test types. With this tools the following tests can be performed (Haunts 2014):

- Check environment variables are set;
- Check files exist;
- Check folders exist;
- Check HTTP connections return specified result codes;
- Check SQL Server connection strings;
- Check minimum disk space requirements.

2.3.6 Functional acceptance tests

Acceptance tests allow to check if the acceptance criteria for a story is fulfilled. These tests check various characteristics of the system, like functionality, capacity, usability, security, availability, etc. The ones that test the functionality of the system are the functional acceptance tests; the others are nonfunctional acceptance tests. These functional tests used to check if the features of the system satisfy the acceptance criteria should be executed in a production-like environment.

As mentioned before, the project this document is about is a Web Application. To implement functional acceptance tests on that type of project there are a lot of tools available. Next, it is discussed three of those tools: Selenium, Watir and Sahi.

[18]https://stephenhaunts.com/projects/post-deployment-smoke-tester/

Selenium, Watir and Sahi

"Selenium is an umbrella project encapsulating a variety of tools and libraries enabling web browser automation. Selenium specifically provides infrastructure for the W3C WebDriver specification — a platform and language-neutral coding interface compatible with all major web browsers." (SeleniumHQ 2018a). This tool is open source and it is released under the Apache 2.0 License.

It can be divided in two components: Selenium WebDriver and Selenium IDE. Selenium WebDriver is used to (SeleniumHQ 2018b):

- "Create robust, browser-based regression automation suites and tests";
- "Scale and distribute scripts across many environments".

Selenium IDE is used to (SeleniumHQ 2018b):

- "Create quick bug reproduction scripts";
- "Create scripts to aid in automation-aided exploratory testing".

This tool uses a custom build system that is available on all platforms (Linux, Mac, Windows), so it is easy to develop tests in any preferred environment.

In Listing 2.5 it is presented an example of a Selenium test.

```java
package com.example.tests;

import com.thoughtworks.selenium.*;
import java.util.regex.Pattern;

public class temp script extends
    SeleneseTestCase {
    public void setUp() throws Exception {
        setUp("http://localhost:8080/", "*
    iexplore");
    }
    public void testTemp script() throws
    Exception {
        selenium.open("/BrewBizWeb/");
        selenium.click("link=Start The BrewBiz
    Example");
        selenium.waitForPageToLoad("30000");
        selenium.type("name=id", "bert");
        selenium.type("name=Password", "biz");
        selenium.click("name=dologin");
        selenium.waitForPageToLoad("30000");
    }
}
```

LISTING 2.5: Selenium test example (Cohen 2012)

Watir is "(...) an open source Ruby library for automating tests. Watir interacts with a browser the same way people do: clicking links, filling out forms and validating text." (Watir 2018). It is open source and it supports different browsers, like Internet Explorer, Firefox, Chrome, Opera and Safari, and different platforms.

The most recent version of the Watir API is based on Selenium. So, Watir API was developed around the Selenium API. As Selenium, it provides infrastructure for the W3C WebDriver specification.

In Listing 2.6 it is presented an example of a Watir test.

```
browser = Watir::Browser.new :chrome

browser.goto 'google.com'
browser.text_field(title: 'Search').
    set 'Hello World!'
browser.button(type: 'submit').click

puts browser.title
# => 'Hello World! - Google Search'
browser.quit
```

LISTING 2.6: Watir test example (Watir 2018)

"Sahi is a free, open source tool for automation of web application testing. Sahi is very tester friendly and allows easy automation of even complex web 2.0 applications with lots of AJAX content. With an excellent recorder, smart object identification, simple scripting, automatic waits and inbuilt reports, Sahi gives the tester a powerful yet simple tool to accomplish testing across various browser and OS combinations. Sahi works on Internet Explorer, Firefox, Chrome, Safari, Opera etc. on Windows, Mac and Linux." (TytoSoftware 2018b).

Its paid version, Sahi Pro "(...) is a suite of mature, business-ready tools for automated testing of Web, Windows Desktop and Java applications. For testing teams which need rapid and reliable automation, Sahi Pro would be the best choice among automation tools." (TytoSoftware 2018a).

Sahi (open source) includes the following features (TytoSoftware 2018b):

- "Record on all browsers";
- "Playback on all browsers";
- "HyperText Markup Language (HTML) playback reports";
- "JUnit Style playback reports";
- "Suites and batch run";
- "Parallel playback (multiple instances of a browser running tests simultaneously)";

Defining tests in Sahi can be done through its Graphical User Interface (GUI) but it also can bw integrated with JUnit. Listing 2.7 is an example of a test script in JUnit.

```
public class test4demo extends BaseTestCase {

    @Before
    public void setUp() {
        login();
    }

    @After
    public void tearDown() {
        logout();
    }

    @Test
    public void testDemo1() {
        browser.textbox("email").setValue("demo
    for sahi");
        browser.submit("Submit").click();
        assertTrue(browser.label("error").exists
    ());
    }

    @Test
    public void testDemo2() {
        browser.textbox("email").setValue("
    demo@sahi.com");
        browser.submit("Submit").click();
        assertTrue(browser.label("ok").exists())
    ;
    }
}
```

Listing 2.7: Sahi test example (Li 2011)

In Table 2.3 it is presented the pros and cons of Selenium, Watir and Sahi.

Table 2.3: Selenium vs. Watir vs. Sahi

	Selenium	**Watir**	**Sahi**
Pros	▪ Multi browser, Operating System (OS) and language support ▪ Has an IDE (development is faster) ▪ Record and playback tests ▪ Choice of programming language	▪ Ruby library ▪ Multi browser and OS support ▪ Rich API ▪ Watij and Watin (Java and .NET)	▪ Multi browser support ▪ Has an IDE (development is faster) ▪ Record and playback tests
Cons	▪ Learn a vendorscript - Selenese (otherwise use another programming language)	▪ Learn Ruby (otherwise use Watij or Watin) ▪ Every browser requires a different library	▪ Interface is confusing ▪ Small community (less developed)

2.3.7 Nonfunctional acceptance tests

Until this section, the types of tests that were presented cover the behaviors of the application, in another words, its functional requirements. Nonfunctional acceptance tests cover nonfunctional requirements like capacity, availability, security, performance and others.

Nonfunctional acceptance tests may require special environments to be executed and specific knowledge to be implemented. They usually run through a long time, so in an automated environment they are executed less frequently than functional acceptance tests.

The concepts and testing tools discussed next will focus on two nonfunctional requirements: performance and capacity. These two concepts are often confused, so lets first clarify them. "(...) performance is a measure of the time taken to process a single transaction, and can be measured either in isolation or under load. Throughput is the number of transactions a system can process in a given timespan. It is always limited by some bottleneck in the system. The maximum throughput a system can sustain, for a given workload, while maintaining an acceptable response time for each individual request, is its capacity." (Jez Humble 2011).

With many tools available to implement nonfunctional tests, next it is presented two of those tools: WebLoad[19] and JMeter[20].

WebLoad and JMeter

WebLoad is a tool that allows to perform load testing, performance testing and stress testing on Web applications. Besides its best known paid version, it has a free edition which "includes 50 virtual users with access to all of WebLOAD features, except for Oracle Forms, Flex/AMF and streaming/multimedia features." (RadView 2018).

This tool developed by RadView is composed by the following features (RadView 2018):

- WebLOAD's IDE provides correlation, parameterization, response validation, messaging, native JS and debugging;
- Load Generation Console: allows massive virtual user load, including "locally and on the cloud, on Windows or Linux, via AWS or other cloud providers.";
- Analytics dashboards offer 80 different report templates;
- Integration with other tools - APM tools (Dynatrace, AppDynamics, New Relic), open source software (Selenium, Jenkins), mobile testing (Perfecto Mobile);
- Supports Web, mobile, and enterprise protocols and technologies - HTTP/HTTPS, WebSocket, PUSH, AJAX, SOAP, HTML5, WebDAV;
- Provides support to the user, by sending detailed documentation when the software is installed and having a support team available for any questions.

"The Apache JMeter™ application is open source software, a 100% pure Java application designed to load test functional behavior and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions." (A. S. Foundation 2018)

This tool has the following features (A. S. Foundation 2018):

[19] https://www.radview.com/
[20] http://jmeter.apache.org/

- Performs load and performance test to many different applications/server/protocol types - Web, SOAP/REST, FTP, Database via JDBC, LDAP, Native commands or shell scripts, Mail - SMTP(S), POP3(S) and IMAP(S), TCP;
- Test IDE that allows fast Test Plan recording, building and debugging;
- Dynamic HTML report;
- Ability to extract data from most popular response formats - HTML, JavaScript Object Notation (JSON) , Extensible Markup Language (XML) or any textual format;
- Complete portability and 100% Java purity;
- Full multi-threading framework;
- Caching and offline analysis/replaying of test results;
- Highly Extensible core.

Table 2.4 presents the pros and cons of WebLoad and JMeter.

TABLE 2.4: WebLoad vs. JMeter

	WebLoad	**JMeter**
Pros	▪ Native JS scripting ▪ UI wizards to enhance the script ▪ Many technologies are supported ▪ Good customer support	▪ Highly portable ▪ Supports all the Java based applications ▪ Less scripting efforts (user-friendly GUI) ▪ Simple charts and graphs to analyze statistics ▪ Supports integrated real-time
Cons	▪ Cannot support Citrix ▪ Cannot support SAP GUI ▪ Cannot support RDP and RTE	▪ Does not record HTTPS communication ▪ Does not intercept AJAX traffic ▪ Does not monitor Application server related statistics ▪ Framework of reports has limited features

2.4 Value Analysis

Value Analysis is an organized process of analysis and evaluation of a product or service. Its goal is to minimize the cost of a product/service, at the same time increasing its value without losing quality. It uses a specified group of techniques and functions to identify unnecessary costs that to a product or service do not improve its quality, efficiency, appearance, use and costumer satisfaction. Value is a subjective concept that is interpreted in different ways by costumers and also depends on the context. In general, value might be defined as a need, interest or preference. The creation of value is difficult to understand in theory, but it can be resumed as the balance between benefits and sacrifices in the costumers' perspective. These subjective benefits and sacrifices influence the value perceived by the costumer. Different costumers have different perceived values of the same products or services. So, the perceived value is the general evaluation that a costumer gives to a product/service utility, based on perceptions of what he receives and what he gives. Concepts as quality, customization, responsiveness, reliability, image and trust are seen as benefit drivers, while price, time/effort/energy and conflict are seen as sacrifice drivers. When these benefits and sacrifices are well combined or there is an considerable amount of benefits, and the costumer percepts advantage on an organization's offer, there is Value for the customer (VC).

2.4.1 New Concept Development (NCD) Model

The New Concept Development (NCD) Model is a relationship model that provides a definition of the key components of the Fuzzy Front End (FFE). It is composed by three key parts: the five key elements related to the Front End of Innovation (FEI) (inner area); the Engine that influences the five elements referred above and it is influenced by the leadership and culture of the organization; the Influencing Factors compose external factors that influence the organization, like distribution channels, costumers and competitors.

In Figure 2.13 there is presented the NCD Model with the key parts described above.

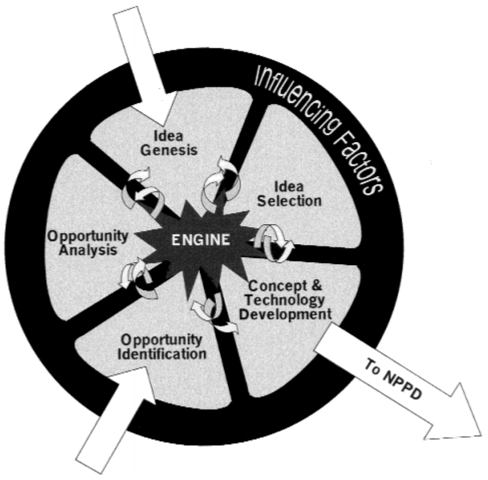


FIGURE 2.13: NCD Model (Koen et al. 2001)

The five key elements of the model are:

1. Opportunity Identification

   In the Opportunity Identification, the organization identifies the opportunities that may follow to achieve its goals. These opportunities can lead to a new business direction or upgrade to existing products. For this identification can be used two types of tools and techniques: formal and informal. Formal techniques include brainstorming, mind mapping, lateral thinking, causal analysis, fishbone diagrams, process mapping and theory of constraints. Informal techniques include ad hoc sessions, water cooler/cyberspace discussions, individual insights and edicts from senior management.

The implementation of the solution documented in this dissertation emerged from the problems identified in GLintt-HS related to errors detected in various test environments and, consequently, in production. Also, it is an opportunity to implement automation in different processes of the organization.

2. Opportunity Analysis

   In this element, additional information is analyzed and studied to understand the specific business and technology opportunities, how they really relate to the organization's strategy and culture and if they are worth following. Techniques used in this element include competitive intelligence and trend analyses.

   The automation of processes in the organization, like automated tests, is not really in its current culture. Despite that, this is a good opportunity to decrease bugs associated to the applications and to even decrease the time to release new features to production. This will increase the quality of products' releases and improve the organization's image.

3. Idea Genesis

   Genesis takes the opportunity studied and transforms it in a new concrete idea. The idea is not expected to be perfect at the time of its creation, it goes through many iterations where it is improved to correspond to discussions and analyses made around it. Activities in Idea Genesis include brainstorming sessions and idea banks.

   In Glintt-HS, the idea of implementing a robust CI/CD process with a good testing strategy emerged and was discussed several times by developers, testers and managers. Along the way, other requirements were added to the idea to resolve secondary related issues.

4. Idea Selection

   In this phase, considering many ideas emerged, it is important to decide the ones to pursue to achieve the most business value. Selection may include a simple individual choice or more formalized, as a prescribed portfolio method.

   To implement a CI/CD process with tests it is normal to exist different methods/approaches and technologies. It is important to analyze and compare these methods and technologies and to decide which are more adequate to the organization.

5. Concept and Technology Development

   In the final element, it is developed a business case based on the following factors: market potential, customer needs, investment requirements, competitor assessments, technology requirements and project risk. This phase may include a Technology Development Process to overcome technical uncertainty.

   After the selection of the correct methods and technologies to pursue, a business case is defined in Glintt-HS, through meetings with managers and stakeholders, to define the risks of the solution, costumer and market needs, investment and effort needed.

2.4.2 Benefits and Sacrifices

Tony Woodall presents a longitudinal perspective on VC on his article "Conceptualising 'Value for the Customer': An Attributional, Structural and Dispositional Analysis", as shown in Figure 2.14. He divides VC into four value temporal positions.

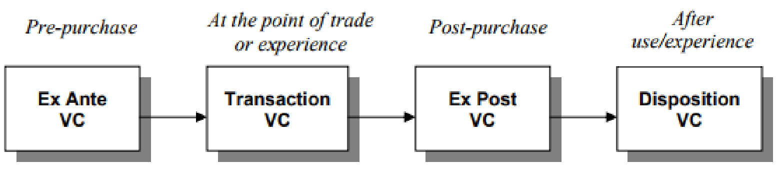


FIGURE 2.14: Longitudinal perspective on VC (Woodall 2003)

The solution described in this dissertation sums up as the improvement of Glintt-HS's Continuous Integration and Continuous Delivery processes. Table 2.5 presents the benefits and sacrifices from pre-purchase phase to after use/experience phase for costumers, in this specific case to Glintt-HS.

TABLE 2.5: Value Analysis - Benefits vs. Sacrifices

	Benefits	**Sacrifices**
Ex Ante VC	▪ Operational Benefits ▪ Product Benefits	▪ Human Energy ▪ Time ▪ Effort
Transaction VC	▪ Trust	▪ Monetary Cost ▪ Time
Ex Post VC	▪ Responsiveness ▪ Technical Competence	▪ Conflict (meetings about the process) ▪ Time/Energy/Effort (meetings as well)
Disposition VC	▪ Service Quality ▪ Time Saving ▪ Product Quality (improves Glintt-HS's product)	

2.4.3 Value Proposition

Value Proposition (VP) is essential to alert new costumers to the value of an organization's products and services. As Alexander Osterwalder wrote, it "is an overall view of a company's bundle of products and services that are of value to the customer." (Osterwalder 2004). VP should make clear what product/service you provide, the target costumers, the inherent value provided and its unique value. The service documented in this dissertation has as its target the organization Glintt-HS and at least one of its products. This service enriches the organization's CI and CD processes, including the implementation of a testing strategy. It allows the organization and the products where it is implemented to deliver products to its costumers with more quality, because they are strictly tested and consequently more bugs are prematurely detected and resolved. The planning and approach here documented is unique in the organization and it is expected to have good feedback and to make other teams implement a similar method. This solution can also be used by other organizations to implement similar processes on their environment, with a couple of adjustments to their business rules and specifications.

2.4.4 FAST Diagram

In Appendix A it is presented the FAST Diagram for the solution described in this document. This representation presents the the functions that need to be performed by the product/service into a How?/Why? relationship, in a logic and integrated way. It allows to analyze the logical relationships that are established between the different functions. This diagram provides a systemic vision of the service in analysis, being able to identify the critical path. The critical path represents the more important functions of the process, in other words, the functions that characterize the process. Besides showing the critical functions of the process, this diagram also shows supporting functions that although they are not in the critical path, their bad performance can harm the normal operation of the critical functions.

Having in mind those critical functions, for almost all of them there are tools that make their implementation easier. The all solution's process documented in this dissertation doesn't have a direct similar service, neither on the market or in the concerned organization. It is difficult to compare the overall process with others implement by another organization or developer. What it's possible to discuss and compare is the tools and approaches that can be used in each function of the process to achieve better results on the overall process. That is what is presented in the previous sections 2.3 and 2.2. These sections already give the comparison of tools to use in the CI/CD process and an idea of the best one for each function of the process. The tools and methods chosen to resolve each step of the process will also influence the value of the solution, so their analysis is very important to choose the most appropriated one to increase the value of the service.

It is also possible to compare this solution with the current state of CI/CD process, having in mind the benefits and sacrifices presented in Table 2.5. In the current state of the process, the automatic build only compiles the solution and executes the deployment of the resulted binaries. Many times this premature process leads to errors on the version of the application published. As this application's project interconnects with other organization's projects, the fix of those errors may take minutes or it may take days. The solution here documented will not let such thing happen, ensuring that a functional version of the application is running on the deployment machine. Besides the sacrifices to the organization presented above, as human energy, time cost and some monetary cost, the overall benefits are more valuable and the final result will clearly make up for the sacrifices. The solution analyzed will even increase the value of the products made by Glintt-HS to its costumers, because it will increase their quality, technical competence and responsiveness, and the trust of Glintt-HS's costumers in their products/services.

Chapter 3

Analysis

This chapter presents information about project *Atendimento*, the current state of CI/CD process at Glintt-HS and then an analysis and evaluation of the methods and approaches discussed in chapter 2, having as objective the conclusion of the best one to implement on the solution here documented.

3.1 Project *Atendimento*

Atendimento is an application developed at Glintt-HS, used in hospitals/clinics to help on the automation of the hospital management. This application facilitates the work done by Front-Office administrative technicians, that are responsible for performing the physical and telephonic attendance of patients. Also, with *Atendimento* they can create new patients' records, perform check-in and check-out of patients, schedule appointments and exams, monitor waiting times and respective lists, manage and print patient's documents and deal with the billing to each patient.

When the development of this application started, in 2013, its business concepts were not entirely new at the company. *Atendimento* is a version of an older application of the company, using more recent technologies. The older application, called GH, was implemented in Oracle Forms, while *Atendimento* is a Web application developed using HTML, JavaScript, Cascading Style Sheets (CSS)[1] and C# for the Web services. More recently, a technological conversion and the standardization of the Web pages' style have been performed, and TypeScript, LESS[2] and Gulp[3] were introduced in the development process. Since the start of the project until now, it is believed that at best 30% of the original application (GH) was converted to HTML. This means that, although *Atendimento* is an application with many features already, there is still a lot of work to be done and the project conclusion is not yet predictable.

As other projects of the company, *Atendimento* started during a time where managers at Glintt-HS didn't care at all about developers implementing tests to the code they were writing. The only type of tests performed at that time, and that still are done, were executed by a group of people on the application itself (functional acceptance tests). Nowadays, with other managers and developers more concerned about that issue, the mentality of the company has changed a little. It is still a subject ignored by some people, but it is now better and it is believed that it will improve with time. The solution documented here is also a chance to make *Atendimento* as an example to that change of mentality.

[1]https://www.w3schools.com/css/
[2]http://lesscss.org/
[3]https://gulpjs.com/

3.2 Current State of CI and CD

Nowadays, Glintt-HS has two branches for each application to manage the developments and bug corrections, besides other branches that represent product versions (for example branch 17). The DEV branch is changed by the developers daily and anytime with new features or corrections to existing ones. The CI branch is used to validate the integration of the changes made to the project in the DEV branch. This branch (CI) is always in a more stable state, because it is only changed in the end of each new development or in the end of any bug correction. Each development or bug has a work item associated that is created in the team's Backlog in Team Foundation Server. When developing or fixing a bug, each developer associates the specific work item to each check in he does to DEV branch. After the last check in associated to that work item is made, a script is executed to catch all the check ins associated to that work item and merged them to the CI branch. This script allows the developers to not waste time on putting their changes manually in the CI branch, decreasing the total development time of a feature or bug fix. Sometimes, the automatic merge executed by the script can result in conflicts that, in that case, must be resolved manually by the developer. Like in Visual Studio, a window is shown with the conflicts marked and the options to resolved them.

After changes are made to the CI branch, their integration in the project are tested in a TFS Build that is scheduled to run every day during the night. These TFS Builds are executed in a specific machine, that was configured to execute them. That configuration involves enabling the Build Configuration in the TFS Administration Console that was installed on that machine. In the Build Configuration, a connection to the Team Project Collection, where the sources of the projects of the organization are maintained, is made. Then, to execute builds on that machine it is required to create a Build Controller. This Build Controller manages a set of Build Agents that enable the execution of new builds. When creating a new build, through a Build Definition, the controller created on the remote machine is selected and one of its agents is assigned to that build. The same agent can be assign to more than one Build Definition, but it can't run more than one build at a time.

In that TFS Build, the first step is to get the latest version of the sources of the project from the CI branch of the Team Foundation Server. After that, a pre-build script is optionally executed. Normally, in that script dependencies from the project are installed, for example through npm (package manager for Node JS packages) or NuGet (package manager for Microsoft development platforms), and other builds from other projects can be run too. To run other builds from projects that the current project has dependencies, it is usually used the TFSBuild command line provided by the Team Foundation extensions and available from the Visual Studio installation.

After dependencies are installed, the solutions defined in the Build definition are compiled (clean and build) using the Microsoft Build Engine (MSBuild). In the Build definition, it can be defined arguments that will be used by the MSBuild, like the MSBuild Tools version, the Visual Studio version usually used to compile the project and the target .NET framework.

In the end of the build process, a post-build script is executed. In this script, the assemblies resulted from the build process are gathered in a folder that will be used to deploy the application. After the assemblies are ready to deploy, the content of this folder is deployed to a specific machine where the CI version of the application is published.

Concluded the deployment of the application, a group of testers manually perform acceptance tests to check if the developments and bug corrections satisfy their acceptance criteria. If an error or deviation from the correct behavior is detected, the development team is informed and must fix it. After correcting the error, the new changes are merged to the CI branch and a new build is triggered to update the published version of the application. Only when the testers team approves those changes in the CI version of the application, the developers can merge them to the production branch (17 branch for example).

3.3 Solution Analysis

In this section, the methods/approaches discussed in the previous chapter (section 2.2) are evaluated to conclude what is the best one for the current solution.

3.3.1 Approaches to build a CI/CD solution

The section 2.2.2 of the previous chapter presented different approaches to build a CI/CD solution. The first approach presented is a very premature CI process that is similar to what Glintt-HS has. The other two approaches are more robust, by including not only the integration of new changes into the project, but also the automation of a complex and important testing strategy. These two approaches only differ in the deployment phase (manual or automatic), so the following evaluation is focused on comparing the first approach with the overall process of the last two.

To evaluate these approaches the following metrics cans be considered:

- Time;
- Costumer satisfaction;
- Reliability.

The first metric is related to the all process of development, CI deployment and production deployment, the time that takes for a new feature for example to be available on all considered branches (DEV, CI and 17) and, consequently, to the costumer.

In the first approach considered to build a CI/CD solution, the time considered above can be easily increased if in the test phase the new feature or correction doesn't satisfy the acceptance criteria one or more times. This implies a new cycle of the all process, where the developer changes the code, push it to source control, merge it to CI branch and the build compiles and deploys it to CI environment. This process takes time and the new corrections take a while to be available at CI environment. Consequently, the time for the quality/testers team to approve these changes to 17 branch will also be longer, and will be later available to the costumer. The second approaches that include automated tests may increase the overall automated build time, but with the automatic verification of units, integration of modules and some acceptance criteria it is expected that the errors in CI environment will decrease. This doesn't mean that no problem will no longer be reported by the testers team, but it is also expected the decrease of those reports. In the overall process, the tests included will give more consistency to the versions deployed to CI environment and with less bugs detected it will be faster to make changes available to the costumer.

The second metric considered, costumer satisfaction, is related to the first one. As stated above, when an error is detected by the testers, it takes a while to correct it and make it again available for testers, and takes more time for them to approve the new features. This will also delay the deployment to production of the new changes, that influences costumer satisfaction a lot. In Glintt-HS, sometimes the delay can cause the renegotiation of delivery deadlines to the costumer, that decreases his satisfaction. The implementation of a new approach on the CI/CD process will allow to decrease the delivery time, obey to the deadlines and increase the product reliability. These factors will definitely increase the costumer satisfaction comparing to the current situation.

To evaluate the last metric, reliability, it is also needed to think about the integration of automated tests, defended in the two approaches different from the first used by Glintt-HS. This tests will allow to identify errors that the current approach can't detect in the phase of compiling the project's source code. The identification of errors through the automated tests will prevent the application from being deployed to the CI environment. It will only be deployed if all tests end successfully. This will increase the reliability of the application, by publishing it only when the things are done right and maintaining a good accuracy on its deployments.

Between the approach that does manual deployment and the other that does automatic deployment, there are not many differences and depends on the organization's preferences and business needs. This matter will be discussed and decided trough meetings with managers of the organization.

Chapter 4

Design

In this chapter, it is presented the design of the solution, being described in the following sections the choices made about the more adequate method and technologies and the design and description of the CD pipeline.

4.1 Choice of Method and Technologies

In this section, it is presented the decisions made about the method and technologies to use on the solution, having in mind the ones presented and discussed in the previous chapters.

About the methods and approaches, it was presented two alternatives to implement. The two processes were composed by a phase of compilation of source code, another for performing tests and the phase of deployment. The difference was in the automatic or manual deployment. To this solution it was decided, along with managers in Glintt-HS, that for now an approach with manual deployment will be implemented. There are a lot of check-ins of code made during one day of work and they simple don't want that every single one of them triggers a deployment. They prefer to join an acceptable number of changes and then deploy them manually.

The solution of Glintt-HS's problems will result in the development of a CD pipeline, that will be discussed on the next section. Currently, the project concerned has a build defined in TFS responsible for building the source code and deploying the application. In this solution, that TFS build will be reused to the first phase of the CI/CD process: compile the sources of the application. Only this process of compilation took a while to implement because of dependency of packages (npm/NuGet) and other organization's projects, so it would take a while to implement it on another CI Server. To implement the desired pipeline, Jenkins was chosen over TeamCity. The main reasons for choosing Jenkins are: the open source factor and the incredibly opportunities that its plugin repository provides. Related to Jenkins, there is another factor that makes the use of TFS, for the first phase of the process, important. The TFS version used in the organization is from 2013, and with that version the integrity with Jenkins has some limitations. A pipeline or any other type of build created in Jenkins that uses a project (or various) that are maintained in a TFS 2013 repository cannot be triggered at each check-in made to that repository. Jenkins does not have a plugin to detect a change in TFS and automatically start a build, as it has for Git[1]. TFS 2015 already has a service hook that allows to do a better integration with Jenkins and allows to triggers builds on each check-in. This limitation in the connection between TFS and Jenkins is also why it was decided to do the first phase of compilation of the source code in the TFS Build. In this build it is possible to define to trigger it at each check-in, what is expected in the documented

[1]https://git-scm.com/

solution. After that build is triggered and the sources are compiled, it is easy to trigger the Jenkins' pipeline through Jenkins CLI. This is a command line interface that allows to access Jenkins through a command line or shell script.

Regarding the test technologies, in chapter 2 some tools were presented for implementing unit tests, integration tests, deployment tests and functional tests. For c# unit tests, the three tools presented have his pros and cons (Table 2.1), none of them is really better than the other one. As the project *Atendimento* doesn't have any unit tests, the important part here is to implement them; the tool to use is not a big deal right know, it is important to start unit testing and the technology can be changed later. For know, it was decided that the Visual Studio Unit Testing Framework (MSTest) will be used, it already comes integrated in Visual Studio and it will be a easy start point. For TypeScript/JavaScript, comparing the pros and cons presented in Table 2.2, Jest seems to be a better choice to implement unit tests. It has less cons than the other tools, and the pros give it a good consistency. Also, Jest configuration/setup is easy to start implementing tests as soon as possible. Although Jest is seen as the better choice, it is detailed in chapter 5 the use of Mocha to implement unit tests in TypeScript. In that chapter, it is also explained why Jest was not used.

About the integration tests, the tools discussed to implement them in C# were the same approached for implementing unit tests. Therefore, Visual Studio Unit Testing Framework (MSTest) will also be used to implement the integration tests needed.

In the deployment tests phase, a limited number of functional acceptance tests will be executed, covering only the core features of the application. This will allow to give a fast response to people involved in the process about the success of the deployment. These tests that cover the main features of the application will be chosen from the all set of acceptance tests that will be implemented. The functional acceptance tests will be implemented using Selenium. This tool has a more complete set of built-in features that give a lot of possibilities to create the application's acceptance tests. Currently, in Glintt-HS another team of testers are developing functional acceptance tests using Selenium to cover for error in many applications, including *Atendimento*. This set of tests will be integrated in the new CI/CD process, being necessary to plan the best way to integrate them in discussions with the team that implemented them.

Considering the tools presented to perform nonfunctional acceptance tests, it was decided that WebLoad will be used. Comparing the pros and cons of this tool and JMeter, WebLoad seems to be the more adequate tool to use. Furthermore, it has a good integration with Jenkins that will also be used.

An overview of the new CI/CD process is presented in an activity diagram in Figure 4.1.

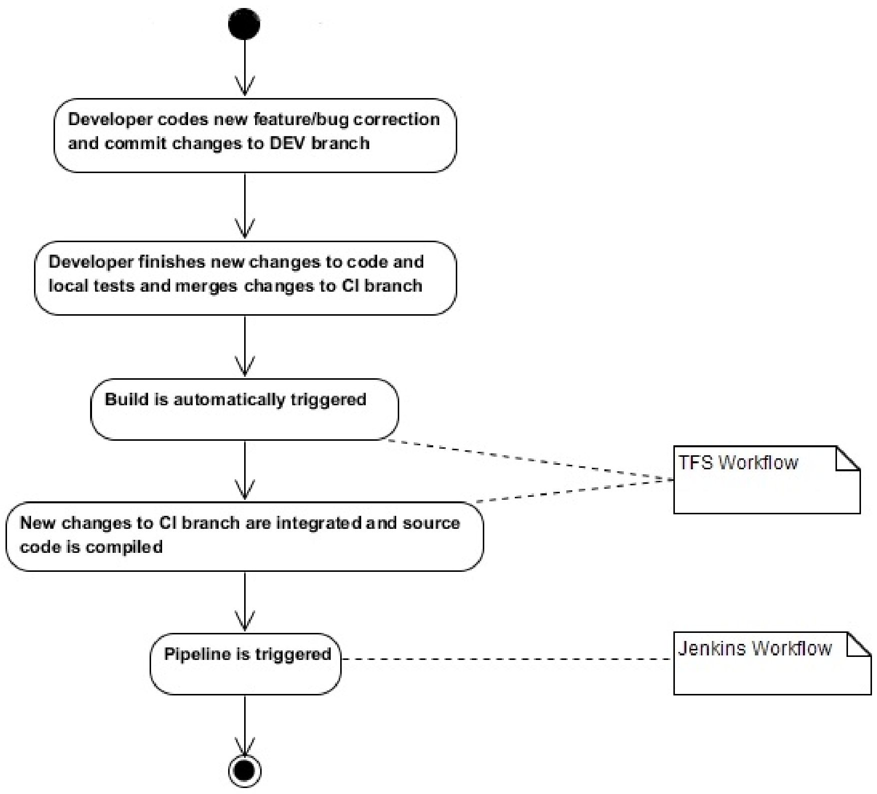

Developer codes new feature/bug correction and commit changes to DEV branch
Developer finishes new changes to code and local tests and merges changes to CI branch
Build is automatically triggered
New changes to CI branch are integrated and source code is compiled
Pipeline is triggered
TFS Workflow
Jenkins Workflow

FIGURE 4.1: Activity Diagram - process overview

The pipeline implemented in Jenkins workflow is detailed in the next section.

4.2 CD Pipeline

The Jenkins workflow is represented in the pipeline diagram in Figure 4.2.

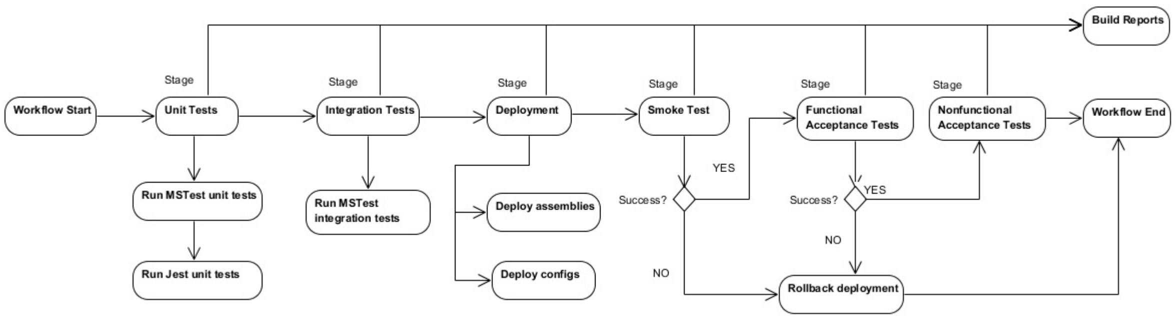


FIGURE 4.2: Pipeline Diagram (also in Appendix B)

The pipeline presented above is composed by the following stages:

- Unit Tests: in this stage, the unit tests developed (in C# e JS/TS are executed. To perform it, MSTest and Jest tools are used.
- Integration Tests: this stage executes the integration tests for c#, using also the MSTest tools as in the previous stage.

- Deployment: the Deployment stage consists in the processes to publish the application to the CI environment already mentioned in this document. This stage is composed by two sub-tasks: deployment of the assemblies resulted from the compilation of the source code - Dynamic-Link Library (DLL), JS and CSS files; deployment of configuration files that allow to run the application.
- Smoke Test: after the deployment finishes, a smoke test composed by functional acceptance tests of the core features of the application is executed. This test doesn't execute every functional acceptance tests, giving a more quick feedback about the success of the deployment. If this test fails, a rollback is performed; a deployment of the previous version of the application is made, restoring the last stable version. After the rollback, the workflow ends.
- Functional Acceptance Tests: in this stage, the acceptance tests developed using Selenium are executed. This tests cover more features of the application. If these tests fail, as the CI environment is supposed to be stable at any time, a rollback to the previous version is also performed. As in the previous stage, after this rollback is performed, the workflow ends.
- Nonfunctional Acceptance Tests: The final tests to be executed are the nonfunctional tests; these tests will evaluate the performance and capacity of the application. It was decided in the organization that for now the failure of these tests will not result in a rollback to a previous version, giving more importance to the smoke test and the functional acceptance tests. After this stage finishes, successfully or otherwise, the workflow ends.

Each build triggered will have an user input associated to the stage Deployment, in another words, the pipeline will stop before that stage starts and will wait for a user to choose if the pipeline continues or not. As said in the previous section, this is the approach of Continuous Delivery that for now will be implemented.

In every stage of the pipeline, as shown in the diagram, build reports are generated and can be seen after the pipeline ends or even during the process in the logs sent to the console output by Jenkins. If any of these stages fails, an email will be sent to the interested people, warning that a specific stage failed; if the stage were it failed is Smoke Test or Functional Acceptance Tests, it also informs that a rollback was performed. If the pipelines goes all the way to the end without failing, an email is also sent confirming the success of the deployment.

As an alternative to the pipeline presented above, in this process there could be another specific environment to run the functional acceptance tests. These tests would be made before the deployment to the CI environment, and the success or fail of the tests in the new intermediate environment would decide if the deployment to the last environment would be done. Since the CI environment is already a test environment and an intermediate to the 17 version internal environment, it was decided that the tests will be done in the CI environment as shown in Figure 4.2.

Chapter 5

Implementation

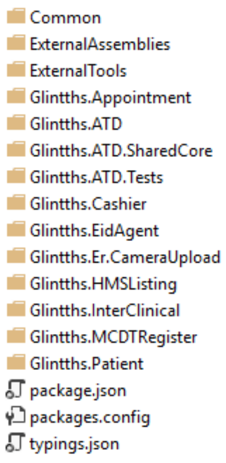
Considering the analysis of the best approaches and technologies presented earlier, and the design of the solution to the current CI/CD problem at Glintt-HS, this chapter describes the implementation of the designed solution. It is described some changes in the approaches/technologies used and the related reasons, and how they influence the final solution.

5.1 Overview

The implementation of the designed solution is divided in the following topics:

- Unit Tests;
- JavaScript/TypeScript Unit Tests;
- C# Unit Tests;
- Integration Tests;
- Functional Acceptance Tests;
- Team Foundation Server Workflow;
- Jenkins Workflow.

Before the above topics are presented, let's take a look at more information about project *Atendimento*. All the Web pages modules from the application are gathered in a single folder called Html, whose structure is observed in Figure 5.1.



Common
ExternalAssemblies
ExternalTools
Glintths.Appointment
Glintths.ATD
Glintths.ATD.SharedCore
Glintths.ATD.Tests
Glintths.Cashier
Glintths.EidAgent
Glintths.Er.CameraUpload
Glintths.HMSListing
Glintths.InterClinical
Glintths.MCDTRegister
Glintths.Patient
package.json
packages.config
typings.json

FIGURE 5.1: *Atendimento* Html Folder

The first three folders shown in that figure contain external assemblies used and common files to configure the application, and the Glintths.ATD.Tests folder contains shared methods used in tests. The other folders contain project solutions that represent a group of screens visible in the application and related logic. The most important ones are Glintths.ATD.SharedCore and Glintths.ATD, that are the base solutions of *Atendimento*. Glintths.ATD contains the base screens that exist for the longest time in the source code. The SharedCore folder contains screens, methods and models used in almost every other project of the Html folder. Also, there is a SharedCore project responsible for receiving requests triggered in the UI and to call the correct Web service. This project is approached again in the next sections of this chapter. As mentioned earlier, there is a lot of new features being developed and many others to be implemented yet. So, to the new developments, new folders in the Html folder have been created as it can be seen in the previous figure, to prevent the excess of projects and Web pages in a single solution like Glintths.ATD.

Besides those solutions referred above, *Atendimento* uses four different Web services components: Glintths.ATD.Services, Glintths.API.Internal, Gplatform and MuleSoft[1]. ATD Services are the oldest Web services used by *Atendimento* and their implementation started at the same time. This services are detailed in the next sections of this chapter. The other components are developed and maintained by other teams at Glintt-HS, being the API.Internal the oldest of those three components. Gplatform represents a microservice API and the last component provides different API's developed using MuleSoft. The Gplatform endpoints are important for the shared features of Glintt-HS applications like register of users, login and change of password, and to *Atendimento*'s basic features as search of patients, list of appointments by patient and patient admission. This component and its endpoints are many times the cause of errors emerged in the CI environment, that's why tests are developed to verify it, as presented in the following sections.

5.2 Unit Tests

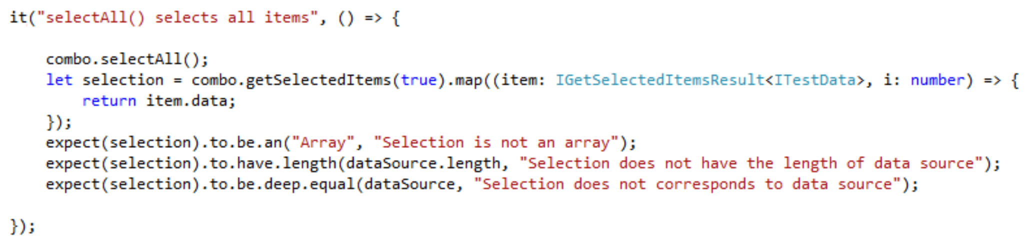
This section describes the development of the unit tests, subdivided in JavaScript/TypeScript tests and C# tests.

5.2.1 JavaScript/TypeScript Unit Tests

As mentioned in chapter 4, the first technology that was chosen to develop these tests was Jest, but eventually Mocha was the one used to implement unit tests. Although Jest has a simpler setup and configuration, which was a great advantage to start developing tests more quickly, there was a problem testing JS files used in project *Atendimento*. In all applications of Glintt-HS, these files are generated from TS files, being compiled as Asynchronous Module Definition (AMD)[2] modules. The AMD API is supported by RequireJS[3] and it is used for JS modules. It allows to format the JS module in a way that it is not necessary to manually write and order script tags with implicit dependencies. JS files compiled in AMD are not supported by Jest. So, instead of changing the way JS files are generated or using a plugin to convert those files to a definition known by Jest, it was decided that was better and simpler to change the technology to use. That's why Mocha was used.

[1]https://www.mulesoft.com/
[2]https://requirejs.org/docs/whyamd.html
[3]https://requirejs.org/

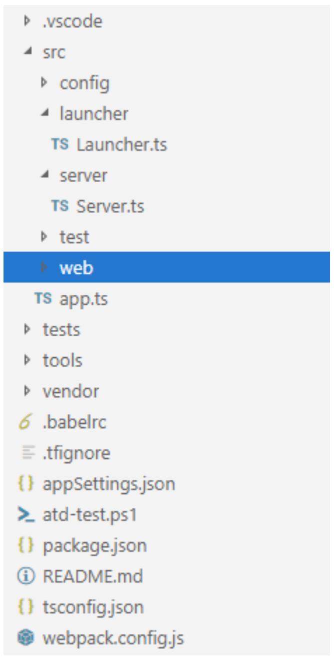
As said earlier, Mocha is a test framework that runs on Node.js and in the browser. Therefore, it was configured a Node.js application to run the necessary JS tests. With Mocha it is possible to use any assertion library, and in this project is used Chai[4]. This is a Behavior-Driven Development (BDD) / TDD assertion library that allows the developers to choose the most comfortable style. The TDD style - assert - provides the classic assert-dot notation and the BDD styles - should and expect - provide a readable style, being similar to a natural language. In this project, the expect style was used, making possible to any person to read and understand what is being tested. As observed in Figure 5.2, anyone can understand each expect of the example test. Also, there is an optional parameter that represents a message that is shown whenever the test fails, so it is easy to understand the behavior not expected.



```
it("selectAll() selects all items", () => {

    combo.selectAll();
    let selection = combo.getSelectedItems(true).map((item: IGetSelectedItemsResult<ITestData>, i: number) => {
        return item.data;
    });
    expect(selection).to.be.an("Array", "Selection is not an array");
    expect(selection).to.have.length(dataSource.length, "Selection does not have the length of data source");
    expect(selection).to.be.deep.equal(dataSource, "Selection does not corresponds to data source");

});
```

FIGURE 5.2: *Atendimento* Mocha Test

Every Node application can be triggered by command line using a JS file. As can be seen in Figure 5.3, that shows the source folder of the project implemented to execute JS tests, the Node application can be run compiling the app.ts file to JS and executing in the command line "node app.js". This file will use other files observed in that figure, like Launcher.ts, Server.ts, an "index.html" file present in "web" folder, and the files from config folder, to trigger a Node server that runs on a local url that can be accessed in a browser.



```
▸ .vscode
◢ src
  ▸ config
  ◢ launcher
     TS Launcher.ts
  ◢ server
     TS Server.ts
  ▸ test
  ▸ web
  TS app.ts
▸ tests
▸ tools
▸ vendor
  .babelrc
  .tfignore
  appSettings.json
  atd-test.ps1
  package.json
  README.md
  tsconfig.json
  webpack.config.js
```

FIGURE 5.3: *Atendimento* Mocha Tests - Project Folder

[4]https://www.chaijs.com/

The files used to test the JS/TS files of project *Atendimento* were added to the folder "tests", also observed in the previous figure. In the "web" folder, it was added a TS file to configure Mocha in the node application and to import the files of the referred "tests" folder. As the files to be tested depend on some global JS and CSS files developed in Glintt-HS, in the "index.html" file it was added tags to inject some of those script files into the run environment.

One more tool was used to help running this node application: Webpack[5]. "At its core, webpack is a static module bundler for modern JavaScript applications. When webpack processes your application, it internally builds a dependency graph which maps every module your project needs and generates one or more bundles." (Webpack 2018). With Webpack it is easier to resolve the dependencies of the JS/TS files and all that modules are exposed in one or more bundle files. In this project, two bundles are generated to facilitate the execution of the unit tests. To use Webpack, a "webpack.config.js" file is created at the root of the project, as can be observed in Figure 5.3.This file content is presented in Listing 5.1.

```
const appConfig = require("./tools/webpack.
    app");
const testConfig = require( "./tools/webpack
    .test");

module.exports = [ appConfig, testConfig ];
```

LISTING 5.1: *Atendimento* Mocha Tests - webpack.config.js

In the previous figure, it can be seen that the only thing that the main file for the webpack configuration does is exporting two other files - webpack.app.js and webpack.test.js. These two files are responsible for generating the two bundles that are mentioned above. Next, a snippet of the content of these two files is presented and explained. In Listing 5.2, it is presented a snippet of webpack.test.js.

[5]https://webpack.js.org/

```
const path = require("path");
const entryPoint = "./src/web/webapp.ts";
const outputDir = path.resolve(__dirname, '../dist/
    web');
const outputFile = "webapp.bundle.js";
const mode = "development";

var testConfig = {

    entry: entryPoint,

    output: {
        filename: outputFile,
        path: outputDir
    },
    mode: mode,
    devtool: "source-map",
    resolve: {
        extensions: [".js", ".ts"],
        modules: ["src", "../node_modules"]
    },
    module: {
        rules: [
            {
                test: /\.js$/,
                exclude: /node_modules/,
                loader: 'babel-loader'
            },
            {
                test: /\.ts$/,
                loader: "ts-loader"
            }
        ]
    }
};
module.exports = testConfig;
```

Listing 5.2: *Atendimento* Mocha Tests - webpack.test.js

Looking at the previous code, the most important configurations are the entry point and the output dir and file. In this case, Webpack is supposed to start processing dependencies between modules from the "webapp.ts" file. This file sets up Mocha in the Node application and imports the files created to test project *Atendimento*'s JS files (files from the tests folder, as mentioned earlier). The result of the bundling executed by Webpack is the output file - "webapp.bundle.js". Furthermore, the "webpack.test.js" contains configurations to compile JS and TS files with specific packages, and all these configurations are exported at the end of the file, so it can be imported in the main Webpack file (webpack.config.js).

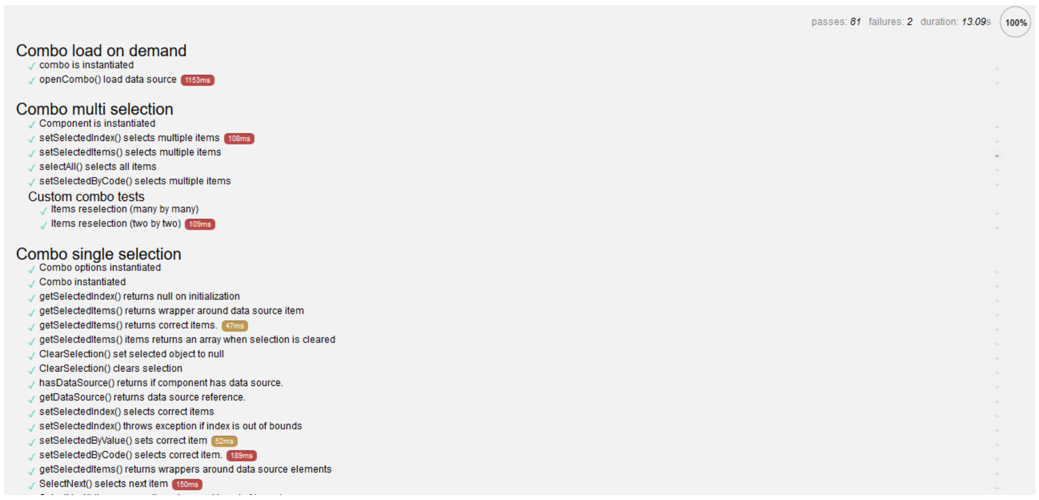
Below, it is presented a snippet of the webpack.app.js file in Listing 5.3.

```
const path = require("path");
const entryPoint = "app.ts";
const outputDir = path.resolve(__dirname, "../");
const outputFile = "index.js";
const mode = "development";

var appConfig = {
    entry: entryPoint,
    output: {
        filename: outputFile,
        path: outputDir
    },
    mode: mode,
    resolve: {
        extensions: [".js", ".ts"],
        modules: ["src", "node_modules"]
    },
    target: "node",
    module: {
        rules: [

            {
                test: /\.ts$/,
                exclude: /node_modules/,
                loader: "ts-loader"
            }
        ]
    }
};
module.exports = appConfig;
```

LISTING 5.3: *Atendimento* Mocha Tests - webpack.app.js

Like the previous snippet, the entry point and the output file are very important. In this configuration, "app.ts" is the first file that Webpack analyses to compile and resolve dependencies. This file, already mentioned in this section, is the start point of the Node application. With this configuration, Webpack generates a bundle file, "index.js". As the previous snippet, the "webpack.app.js" file specifies packages to compile JS and TS files and exports its configurations to be imported in the "webpack.config.js" file.

As mentioned earlier, the file "app.ts" is the one who starts the launching of the Node application, using the file "index.html" and other configuration files referred. Bundling all the JS/TS with the configurations described above, using the command "webpack", two JS files are created - index.js and webapp.bundle.js. To run the Node application, it is needed only one JS file, so in the "index.html" file a script tag to load the "webapp.bundle.js" was added. This way the generated file "index.js" contains all the information needed to create and run a Node application and to execute the implemented tests. Running the script "node index.js" in the command line launches the application in a local site, being used the default port 3000. In Figure 5.4, there is presented a screen shot of the Mocha tests of project *Atendimento*, executed in a browser.



```
passes: 81  failures: 2  duration: 13.09s  100%

Combo load on demand
  ✓ combo is instantiated
  ✓ openCombo() load data source 1153ms

Combo multi selection
  ✓ Component is instantiated
  ✓ setSelectedIndex() selects multiple items 108ms
  ✓ setSelectedItems() selects multiple items
  ✓ selectAll() selects all items
  ✓ setSelectedByCode() selects multiple items
  Custom combo tests
    ✓ Items reselection (many by many)
    ✓ Items reselection (two by two) 109ms

Combo single selection
  ✓ Combo options instantiated
  ✓ Combo instantiated
  ✓ getSelectedIndex() returns null on initialization
  ✓ getSelectedItems() returns wrapper around data source item
  ✓ getSelectedItems() returns correct items. 47ms
  ✓ getSelectedItems() items returns an array when selection is cleared
  ✓ ClearSelection() set selected object to null
  ✓ ClearSelection() clears selection
  ✓ hasDataSource() returns if component has data source.
  ✓ getDataSource() returns data source reference.
  ✓ setSelectedIndex() selects correct items
  ✓ setSelectedIndex() throws exception if index is out of bounds
  ✓ setSelectedByValue() sets correct item 52ms
  ✓ setSelectedByCode() selects correct item. 189ms
  ✓ getSelectedItems() returns wrappers around data source elements
  ✓ SelectNext() selects next item 150ms
```

Figure 5.4: *Atendimento* Mocha Tests - Results in the Browser (also in Appendix C)

In the previous figure, it is observed the results of the Mocha tests implemented, organized by groups and subgroups that represent JS modules and submodules, respectively. In the top right corner, it is presented the percentage of tests executed (100% in this example), the number of tests passed (81), the number of failures (2) and total duration of execution of tests. Also, if a test execution takes more time than the Mocha default patterns, that time is presented next to the name of the test and it is highlighted with the color yellow or red, being this last color marked in the tests that take the longest time. By now, these tests cover some shared files that are used by most of Glintt-HS applications and two base files of project *Atendimento*. The first files define front-end components that are used in various company applications, like datepickers, combo boxes, input boxes and lists with data. The files of *Atendimento* tested define general functions that are also used more than one time in the project. This means that it was decided that first it is important to test the general modules that are used all over the application and the ones which usually generate errors.

Issues developing JS/TS tests

While implementing the JS/TS unit tests, some difficulties emerged and that's what is discussed next.

Webpack is a very useful tool to compile JavaScript modules and resolve the dependencies between them. The use of this tool reduces the number of files needed to run the application to one file only. This result facilitates the developer's work, but it's configuration is not that easy. Webpack depends on specific configuration, forcing a developer to read and understand its documentation. Although it takes some time, the documentation is an advantage, because Webpack is well documented by its developers and other developers on the Web that have experience with this tool. In the end of the configuration, Webpack is a very powerful tool, but more recent JavaScript bundlers don't need much configuration as Webpack. In this project, this last tool is used because of its consistent documentation and support, and it was already used in the company, giving opportunity of having more help in the internal environment.

After running the tests in a browser using Node.js, there were some difficulties to run those same tests in the command line. As the solution described in this document plans the construction of a pipeline to run these and other tests, it was very important to be able to execute them in the command line. Running the tests via Node launches a site that after the execution is finished keeps active and watching for changes in the test files. In the pipeline context, it is necessary that the execution of the unit tests gives feedback when it is finished. Mocha installation comes with a Command-Line Interface (CLI) that can be used to run tests in a command line. Running the command "mocha 'tests/*.test.ts'" will execute every test.ts files of the tests folder. As this tests project is prepared to use Webpack, it is possible to use another tool: mocha-webpack[6]. This tool basically joins the functionalities of mocha CLI and Webpack. Having has example a test file "test.js" and an output file "output.js", mocha-webpack represents the command "webpack test.js output.js && mocha output.js", but in a more optimized way.

As suggested in mocha-webpack documentation, a Webpack configuration file for this tool must have some specific options that differ from the ones used when bundling files to run on a Web browser (Zinser 2017b). Therefore, a new Webpack configuration file was added to the tests project, named webpack-config-mocha.test.js. In Listing 5.4, there is presented a snippet of that new file.

```
var nodeExternals = require('webpack-node-
    externals');

module.exports = {
  target: 'node',
  externals: [nodeExternals()]
};
```

LISTING 5.4: *Atendimento* Mocha Tests - webpack-config-mocha.test.js

In addition to the options presented in the figure above, packages to compile JS and TS files (ts-loader and babel-loader) are also specified, like in the other configuration files presented earlier. About the new options added to the last file, "target" makes Webpack compile the files for Node.js. If this option is not specified, the code is compiled for Web environment and code-splitting[7] doesn't work. Code-splitting is a Webpack functionality that makes possible to split code into various bundle files, like what was made in this project and explained earlier. The other option "externals" uses a webpack-node-externals plugin, responsible for stopping Webpack from bundling files inside node_modules (folder that contains npm packages). This plugin is used because Webpack can't compile node_modules as server only modules and it also increases performance (Zinser 2017b).

After that configuration, another problem occurred running the mocha-webpack command. As the modules tested and its related files depend on Web browser variables, as window and document, the tests failed as soon as one of those variables appeared. To resolve this issue, as also suggested in the mocha-webpack documentation, the jsdom-global plugin was used (Zinser 2017a). This plugin creates a Web testing environment, making available the variables needed (as window and document). To import this plugin, the option "-r jsdom-global/register" is added to the mocha-webpack command. Following this, another issue related to variables exported to the window emerged. In the Node environment, its variables are changed and accessed using the variable "global" (global.Inputmask for example). The Web environment variables are mapped in the variable "window" (window.Inputmask for example). This means

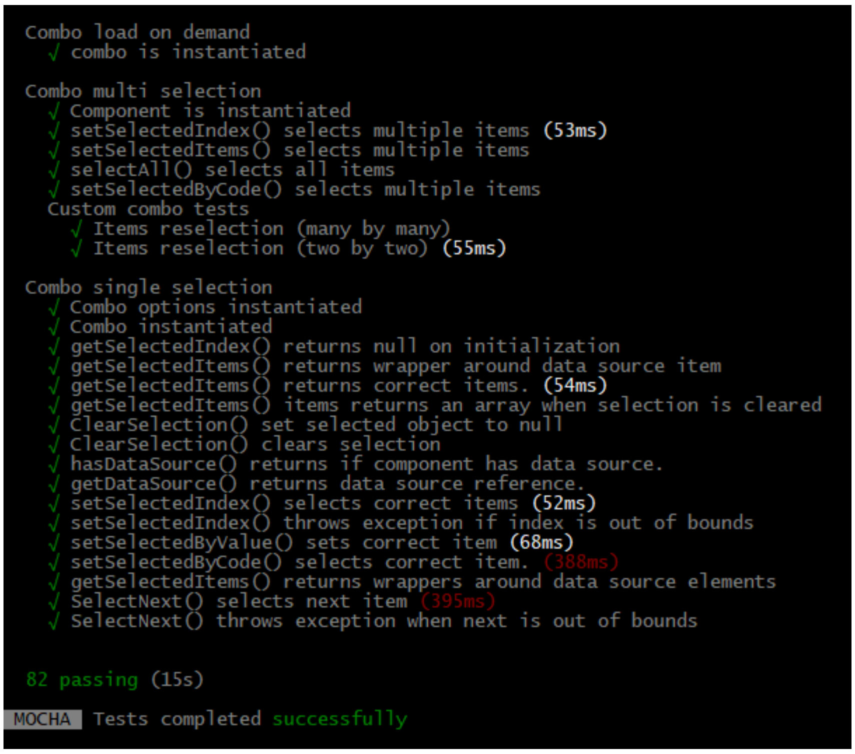
[6]https://github.com/zinserjan/mocha-webpack
[7]https://webpack.js.org/guides/code-splitting/

that if a file related to the tested modules tries to export a variable using "window" (window.Inputmask = ...), then the variable Inputmask is still undefined, unlike what happens in a Web environment. In Node, for this variable to be defined, the variable "global" must be used. To overcome these problems, a new JS file - mockBrowser.js - was created to be imported in the mocha-webpack command. Listing 5.5 presents the content of that file.

```
var jq = require("../../vendor/glintt/scripts/jquery
    -2.0.0.min");
global.jQuery = jq;
global.$ = jq;

global.navigator = {
    userAgent: 'node.js'
};
global.Option = window.Option;

require("../../vendor/glintt/scripts/jquery-ui/
    jquery-ui-1.10.3.min");
_ = require("../../vendor/glintt/scripts/lodash.min"
    );
require("../../vendor/glintt/scripts/infragistics.
    core");
require("../../vendor/glintt/scripts/infragistics.
    lob");
Mustache = require("../../vendor/glintt/scripts/
    mustache");
require("../../vendor/glintt/scripts/bootstrap.min")
    ;

let container = document.createElement("div");
container.className = "container";

for (var i = 0; i < 4; i++) {
    let row = document.createElement("div")
    row.className = "row";
    let root = document.createElement("div");
    root.className = "col-md-12";
    let rootBaseId = "root";
    root.id = i > 0 ? (rootBaseId + i) : rootBaseId
    ;
    row.appendChild(root);
    container.appendChild(row);
}
document.body.appendChild(container);

window = global;
```

LISTING 5.5: *Atendimento* Mocha Tests - mockBrowser.js

In the file presented above, some external needed modules are loaded, being some of them exported as global variables (like jQuery). Also, HTML required by some tests is constructed and added to the document variable of the test environment. Variable global is assigned to the variable window, meaning that any new variable exported using window (window.some_variable = ...) will now be exported to the global scope. Finally, running the command "mocha-webpack -r jsdom-global/register -r mockBrowser.js –webpack-config webpack-config-mocha.test.js 'tests/*.test.ts'" gives the same results of the browser, as shown in Figure 5.5.



```
Combo load on demand
  √ combo is instantiated

Combo multi selection
  √ Component is instantiated
  √ setSelectedIndex() selects multiple items (53ms)
  √ setSelectedItems() selects multiple items
  √ selectAll() selects all items
  √ setSelectedByCode() selects multiple items
  Custom combo tests
    √ Items reselection (many by many)
    √ Items reselection (two by two) (55ms)

Combo single selection
  √ Combo options instantiated
  √ Combo instantiated
  √ getSelectedIndex() returns null on initialization
  √ getSelectedItems() returns wrapper around data source item
  √ getSelectedItems() returns correct items. (54ms)
  √ getSelectedItems() items returns an array when selection is cleared
  √ ClearSelection() set selected object to null
  √ ClearSelection() clears selection
  √ hasDataSource() returns if component has data source.
  √ getDataSource() returns data source reference.
  √ setSelectedIndex() selects correct items (52ms)
  √ setSelectedIndex() throws exception if index is out of bounds
  √ setSelectedByValue() sets correct item (68ms)
  √ setSelectedByCode() selects correct item. (388ms)
  √ getSelectedItems() returns wrappers around data source elements
  √ SelectNext() selects next item (395ms)
  √ SelectNext() throws exception when next is out of bounds


82 passing (15s)

MOCHA Tests completed successfully
```

FIGURE 5.5: *Atendimento* Mocha Tests - Results in the Command Line

5.2.2 C# Unit Tests

As mentioned in section 2.2 of chapter 2, *Atendimento* is considered a midproject and a legacy system. At the current moment, this project has a lot of functionalities and being a conversion of an old application, as described earlier, it still has a lot of changes and requirements to be added. In the section referred, it is also explained that in a project of this type it is important to start covering and testing the code that traduces the most common, important and high-value features of the application. That said, like in the unit tests developed for JS, the goal of the first unit tests implemented in C# is to find possible errors in most common functions and in the ones that more often cause failures while running the application.

One of the main projects of *Atendimento*, called Glintths.ATD.SharedCore, contains a lot of functions that are used by other projects of the application. Therefore, the implementation of the c# unit tests started in this project. SharedCore also contains common used functions defined in JS that are tested using Mocha tests, as documented in the previous section. The Listing 5.6 presents a snippet of a unit test implemented in the new project Glintths.ATD.Core.UnitTests.

```
    /// <summary>
    /// This class tests Infragistics Filter Helper.
    /// </summary>
    [TestClass]
    public class IgFilterHelperUnitTest
    {
        /// <summary>
        /// Test method GetContainsFilterValue with
igFilter "contains(Glintt)"
        /// </summary>
        [TestMethod]
        public void GetContainsFilterValueTest_1()
        {
            var input = "contains(Glintt)";

            try
            {
                var result = IgFilterHelper.
GetContainsFilterValue(input);
                Assert.AreEqual("Glintt", result);
            }
            catch (Exception ex)
            {
                Assert.Fail(ex.Message);
            }
        }

        /// <summary>
        /// Test method GetContainsFilterValue with
igFilter "contains( Glintt)"
        /// </summary>
        [TestMethod]
        public void GetContainsFilterValueTest_2()
        {
            var input = "contains( Glintt)";

            try
            {
                var result = IgFilterHelper.
GetContainsFilterValue(input);
                Assert.AreEqual(" Glintt", result);
            }
            catch (Exception ex)
            {
                Assert.Fail(ex.Message);
            }
        }
    }
```

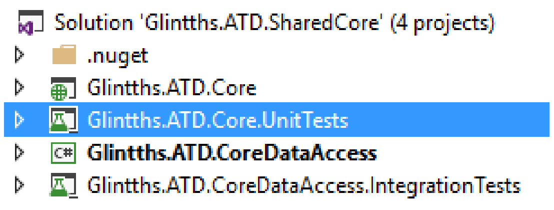
LISTING 5.6: *Atendimento* C# Unit Test - Glintths.ATD.Core.UnitTests

The tests presented in the above Listing are testing an helper class created in Glintths.ATD.SharedCore to get filters inserted by the user in form controls, like in a combo box. These controls, developed in Glintt-HS and approached in the previous section, are an interface to the Infragistics[8] UI Controls. In the combo box case, normally the user can insert some text to filter the datasource loaded. When that happens, a new request to the Web Service that returns the

[8]https://www.infragistics.com/

datasource is made with the text inserted as parameter. That text is passed through a contains expression, as can be seen in the tests implemented. These tests basicly check if the helper created is getting correctly the text inside the contains expression.

Figure 5.6 presents the new structure of the Glintths.ATD.SharedCore solution.

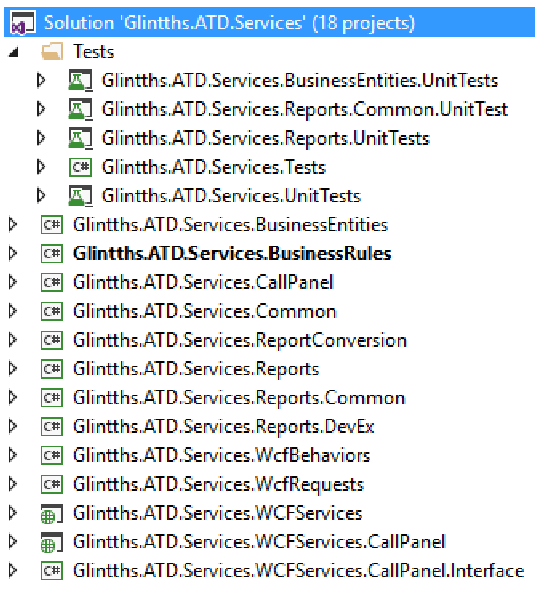


Solution 'Glintths.ATD.SharedCore' (4 projects)
.nuget
Glintths.ATD.Core
Glintths.ATD.Core.UnitTests
Glintths.ATD.CoreDataAccess
Glintths.ATD.CoreDataAccess.IntegrationTests

FIGURE 5.6: *Atendimento*'s Glintths.ATD.SharedCore Structure

As can be seen in the previous figure, the new project Glintths.ATD.Core.UnitTests was added to the intended solution, having the name of the project it is testing (Glintths.ATD.Core) as its partial name. Besides that, in the figure presented it is also possible to observe another project of the solution: Glintths.ATD.CoreDataAccess. This project basically consists on a bridge between the requests made by *Atendimento*'s projects controllers and the different services that the application depends on. Any request to a Web Service will first pass through CoreDataAccess and then move on to the correspondent service. That said, most of the code implemented in this project gets and changes information from the database. That's why in this case, a project of integration tests was created, as observed in Figure 5.6. Glintths.ATD.CoreDataAccess.IntegrationTests is approached next, in section 5.3.

Besides Glintths.ATD.SharedCore, there is one more solution where some tests were implemented: Glintths.ATD.Services. This solution represents the first Web Services ever used in project *Atendimento*. It contains very important projects with business entities and rules, template reports to present to the user in the application and two different Web Services projects. Figure 5.7 shows the structure of Glintths.ATD.Services solution.



Solution 'Glintths.ATD.Services' (18 projects)
Tests
Glintths.ATD.Services.BusinessEntities.UnitTests
Glintths.ATD.Services.Reports.Common.UnitTest
Glintths.ATD.Services.Reports.UnitTests
Glintths.ATD.Services.Tests
Glintths.ATD.Services.UnitTests
Glintths.ATD.Services.BusinessEntities
Glintths.ATD.Services.BusinessRules
Glintths.ATD.Services.CallPanel
Glintths.ATD.Services.Common
Glintths.ATD.Services.ReportConversion
Glintths.ATD.Services.Reports
Glintths.ATD.Services.Reports.Common
Glintths.ATD.Services.Reports.DevEx
Glintths.ATD.Services.WcfBehaviors
Glintths.ATD.Services.WcfRequests
Glintths.ATD.Services.WCFServices
Glintths.ATD.Services.WCFServices.CallPanel
Glintths.ATD.Services.WCFServices.CallPanel.Interface

FIGURE 5.7: *Atendimento*'s Glintths.ATD.Services Structure

The new tests implemented are part of the projects Glintths.ATD.Services.BusinessEntities.UnitTests, Glintths.ATD.Services.Reports.Common.UnitTest and Glintths.ATD.Services.Reports.UnitTests. The other two projects (Glintths.ATD.Services.Tests and Glintths.ATD.Services.UnitTests) represent some old test implementations that were not maintained by the developers. The two projects basically contain integration tests, although one of them has UnitTests in its name.

The first added project, Glintths.ATD.Services.BusinessEntities.UnitTests, contains tests that check constructors of business entities that receive as parameter an IDataReader object that has data obtained from the database. To mock this object, an excel file is used containing columns with the names of attributes to be read and values to fill the business entity's properties. Listing 5.7 shows one of the tests made in that project.

```
 [TestMethod]
     public void
TestMethodReader_GetDoctorAppointmentByPatient()
     {
          var fileName = Path.Combine(AppDomain.
CurrentDomain.BaseDirectory,
               "DoctorAppointment",
               "BusinessEntities",
               "Inputs",
               "HPA_GetDoctorAppointmentByPatient.xlsx");
          var dt = ConvertExcelToDataTable(fileName);
          var reader = dt.CreateDataReader();
          reader.Read();
          var result = new DoctorAppointmentEpisode(
reader, "DEMOPQR");

          Assert.IsNotNull(result);
          Assert.IsNotNull(result.MedicalAct);
          Assert.IsNotNull(result.AppointmentHour);
          Assert.IsNotNull(result.DoctorAppointEpisDate);
          Assert.IsNotNull(result.AppointmentDateState);
     }
```

LISTING 5.7: *Atendimento* C# Unit Test - Glintths.ATD.Services.BusinessEntities.UnitTests

Glintths.ATD.Services.Reports.Common.UnitTest has for now one test that checks the generation of a specific url prepared to be sent to a Glintt-HS program that sends the reports to a printer. Although the method tested is used almost every time in generation of reports, as it is defined in the Common project, a single test project was created. In Glintths.ATD.Services.Reports.UnitTests project, the generation of report urls that represent PDF files are tested. To perform these tests, data is mocked and passed to the respective methods that create the reports. In Listing 5.8, it is presented a test implemented in the last referred project.

```
 [TestMethod, TestCategory("New_Implementation")]
     public void TestMethod_NewImplementation_CSA()
     {
         // Assert value of key '
ATD_USE_SMALL_PATIENT_TICKET'
         OperationContext operationContext =
getOperationContext();
         var value = getReportType(operationContext);
         Assert.AreEqual(value, "CSA", "The value of the
 key 'ATD_USE_SMALL_PATIENT_TICKET' does not match the
test case.");

         getPatientTicketAndAssertTest(operationContext)
;
     }

 private void getPatientTicketAndAssertTest(
OperationContext operationContext)
     {
         string episodeId = "1282";
         string episodeType = "Consultas";
         string patientType = "HS";
         string patientId = "22";
         int copies = 1;

         string urlResult = string.Empty;
         try
         {
             var devEx = new ReportsDevEx();
             urlResult = devEx.
GetPatientTicketReportPdfUrl(operationContext, episodeId
, episodeType, patientType, patientId, copies);

             Assert.IsFalse(string.IsNullOrEmpty(
urlResult));
             Assert.IsTrue(Uri.IsWellFormedUriString(
urlResult, UriKind.Absolute), "Invalid Uri");

             openOrSavePdf(urlResult);
         }
         catch (Exception ex)
         {
             Assert.Fail(ex.Message);
         }
     }
```

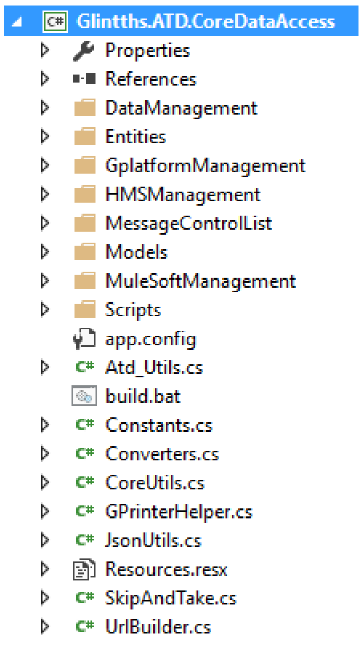
Listing 5.8: *Atendimento* C# Unit Test - Glintths.ATD.Services.Reports.UnitTests

The test presented above verifies the creation of a patient ticket to a specific context (CSA). As other contexts are tested, a general method to test patient ticket was created - getPatientTicketAndAssertTest. In this method the url generated is checked and using the method openOrSavePdf, that opens the url or saves the file it contains locally, validates its existence and that it has content.

5.3 Integration Tests

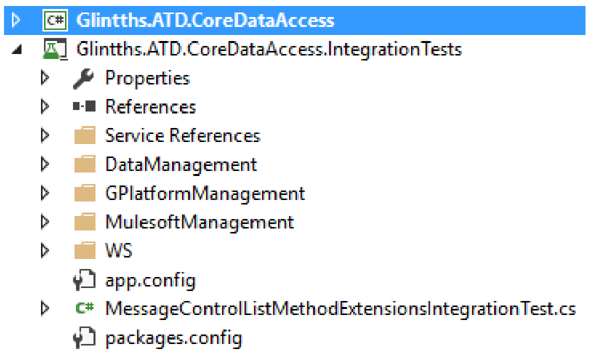
In section 4.1 of chapter 4 the technology to use in the implementation of integration tests is approached. As in the C# unit tests presented above, these tests were implemented using the Visual Studio Unit Testing Framework (MSTest).

In the previous section, it is presented the Glintths.ATD.SharedCore solution structure in Figure 5.6. As mentioned in that section, a new integration tests project was added to that solution - Glintths.ATD.CoreDataAccess.IntegrationTests - to test the code implemented in project Glintths.ATD.CoreDataAccess. Figures 5.8 and 5.9 present the internal structure of the projects Glintths.ATD.CoreDataAccess and Glintths.ATD.CoreDataAccess.IntegrationTests, respectively.



```
Glintths.ATD.CoreDataAccess
    Properties
    References
    DataManagement
    Entities
    GplatformManagement
    HMSManagement
    MessageControlList
    Models
    MuleSoftManagement
    Scripts
    app.config
    Atd_Utils.cs
    build.bat
    Constants.cs
    Converters.cs
    CoreUtils.cs
    GPrinterHelper.cs
    JsonUtils.cs
    Resources.resx
    SkipAndTake.cs
    UrlBuilder.cs
```

Figure 5.8: *Atendimento*'s Glintths.ATD.CoreDataAccess Structure



```
Glintths.ATD.CoreDataAccess
Glintths.ATD.CoreDataAccess.IntegrationTests
    Properties
    References
    Service References
    DataManagement
    GPlatformManagement
    MulesoftManagement
    WS
    app.config
    MessageControlListMethodExtensionsIntegrationTest.cs
    packages.config
```

Figure 5.9: *Atendimento*'s Glintths.ATD.CoreDataAccess.IntegrationTests Structure

The structure of the two mentioned projects is very similar, as the figures show. The folders that have "Management" as its partial name exist in the two projects containing files with methods to access differente Web Services. The files from the DataManagement folder connects to the older services of *Atendimento*, Glintths.ATD.Services; the GPlatformManagement and MulesoftManagement files call more recent Web Services, that are represented by a microservice API (GPlatform) and various API's implemented using MuleSoft. Particularly, in the integration tests project, the folder Service References has one reference to an older entities management service, developed and maintained by other teams of the company. In the folder WS, it is tested methods from the services contained in this last referred folder.

Listings 5.9 and 5.10 present, respectively, tests to a GplatformManagement method and to a MulesoftManagement file.

```
[TestClass]
  public class GplatformTokenHelperIntegrationTests
  {
      private HttpSessionStateBase sessionBase;
      private GplatformTokenHelper gplatformTokenHelper;
      [TestInitialize]
      public void Initialize()
      {
          Dictionary<string, object> customsessionData =
 new Dictionary<string, object>();
          customsessionData.Add(StoreManagerKeys.Username
 .ToString(), "ADMIN");
          customsessionData.Add(StoreManagerKeys.
 SecurityApplicationId.ToString(), "0");
          customsessionData.Add("USERID", "14");
          customsessionData.Add("tenant_id", null);
          sessionBase = SessionHelper.GetSessionStateBase
 (SessionEnvironmentsEnum.DEMOQPR, customsessionData);
          gplatformTokenHelper = new GplatformTokenHelper
 ();
      }

      [TestMethod]
      public void InitializeGPTokenTest()
      {
          try
          {
              string token = gplatformTokenHelper.
 InitializeGPToken(sessionBase);
              Assert.IsNotNull(token);
          }
          catch (Exception ex)
          {
              Assert.Fail(ex.Message);
          }
      }
  }
```

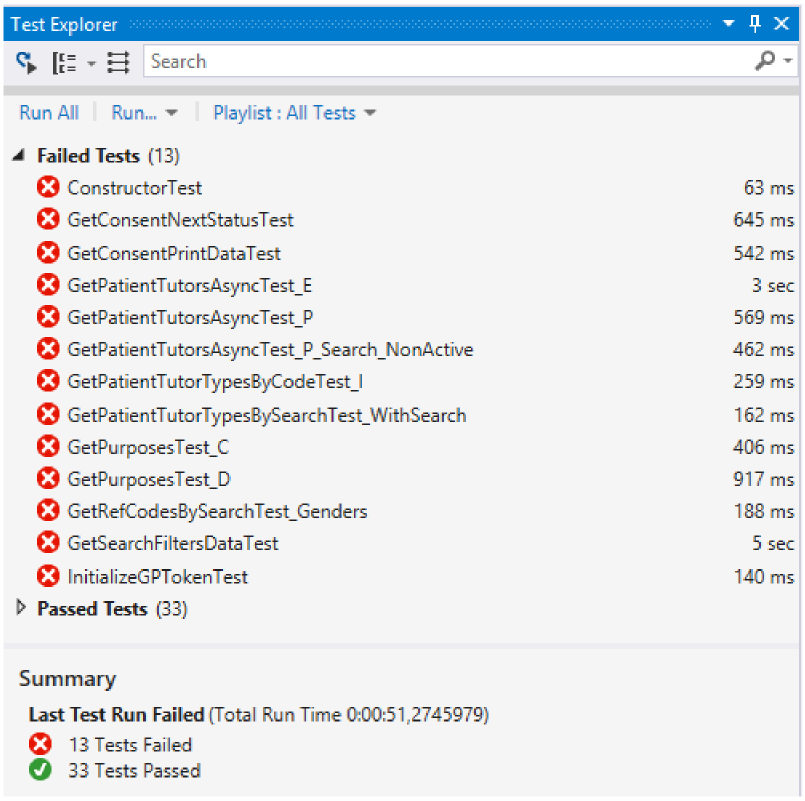
Listing 5.9: *Atendimento* C# Integration Test - GplatformManagement

```
[TestClass]
public class PatientManagementIntegrationTests
{
  [TestMethod]
      public async Task GetPatientTutorsAsyncTest_E()
      {
          try
          {
              string patientType = "HS";
              string patientId = "00000044711";
              string powerType = "E";
              DateTime? startDate = null;
              DateTime? endDate = null;
              int skip = 0;
              int take = 1;

              var result = await patientManagement.
 GetPatientTutorsAsync(patientType, patientId, powerType,
  startDate, endDate, skip, take);

              Assert.IsNotNull(result);
              Assert.AreEqual(1, result.Count);
              Assert.AreEqual(powerType, result[0].
 PowerType);
          }
          catch (Exception ex)
          {
              Assert.Fail(ex.Message);
          }
      }
}
```

LISTING 5.10: *Atendimento* C# Integration Test - MulesoftManagement

The first Listing contains a test of the generation of a new token to access the GPlatform API. Before the test is executed, an initialization of session data to be used in the generation token method is done. This session data includes the specific environment where the Web Services to test are running. In this case, the environment used is DEMOQPR, that represents the CI environment that should be tested. The other environments available test DEV and 17 versions. The second Listing has a test that checks the results of calling a MuleSoft service to obtain a list of tutors of a specific patient. It has a similar initialization to the previous test, therefore it was omitted this time.

In the next figure, Figure 5.10, it is presented the Test Explorer with the results of executing the integration tests using the DEMOQPR session environment.



Failed Tests (13)

ConstructorTest	63 ms
GetConsentNextStatusTest	645 ms
GetConsentPrintDataTest	542 ms
GetPatientTutorsAsyncTest_E	3 sec
GetPatientTutorsAsyncTest_P	569 ms
GetPatientTutorsAsyncTest_P_Search_NonActive	462 ms
GetPatientTutorTypesByCodeTest_I	259 ms
GetPatientTutorTypesBySearchTest_WithSearch	162 ms
GetPurposesTest_C	406 ms
GetPurposesTest_D	917 ms
GetRefCodesBySearchTest_Genders	188 ms
GetSearchFiltersDataTest	5 sec
InitializeGPTokenTest	140 ms

Passed Tests (33)

Summary

Last Test Run Failed (Total Run Time 0:00:51,2745979)

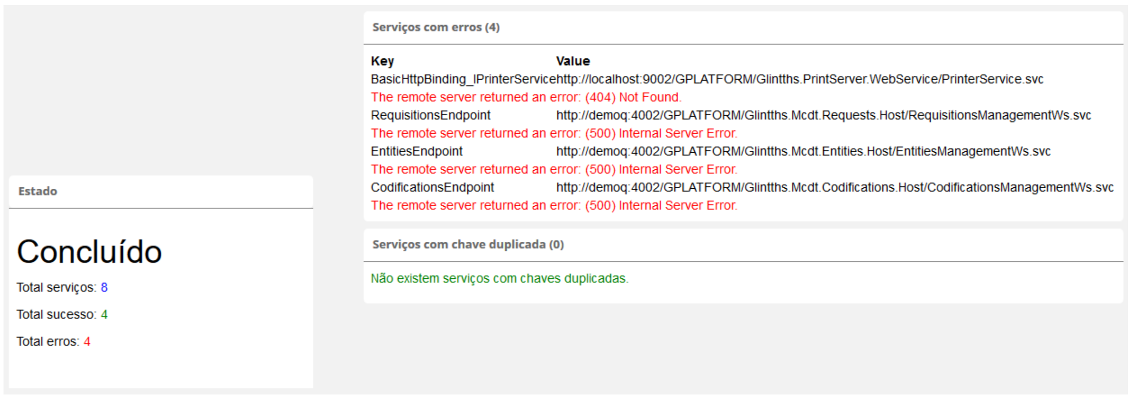
13 Tests Failed

33 Tests Passed

FIGURE 5.10: *Atendimento* C# Integration Test - Test Explorer

Beyond the project created and described above, as the application correct behavior depends on Web Services developed by other teams and that are accessed through its endpoints, a test Web page was developed to check the endpoints that the application relies on. Currently, this Web page tests if the endpoints are running and responding correctly (status code 200) and checks if in the configuration file of the application (Web.config) there are duplicated endpoints. If the endpoints are not responding correctly, they are listed in the Web page with the respective HTTP error code. This Web page and the tests it runs are very important, because one of the most common issues that occur in the CI environment is when the Web Services are down or not working correctly. This does not mean that their malfunction were caused by a deployment of *Atendimento*'s files, but it is important for detecting service errors as soon as possible and to inform more quickly back-end developers to check the errors.

These Web Service check tests are not yet integrated in the CI pipeline, because it is still being planned what is the best way to get page results automatically. Besides that, the referred Web page is already used by developers as one more resource to detect CI problems. Figure 5.11 presents an example of results generated by the created Web page.



Estado

Concluído

Total serviços: 8

Total sucesso: 4

Total erros: 4

Serviços com erros (4)

```
Key                              Value
BasicHttpBinding_IPrinterServicehttp://localhost:9002/GPLATFORM/Glintths.PrintServer.WebService/PrinterService.svc
The remote server returned an error: (404) Not Found.
RequisitionsEndpoint             http://demoq:4002/GPLATFORM/Glintths.Mcdt.Requests.Host/RequisitionsManagementWs.svc
The remote server returned an error: (500) Internal Server Error.
EntitiesEndpoint                 http://demoq:4002/GPLATFORM/Glintths.Mcdt.Entities.Host/EntitiesManagementWs.svc
The remote server returned an error: (500) Internal Server Error.
CodificationsEndpoint            http://demoq:4002/GPLATFORM/Glintths.Mcdt.Codifications.Host/CodificationsManagementWs.svc
The remote server returned an error: (500) Internal Server Error.
```

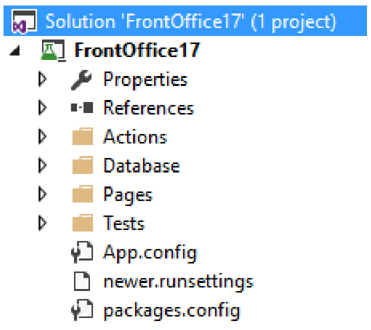
Serviços com chave duplicada (0)

Não existem serviços com chaves duplicadas.

FIGURE 5.11: *Atendimento* Web Service Checker

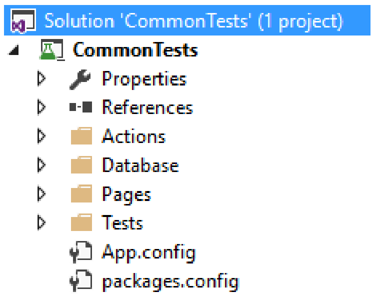
5.4 Functional Acceptance Tests

As mentioned earlier in section 4.1, Selenium was the chosen tool to develop functional acceptance tests to cover the *Atendimento* application. So, a new test project was created referring Selenium NuGet packages, as shown in Figure 5.12. Also, as some of the *Atendimento*'s screens are used by other applications at Gllintt-HS, a common tests project was created as it can be seen in Figure 5.13.



```
Solution 'FrontOffice17' (1 project)
◢ FrontOffice17
   ▷ Properties
   ▷ References
   ▷ Actions
   ▷ Database
   ▷ Pages
   ▷ Tests
     App.config
     newer.runsettings
     packages.config
```

FIGURE 5.12: Functional Acceptance Tests - Front Office Project



```
Solution 'CommonTests' (1 project)
◢ CommonTests
   ▷ Properties
   ▷ References
   ▷ Actions
   ▷ Database
   ▷ Pages
   ▷ Tests
     App.config
     packages.config
```

FIGURE 5.13: Functional Acceptance Tests - Common Tests Project

Atendimento is an application used by administrative technicians in an hospital/clinic, so it is also referred as Front Office like in the project presented in the first Figure. The common tests are referred in the front office project in their respective screen tests, to be executed side-by-side with the front office tests. These tests implemented using Selenium can be categorized, which is important to characterize them as Common, only Front Office or even Parametrized tests. Also, it is easier to differ Smoke tests from normal Functional ones. In Listings 5.11 and 5.12 there is presented a functional acceptance test developed to the *Atendimento*'s main page, where patients can be searched.

```
 [Test, Description("Search for a patient(s) by his
birthday date")]
        [Retry(nRetries)]
        [Category("Functional")]
        [Category("Common")]
        public void SearchByBirthdayDate()
        {
            if (!this.setupSuccessful)
            {
                throw new InconclusiveException("Test setup
 ended unsuccessfully.");
            }
            utils.CheckTestRetryCounter(this.lastTest, this
.retryCounter, out this.lastTest, out this.retryCounter)
;
            utils.SetWindowTitle(driver, TestContext.
CurrentContext.Test.Name);

            try
            {
                this.searchPatientsTestsCommon.
SearchByBirthdayDate();

            }
            catch (Exception e)
            {
                Assert.That(false, e.Message + "Failed test
, retrying");
            }
        }
```

Listing 5.11: *Atendimento* Functional Acceptance Test - Front Office Project

```
 public void SearchByBirthdayDate()
        {
            Assert.That(this.searchPatientsActionsCommon.
SearchByBirthday(searchPatientsPageCommon.
SearchPatientInput, searchPatientsActionsCommon.
CORRECT_BIRTHDAY_TO_TEST), "Search for a patient(s) by
his birthday date");
        }
```

Listing 5.12: *Atendimento* Functional Acceptance Test - Common Tests Project

As the Search Patients screen is a screen used in other applications, the Front Office project calls a method from the Common Tests. This common method calls a searchPatientsAction-sCommon method that writes in the search input the CORRECT_BIRTHDAY_TO_TEST, and then checks if all the patients loaded in the screen have that birthday date.

To run the functional acceptance tests, a Selenium-Grid[9] can be created. This grid allows to run the tests in parallel in different machines, operating systems and browsers. With the grid, it is also possible to run the tests against multiple versions of a browser and it reduces the completion of a test pass. Listing 5.13 presents the script used to start the Selenium-Grid. To start it, it is only needed the selenium-server-standalone jar file, the chrome driver and the internet explorer driver, all available online freely.

```
# start grid
$HUB = start -FilePath java -ArgumentList "-jar","E:\DOTNET
    -CI\ATD\Tests\functionalAcceptanceTests\Configuration\
    selenium-server-standalone-3.14.0.jar","-role","hub","-
    maxSession","8" -PassThru
$HUBID = $HUB.ID

$CHROME = Start-Process -FilePath java -ArgumentList "-
    Dwebdriver.server.session.timeout=180","-Dwebdriver.
    chrome.driver=D:\ChromeDriver\chromedriver.exe","-jar","
    E:\DOTNET-CI\ATD\Tests\functionalAcceptanceTests\
    Configuration\selenium-server-standalone-3.14.0.jar","-
    role","node","-browser","browserName=chrome,maxInstances
    =6","-hub","http://localhost:4444/grid/register","-port"
    ,"5556","-maxSession","6" -PassThru
$CHROMEID = $CHROME.ID

$IE = Start-Process -FilePath java -ArgumentList "-
    Dwebdriver.server.session.timeout=180","-Dwebdriver.ie.
    driver=D:\IEDriverServer\IEDriverServer.exe","-jar","E:\
    DOTNET-CI\ATD\Tests\functionalAcceptanceTests\
    Configuration\selenium-server-standalone-3.14.0.jar","-
    role","node","-browser","browserName=‘"internet explorer
    ‘",version=11,maxInstances=6","-hub","http://localhost
    :4444/grid/register","-port","5557","-maxSession","6" -
    PassThru
$IEID = $IE.ID
```

LISTING 5.13: *Atendimento* Functional Acceptance Tests - Grid Creation

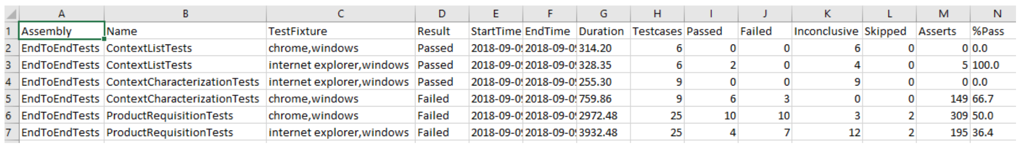
After starting the grid, it is possible to run the functional acceptance tests using the NUnit Console Command Line[10]. In Listing 5.14 it is possible to see some content of the batch file used to run the functional acceptance tests.

[9] https://www.seleniumhq.org/docs/07_selenium_grid.jsp
[10] https://github.com/nunit/docs/wiki/Console-Command-Line

```
nunit3-console ./FrontOffice17.dll --workers=4 --where "cat
     == "Smoke" && cat != Parameterized" --labels=All --
    params=ENVIRONMENT=DEMOQPR;
    PARAMETERSCONNECTIONSTRING_SERVER=10.250.39.126;
    PARAMETERSCONNECTIONSTRING_DATABASE=Parameters;
    PARAMETERSCONNECTIONSTRING_UID=adminqa;
    PARAMETERSCONNECTIONSTRING_PWD=adminqa;
    PARAMETERSCONNECTIONSTRING_SSLMODE=none
set exitcode=%errorlevel%

ren TestResult.xml TestResult%datetime%.xml
ParseTestResult TestResult%datetime%.xml
ReportUnit TestResult%datetime%.xml

move /Y Test*.* TestResults\%date%\%time%\
move /Y *.jpg TestResults\%date%\%time%\
```

LISTING 5.14: *Atendimento* Functional Acceptance Tests
- Execution Batch File

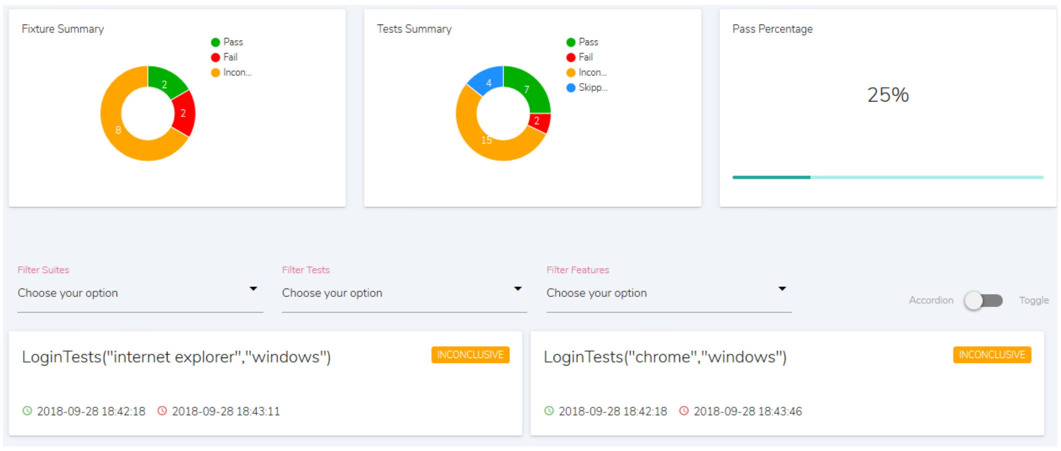
The example shown in the Listing above runs only smoke tests, the functional acceptance tests categorized as "Smoke", and the ones not Parameterized. Removing this "Smoke" filter, it runs all the functional acceptance tests not Parameterized. So, this filter is the difference in the scripts to run a smoke test and all functional acceptance tests in the Jenkins pipeline. After executing the tests, the XML file with the test results is used to create reports. The ParseTestResult is an utility created in Glintt-HS that creates a Comma-Separated Values (CSV) file containing information of the test results. It lists the test classes executed, its result and duration, its test cases and number of passed, failures and inconclusive. The other tool used, ReportUnit[11], generates an HTML report containing information of each test classes executed, filters by suites results, by tests results and by features (Smoke or Common for example). In Figures 5.14 and 5.15, it is presented an example of a ParseTestResult XML file and a ReportUnit HTML report, respectively.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Assembly	Name	TestFixture	Result	StartTime	EndTime	Duration	Testcases	Passed	Failed	Inconclusive	Skipped	Asserts	%Pass
2	EndToEndTests	ContextListTests	chrome,windows	Passed	2018-09-0	2018-09-0	314.20	6	0	0	6	0	0	0.0
3	EndToEndTests	ContextListTests	internet explorer,windows	Passed	2018-09-0	2018-09-0	328.35	6	2	0	4	0	5	100.0
4	EndToEndTests	ContextCharacterizationTests	internet explorer,windows	Passed	2018-09-0	2018-09-0	255.30	9	0	0	9	0	0	0.0
5	EndToEndTests	ContextCharacterizationTests	chrome,windows	Failed	2018-09-0	2018-09-0	759.86	9	6	3	0	0	149	66.7
6	EndToEndTests	ProductRequisitionTests	chrome,windows	Failed	2018-09-0	2018-09-0	2972.48	25	10	10	3	2	309	50.0
7	EndToEndTests	ProductRequisitionTests	internet explorer,windows	Failed	2018-09-0	2018-09-0	3932.48	25	4	7	12	2	195	36.4

FIGURE 5.14: *Atendimento* Functional Acceptance Tests - CSV Report (also in Appendix D)

[11] http://reportunit.relevantcodes.com/



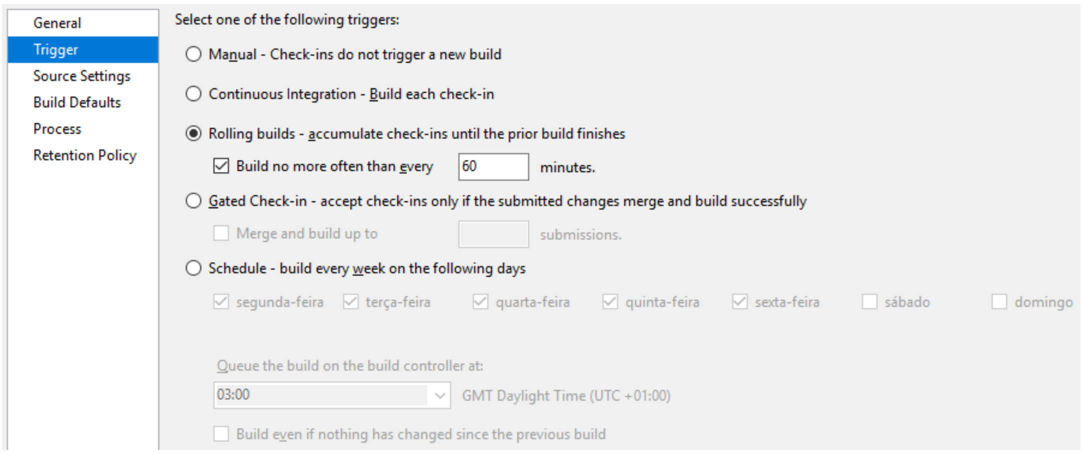
LoginTests("internet explorer","windows") INCONCLUSIVE
2018-09-28 18:42:18 2018-09-28 18:43:11

LoginTests("chrome","windows") INCONCLUSIVE
2018-09-28 18:42:18 2018-09-28 18:43:46

FIGURE 5.15: *Atendimento* Functional Acceptance Tests - HTML Report (also in Appendix E)

5.5 Team Foundation Server Workflow

The previous chapters of this dissertation mention that a CI premature process was already in progress at Glintt-HS. As mentioned in section 3.2 of chapter 3, a TFS build was created to automatically build the *Atendimento* project every night in a specific machine, only used to this purpose. The creation of a new build definition is possible in Visual Studio, in every user machine with enough TFS permissions to do it. After creating a new build definition, a new wizard is presented in Visual Studio. This wizard contains six different tabs to configure the build process. In the first tab (General), it is only defined the name of the build definition and its state (Enabled, Paused or Disabled). In the "Trigger" tab, it is decided how the build should be triggered, having the options shown in Figure 5.16.



Select one of the following triggers:

○ Manual - Check-ins do not trigger a new build

○ Continuous Integration - Build each check-in

[✓ ● Rolling builds - accumulate check-ins until the prior build finishes]

[✓ Build no more often than every] 60 minutes.

○ Gated Check-in - accept check-ins only if the submitted changes merge and build successfully

☐ Merge and build up to submissions.

○ Schedule - build every week on the following days

✓ segunda-feira ✓ terça-feira ✓ quarta-feira ✓ quinta-feira ✓ sexta-feira ☐ sábado ☐ domingo

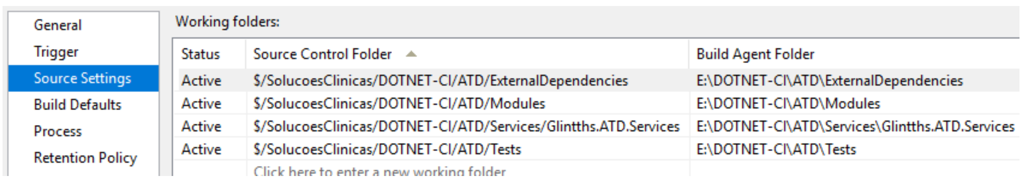
Queue the build on the build controller at:

03:00 GMT Daylight Time (UTC +01:00)

☐ Build even if nothing has changed since the previous build

FIGURE 5.16: TFS Build Definition - Triggers Tab

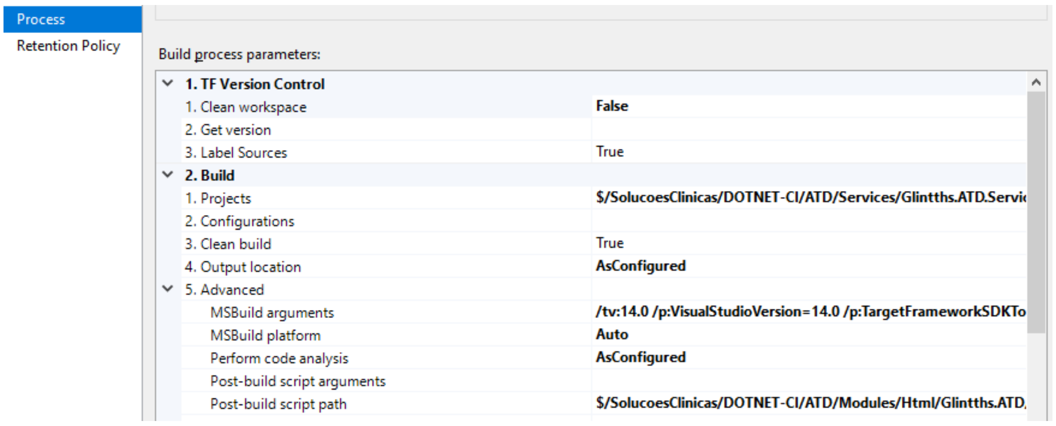
The TFS build, as can be seen in the previous figure, allows to choose five different options to trigger a new build. This build can be triggered manually, on each check-in (Continuous Integration), after accumulating check-ins, only accepting changes when they build successfully and on chosen days in a specific hour. Until this build was changed, it was scheduled to build every weekday at 03:00 AM. Now, looking to have a more quick feedback of changes made to the CI sources, check-ins are accumulated during one hour and after that time stamp a new build is triggered if changes were made. In the Source Settings tab of the build configuration wizard, it is defined the source control folders of the project and the respective folders of the remote machine to where they are downloaded. As presented in Figure 5.17, the different folders needed to build project *Atendimento* are shown in the Source Control Folder column, and the folder where they are maintain in the build machine are referred in the Build Agent Folder column.



Status	Source Control Folder	Build Agent Folder
Active	$/SolucoesClinicas/DOTNET-CI/ATD/ExternalDependencies	E:\DOTNET-CI\ATD\ExternalDependencies
Active	$/SolucoesClinicas/DOTNET-CI/ATD/Modules	E:\DOTNET-CI\ATD\Modules
Active	$/SolucoesClinicas/DOTNET-CI/ATD/Services/Glintths.ATD.Services	E:\DOTNET-CI\ATD\Services\Glintths.ATD.Services
Active	$/SolucoesClinicas/DOTNET-CI/ATD/Tests	E:\DOTNET-CI\ATD\Tests
	Click here to enter a new working folder	

FIGURE 5.17: TFS Build Definition - Source Settings Tab

After defining the source settings, in the Build Defaults tab the Build Controller to run the current build definition is chosen from a list of options. This list of options contains all controllers that at least were configured to run builds of the Glintt-HS TFS collection. As referred in section 3.2, to run builds in a remote machine, it is necessary to create a Build Controller through the TFS Administration Console. At this point, the basic configuration is concluded and the machine to execute the builds is chosen and configured. Next, it is necessary to decide what happens in the build process itself, that is made in the Process tab presented in Figure 5.18.



1. TF Version Control	
1. Clean workspace	**False**
2. Get version	
3. Label Sources	True
2. Build	
1. Projects	**$/SolucoesClinicas/DOTNET-CI/ATD/Services/Glintths.ATD.Servic**
2. Configurations	
3. Clean build	True
4. Output location	**AsConfigured**
5. Advanced	
MSBuild arguments	**/tv:14.0 /p:VisualStudioVersion=14.0 /p:TargetFrameworkSDKTo**
MSBuild platform	**Auto**
Perform code analysis	**AsConfigured**
Post-build script arguments	
Post-build script path	**$/SolucoesClinicas/DOTNET-CI/ATD/Modules/Html/Glintths.ATD.**

FIGURE 5.18: TFS Build Definition - Process Tab

The only required option to be filled in the Process tab is Projects. In here, only files with the extension .sln (Visual Studio Solution File) or .csproj (Visual C# Project File) are allowed, and they will be used to compile the respective solutions/projects in the build process. The first option seen, Clean workspace, it is recommended to be defined as False. This will configure the process to not clean the workspace and only replace the files that were changed since the last build. Also, although all paths configured in Source Settings tab consist in several folders created in the machine drive, the activation of the clean workspace flag can remove more files than expected, even files from other projects referred in other build definitions. Another rule presented in the previous figure, MSBuild arguments, is important to define specific options to be used by MSBuild to compile the solutions and projects mentioned earlier in this paragraph. In these arguments, it can be introduced the version of the MSBuild that should be used (14.0

for example) and the Visual Studio Version to consider (also 14.0 in the figure above). Also in the Build option of the current tab, there is the possibility of referring scripts (batch or powershell) saved in the source control that will be execute before and after the compilation of the projects. As referred in section 3.2, these scripts allow to install or update dependencies of projects before they are built and to gather the assembly files resulted from the compilation to immediately execute a deploy (in the old CI process) or various tests (in the improved CI process).

In the last tab, Retention Policy, it is defined the number of builds that should be maintained for each result status possible - Stopped, Failed, Partially Succeeded, Succeeded. For example, the default options are to retain the 10 latest builds that Failed, Partially Succeeded and Succeeded and the latest only that Stopped.

Comparing to the old CI process, in the TFS build it is possible to indicate some changes: the trigger of the build changed from schedule to rolling builds, as seen above; the source files of the various tests implemented were added to the source settings; the post-build script does not deploy the assemblies to the remote machine anymore, it triggers the Jenkins pipeline that is approached in the next section. The trigger of the pipeline is realized using the Jenkins CLI, that allows to run any job through the command line.

5.5.1 Issues implementing the designed solution

To configure a TFS Build environment in a remote machine, supposedly it is not needed to install Visual Studio IDE. As Microsoft suggests, if this IDE is not installed in any machine, Microsoft Build Tools provides enough tools to build managed applications (Microsoft 2015). In this perfect scenario, it wouldn't be necessary to install Visual Studio in the build machine. After installing the Build Tools and starting the compilation of the projects, errors appear on the first project built. Looking at the error messages, it seemed that .NET dependencies were missing in the MSBuild environment, and all the needed .NET framework version were also installed. After a long search and read of the build tools documentation and many forums with posts related to this problem, no clear solution was found. The only quick and known solution was to install Visual Studio IDE in the remote machine, allowing to have access to dependencies that only come with this installation.

As mentioned before in this dissertation, the improvement of the CI process here documented was expected to accomplish some basic theory aspects. In the premature process, the build was scheduled to run every week day during the night, and in this newer process it should be triggered on each check-in. Above it is shown in Figure 5.16 that the current build definition is not configured to trigger each check-in, but using rolling builds. This is justified by the fact that the merge of code from the DEV branch to the CI branch is made using a script, that on each changeset merged checks in the changes to the source control. This means that when working with work items as referred in section 3.2, merging a specific work item to the CI branch will (in most cases) consist on more than one changeset made to that branch. Therefore, each one of these changesets would trigger a new build if the designed type of trigger was used, and it will in most cases overload the remote machine with various builds in queue. The solution to this problem was debated and it consisted on only doing the check-in to the CI branch when all the changesets of the current work item where merged. At this point, as there are backlog work items that sometimes are dependent to others and the merge script allows to use more than one work item at a time, considering all the changesets of that work items ordered by date, there were some distrust on the team and managers to perform only one check-in for each time the merge script is executed. The lack of trust to make this change is understandable and has in mind keeping the CI environment in the best possible state. In the near future, it is expected that some simple tests are made to ensure that one check-in for merge won't result in lost of code and stability of the CI branch, and posteriorly it is expected that this policy of one check-in for merge is used across the development process and the TFS build is triggered on each check-in.

5.6 Jenkins Workflow

To implement the designed pipeline, Jenkins supports two different syntaxes: declarative and scripted. Groovy[12] was the language used to create the foundation of Jenkins Pipeline and both syntaxes have that language and the same Pipeline sub-system underneath. The main difference between the two syntaxes is the limits defined to the code that is implemented. Script pipeline doesn't define almost any specific limits, being the most code only limited by Groovy itself. On the contrary, declarative has a more strict and pre-defined code structure. Also, declarative has a simpler syntax and scripted allows to have a very free and subjective code structure, which can lead to newer and various problems that can have no feedback and pre-designed solution yet. With declarative, it is more certain to achieve what is required without having many issues and too different code alternatives. For those cons and pros, it was decided to implement a declarative pipeline. As it will be presented next in the Jenkinsfile code, other reason to use declarative pipeline is the existence of the post section to handle failures and errors, instead of the try-catch-finally structure used in scripted.

In Listing 5.15, it is presented a first snippet of the Jenkinsfile created, file that contains the code used in the pipeline and that is saved in source control.

[12]http://groovy-lang.org/

```
def currentStage = ''
def bodyEmail = 'Build successfully!!! \n\nDeployed at:
    http://demoapliis.glinttocm.com:6002/ATENDIMENTO_WEB/
    Glintths.Shell/Atd_Base'
def smokeTestStage = 'Smoke Test'
def stagesToRollback = [smokeTestStage]

pipeline {
    agent {
        label {
            label ''
            customWorkspace 'E:\\DOTNET-CI\\ATD'
        }
    }
    options {
    skipDefaultCheckout()
    disableConcurrentBuilds()
  }
    environment {
        ATD_CI_DIR = 'E:\\DOTNET-CI\\ATD\\'
        ATD_CI_BUILD_CONFIGS_DIR = 'E:\\DOTNET-CI\\ATD\\
    Modules\\Html\\Glintths.ATD\\BuildConfigs'
        CC_EMAILS = 'jcampos@glintt.com, tfaria@glintt.com,
     andre.d.santos@glintt.com, joao.c.ferreira@glintt.com,
    pedro.monteiro_ext@glintt.com, fabio.carneiro_ext@glintt
    .com'
    EMAIL = 'franciscoramalho@glintt.com'
        SUBJECT_EMAIL = 'ATD_CI Pipeline Result'
    }
```

LISTING 5.15: *Atendimento* Jenkinsfile - agent options and environment

In the snippet above, it can be seen that the code begins with a pipeline block, indicating that the declarative syntax is used. Before that block, there is defined some variables that will be read and changed in the pipeline. The declarative syntax does not allow to define these variables inside its blocks and directives, that's why they are written before the initial block. Inside the pipeline block, the agent environment where the job will be executed is configured, being set the custom workspace of the remote machine. In the options directive, it is defined that checking out code from source control should skipped, because in the TFS build all the source code needed is obtained. Also, it disables concurrent builds to prevent simultaneous access to the assemblies generated by the TFS build. The environment directive specifies read-only environment variables to be used all over the pipeline.

```
stages {
       stage ('Unit Tests') {
            failFast true
            parallel {
        stage ('JS/TS Unit Tests'){
          steps {
            script{
              currentStage = 'Unit Tests'
            }
            //JS/TS unit tests
            dir("${ATD_CI_BUILD_CONFIGS_DIR}") {
              powershell '.\\JsTests.ps1'
            }
          }
        }
        stage ('C# Unit Tests'){
          steps {
            //CSHARP unit tests
            dir("${ATD_CI_BUILD_CONFIGS_DIR}") {
              powershell '.\\CSharpUnitTests.ps1'
            }
          }
        }
            }
        }
```

LISTING 5.16: *Atendimento* Jenkinsfile - stage Unit Tests

The Listing 5.16 presents the beginning of the stages section and stage Unit Tests. In this stage, two child stages are executed in parallel, the JS/TS Unit Tests and the C# Unit Tests. The failFast option above the parallel expression forces the two stages to be aborted when one of them fails. Both child stages run a powershell script; the first runs a script that invokes mocha-webpack to execute the JS unit tests as explained before; the second uses MSBuild to compile the test projects and MSTest to execute them, as also described in section 5.2. In the source code above, it is also possible to see the use the script step. This step represents a scripted pipeline block executed in the declarative pipeline, and it provides a "escape hatch" as the definition and reassign of variables (like currentStage in the code). The next source code presented, in Listing 5.17, shows the code of the stage Integration Tests.

```
  stage ('Integration Tests') {
        steps {
            script{
                currentStage = 'Integration Tests'
            }
            //Integration tests
            dir("${ATD_CI_BUILD_CONFIGS_DIR}") {
                powershell '.\\IntegrationTests.ps1'
            }
        }
    }
```

LISTING 5.17: *Atendimento* Jenkinsfile - stage Integration Tests

The Integration Tests stage runs a powershell script that compiles all test project using MSBuild, and executes all tests through the MSTest tool.

```
    stage ('Deploy') {
            steps{
                script {
                    currentStage = 'Deploy'
                }
        mail bcc: '', body: "Build number ${env.
BUILD_NUMBER} ready for deployment.\nAccess http://devci
:8080/job/ATD_CI/ to confirm deployment or to stop the
build. \nPut your mouse on top of the stage that is
waiting for input and choose YES! to deploy or Abort to
stop it. \n\nWARNING: there may be other builds in
progress. Make sure you deploy or stop the right one
!!!", cc: "${CC_EMAILS}", from: 'gituser@glintt.com',
replyTo: '', subject: 'Deploy build', to: "${EMAIL}"

                input message: "Deploy build number ${env.
BUILD_NUMBER}?", ok: 'Yes!'

        //Archive all artifacts for needed rollbacks
        archiveArtifacts artifacts: 'Modules/Html/Glintths.
Shell/**/*.*', excludes: '**/*.config **/*.pdb'
        archiveArtifacts artifacts: 'Services/Glintths.ATD.
Services/Glintths.ATD.Services.WCFServices/**/*.*',
excludes: '**/*.config **/*.pdb'
        archiveArtifacts artifacts: 'Services/Glintths.ATD.
Services/Glintths.ATD.Services.WCFServices.CallPanel/**/
*.*', excludes: '**/*.config **/*.pdb'
        archiveArtifacts artifacts: 'Modules/Html/Glintths.
EidAgent/Glintths.EidAgent.WebServices/**/*.*', excludes
: '**/*.config **/*.pdb'
        archiveArtifacts 'Modules/Html/Glintths.ATD/Configs
/DeployConfigs/GIIS_CI/**/*.*'

                //deploy new version to CI
                powershell '''$drive = (get-location).Drive
.Name
                $deployMachine="\\\\demoapliis.glinttocm.
com"
                $user="install"
                $pass="*******"
                $deployMachineProjectFolder="GIIS\\CI"
        $projectFolder="$env:JENKINS_HOME\\jobs\\$env:
JOB_NAME\\builds\\$env:BUILD_NUMBER\\archive"
        $projectConfigsFolder="$env:JENKINS_HOME\\jobs\\
$env:JOB_NAME\\builds\\$env:BUILD_NUMBER\\archive\\
Modules\\Html\\Glintths.ATD\\Configs\\DeployConfigs\\
GIIS_CI"

                # Deploy dll and services to GIIS\\CI
                & "$($drive):\\DOTNET-CI\\ATD\\Modules\\
Html\\Glintths.ATD\\BuildConfigs\\DeployAssemblies.ps1"
-deployMachine $deployMachine -user $user -pass $pass -
deployMachineProjectFolder $deployMachineProjectFolder -
projectFolder $projectFolder

                # Deploy configs to GIIS\\CI
                & "$($drive):\\DOTNET-CI\\ATD\\Modules\\
Html\\Glintths.ATD\\BuildConfigs\\DeployConfigs.ps1" -
deployMachine $deployMachine -user $user -pass $pass -
deployMachineProjectFolder $deployMachineProjectFolder -
projectConfigsFolder $projectConfigsFolder'''
            }
        }
```

LISTING 5.18: *Atendimento* Jenkinsfile - stage Deploy

The Deploy stage, presented in Listing 5.18, begins with an email being sent to the *Atendimento*'s members informing that a new version is ready to be deployed to the CI environment. As designed, the deploy will not be executed automatically, therefore an input directive is used and the pipeline waits for a response to continue to the deployment or to abort the current job. If the user responds positively, the build continues and the current version artifacts are archived, in other words they are saved in the current build folder in the remote machine. These saved artifacts will be used to rollback new versions to this one, if it is necessary. After the archive, a powershell is executed to deploy the assemblies and config files to the CI publication folder.

```
stage ('Smoke Test') {
        steps {
            script {
                currentStage = smokeTestStage
            }
            dir("${ATD_CI_BUILD_CONFIGS_DIR}") {
                powershell '.\\SmokeTest.ps1'
            }
        }
    }
```

LISTING 5.19: *Atendimento* Jenkinsfile - stage Smoke Test

Listing 5.19 presents stage Smoke Test that also executes a powershell saved in source control. This script creates the Selenium-Grid explained in section 5.4 and runs the tests using NUnit console. In the end of the tests, the grid created is stopped and destroyed, so that it can be restored in the next stage of the pipeline and to release resources of the remote machine. The nest stage, Functional Acceptance Tests, presented in Listing 5.20 runs another powershell script that revives the Selenium-Grid and executes a larger set of Selenium tests.

```
stage ('Functional Acceptance Tests') {
        steps {
            script{
                currentStage = 'Functional Acceptance
Tests'
            }
            //Functional Acceptance tests
            dir("${ATD_CI_BUILD_CONFIGS_DIR}") {
                powershell '.\\
FunctionalAcceptanceTests.ps1'
            }
        }
    }
```

LISTING 5.20: *Atendimento* Jenkinsfile - stage Functional Acceptance Tests

The final snippet of the Jenkinsfile, presented in Listing 5.21, shows the final section of the pipeline, already outside the stages section. The post sections configures additional and optional steps that are executed after the conclusion of the pipeline or one of its stages. To execute this post steps, a specific condition must be verified. In the snippet presented, if the pipeline or any stage finishes with "failed" status, an email is sent to *Atendimento*'s team members informing the stage that failed and the console log of the failed build. Also, if the failure of the current stage should result in a rollback to the previous successful version, a powershell script is executed to deploy the artifacts of the last successful build. When the pipeline ends successfully, an email is also sent to the members of *Atendimento*, informing the result of the build and the link to the CI environment where they can see the more recent version of the application.

```
post {
      failure {
          script {
              bodyEmail = "The build failed on stage (${
  currentStage}). \n" +
                          "\nReport at: http://devci
  :8080/job/ATD_CI/${env.BUILD_NUMBER}/console"
      if(stagesToRollback.contains(currentStage)){
         //rollback
                   powershell '''$drive = (get-location).
  Drive.Name
                       $jenkinsHome="$env:JENKINS_HOME"
                       $jenkinsJobName="$env:JOB_NAME"

                       & "$($drive):\\DOTNET-CI\\ATD\\
  Modules\\Html\\Glintths.ATD\\BuildConfigs\\Rollback.ps1"
   -jenkinsHome $jenkinsHome -jenkinsJobName
  $jenkinsJobName'''

          bodyEmail += "\n\nRollback to the previous
  successfull version executed!"
      }
              mail bcc: '', body: bodyEmail, cc: "${
  CC_EMAILS}", from: 'gituser@glintt.com', replyTo: '',
  subject: "${SUBJECT_EMAIL}", to: "${EMAIL}"
          }
      }
      success {
          mail bcc: '', body: bodyEmail, cc: "${CC_EMAILS
  }", from: 'gituser@glintt.com', replyTo: '', subject: "$
  {SUBJECT_EMAIL}", to: "${EMAIL}"
      }
 }
```

LISTING 5.21: *Atendimento* Jenkinsfile - post section

5.6.1 Issues implementing the pipeline

While implementing the pipeline in the Jenkins workflow, there were some issues that required online search to resolve them. One of the first problems that emerged was the difficulty to access environment variables, both the specific variables of Jenkins and the variables initialized in the environment section. As mentioned in Jenkins documentation, to access environment variables $env.VARIABLE_NAME or just $VARIABLE_NAME - for example $env.BUILD_NUMBER or $BUILD_NUMBER. Using the pipeline syntax, to do a simple log of one of those variables is just needed a command "echo '$env.BUILD_NUMBER' ". With this command, the pipeline doesn't really prints the value of the environment variable. The problem is related to the use of single quotes (') instead of double quotes ("). If the command is changed to "echo "$env.BUILD_NUMBER" ", it will then print the desired value. Besides that issue, in the powershell step the referred variables are not accessed using the same notation. In the powershell context, the variable environments are read through

$env:VARIABLE_NAME, which can result in confusions to when to use one of the expressions mentioned.

In the Deploy stage, there is an input step to check if the deployment of the artifacts of the current build is approved. As this stage runs two different scripts, one that deploys the assemblies and other that deploys configuration files of the application, these scripts could be run in parallel. At this point, it was not possible yet to configure two parallel stages with an input step.As mentioned in Jenkins documentation, a stage must have only one of the following sections: steps, stages or parallel. In the future maintenance of the pipeline, it is expected that this issue is resolved and the pipeline time execution to be improved.

Chapter 6

Evaluation

Considering the importance of the preconized solution and people interested in its good implementation, some methods can be used to evaluate it.

The factors that will be used to evaluate the solution are:

- User Satisfaction (people involved in the process);
- Average Time for CI Approval.

User Satisfaction will evaluate the use of the new solution to the CI/CD process by the people involved: developers, testers, managers. This group of people is in the first line to deal with the process and is the one that can give a more important and reliable feedback. To test this factor, an inquiry will be created and provided to each person of that group to give his opinion about the new process implemented. Having in mind that the group of respondents have different functions on the organization and not all of them have technical knowledge about each stage of the CI/CD process, it is expected that the inquiry will have more general questions and at the same time will contribute to a reliable evaluation. This inquiry is expected to be organized with only level answers (0 to 5 for example) to allow a more objective analysis of the factor in question. The results of the inquiry will be analyzed and will contribute to understand the feedback of the parts interested in the process.

The hypotheses to test User Satisfaction include the hypothesis of the involved people in the process being more satisfied and confident with the new implemented process. The null hypothesis to consider is the acceptable average score for each question of the inquiries made, and that average number was discussed and defined with this project's supervisor and interested managers. Regarding statistic tests, the appropriate test to use is approached in the next sections, in this case in section 6.1. The answers to the inquiry, organized by levels (0 to 5 for example), are analyzed and the average for each question is compared to the acceptable average score defined. This comparison allows to conclude if people involved in the process are happier with its new implementation.

Regarding Average Time for CI Approval, it will evaluate the time that takes the testers team to approve a development published in the CI environment. It will compare the average time in the previous CI/CD state and in the new one. To do that comparison, it will be considered a specific number of sprints occurred in the previous state of CI/CD process and the same number of sprints occurred in the new state of the process. In each sprint, it will be calculated the average time from the moment that the developer deploys his development to CI environment until testers approve it. It is easy to calculate that time for each development in a sprint because in project *Atendimento* we use the product backlog available in the TFS. When a developer deploys his development to the CI environment, he creates a new task in the backlog item associated to his development called *Testes Qualidade CI* (Quality Tests CI), with the "To Do" state. When a task with this description is created, the testers team automatically receive a notification saying that they can test the related development. When a tester starts validating, he/she changes the task state to "In Progress" and when he/she approves that development, the task state is changed to "Done". So, to verify the time it took to approve that development it is just needed to calculate the time since the task state was changed to "In Progress" until it was changed to "Done".

The hypotheses to test this factor include the hypothesis of the new process to have less average time for CI approval than the old organization's process. Since this time for each development can take hours and sometimes days, this is a quantitative analysis, that will be traduced in hours. To evaluate this factor, the appropriate test is also explained in an upcoming section, section 6.2.

6.1 User Satisfaction

To evaluate user satisfaction, an inquiry was created to be answered by the people involved in the process (developers, testers, managers). This inquiry is presented in Appendix F. Of the group of people considered, 10 elements answered to all the five required questions. Table 6.1 presents the results of the inquiry made, being shown the number of people that answered with each option for each question.

TABLE 6.1: User satisfaction - number of answers by question.

Questions	Answers				
	1	2	3	4	5
1	-	-	1	1	8
2	5	4	1	-	-
3	1	-	1	5	3
4	-	-	1	7	2
5	-	-	1	4	5

By adding the numbers on each row, it can be confirmed that 10 persons answered the inquiry. The hyphen (-) in some of the table cells indicates that nobody chose the related answer option. For each question observed in Appendix F and enumerated in table 6.1, it is possible to compare the obtained results with the expected ones.

Before starting the statistical analysis of any metric, it is necessary to say that a tool was used to perform the appropriated tests: RStudio [1]. It is an IDE that allows to use the programming language R[2] to create graphics and perform statistical calculations.

In the next subsections, the answers to each question of the inquiry are evaluated.

[1]https://www.rstudio.com/

[2]https://www.r-project.org/

6.1.1 Question 1

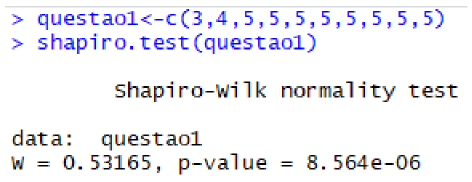
Analyzing the results of the first question of the inquiry - "What do you think about the use of automated tests besides the ones performed by QA team"? -, it is observed that the most of the respondents (80%) believe that the addition of automated tests to the ones performed by the QA team is very good. The remaining 20% is divided between option 3 and 4, so one person in ten is still not sure about the use of automated tests.

As mentioned earlier, for each question of the inquiry it is calculated the mean of the answers and it is compared to a value defined as expected. Regarding statistic tests, t-test is a hypothesis test that allows to compare means. To guarantee that the t-test is the adequate test to perform, some assumptions must confirmed. The first assumption defines that the data being analyzed must be continuous (not discrete), and this point is verified in the answer options (0-5). Another assumption mentions that the sample considered is a simple random sample, which means that each item in the population has the same probability of being selected. Last but not least, the data analyzed must follow the normal probability distribution. Normally, when analyzing a sample with 30 or more observations, the sampling distribution is assumed to be normal. In this particular case, the sample has 10 observations, so it is not possible to make that assumption. To assume the normality, a test must be made; a very known and powerful test is the Shapiro-Wilks test. This test consists in two hypotheses:

H_0: Population follows a normal distribution

H_1: Population does not follow a normal distribution

To perform the referred test, RStudio was used, as it can be seen in Figure 6.1.



```
> questao1<-c(3,4,5,5,5,5,5,5,5,5)
> shapiro.test(questao1)

        Shapiro-Wilk normality test

data:  questao1
W = 0.53165, p-value = 8.564e-06
```

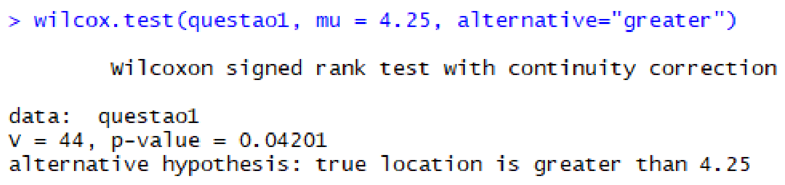
FIGURE 6.1: RStudio - Shapiro-Wilks test for question 1

Considering a confidence interval of 95%, which means the significance level (α) is 0.05, the p-value presented in the above figure is less than α. Therefore, the normal distribution of the data considered is rejected and the t-test cannot be used. An alternative to the t-test in this case can be the Wilcoxon test, that is a non-parametric hypothesis test. To evaluate the results of the first question of the inquiry, it is defined that the average should be greater than 4.25. Therefore, the hypotheses to the current test are:

H_0: m $\leq$ 4.25

H_1: m $>$ 4.25

where m represents the mean of the sample



```
> wilcox.test(questao1, mu = 4.25, alternative="greater")

        Wilcoxon signed rank test with continuity correction

data:  questao1
V = 44, p-value = 0.04201
alternative hypothesis: true location is greater than 4.25
```

FIGURE 6.2: RStudio - Wilcoxon test for question 1

Figure 6.2 presents the result of the Wilcoxon test in RStudio. As the p-value observed in the previous figure is less than α (0.05), the null hypothesis is rejected, which means that with 95% of confidence the mean evaluated overpasses the value chosen. Therefore, the addition of automated tests to the *Atendimento*'s CI process seems to be approved by the majority of people involved.

6.1.2 Question 2

The second question - Consider this scenario: Manel is pressured to resolve issues at the CI environment and estimates that they will be resolved in 3 hours. He takes more time than he thought to resolve the problem and decides to make the correction available in the CI environment without implementing the needed tests. Do you agree with Manel's behavior? - answers are not so unanimous as the first one. In this question, 50% of the respondents answered "Totally disagree" (option 1) and four of them (40%) chose option 2. Once again, one person chose option 3, which can have many reasons: did not understood the question; really doesn't have any thought on the issue; did not only thought about the decision only taken by Manel, but also in other situations where, for example, Manel's boss orders him to publish corrections without testing them. This last reason is as viable as the others; this question can be a little bit subjective and dependent of the specific situation, but it was intended to focus on the correct behavior in general situations.

As in the previous question analysis, a Shapiro-Wilks test must be realized to verify if the population follows a normal distribution. The two hypothesis are equal to the previous Shapiro-Wilks test. Running the test in RStudio, the p-value obtained is 0.008489. Once again, the p-value is less than α, the normal distribution is rejected and the t-test cannot be used. Therefore, it is used again the Wilcoxon test. The reference value defined here is a mean smaller than 2.5, and the hypotheses to this test are:

H_0: m $\geq$ 2.5

H_1: m $<$ 2.5

where m represents the mean of the sample

To run this test in RStudio, comparing to the one presented in Figure 6.2, it is passed the answers to the second question, the mu option changes to 2.5 and the alternative to "less". The p-value obtained is 0.005995 and less than α. Again the null hypothesis is rejected and, with 95% confidence, the average answer is in the range of the value defined as reference. In this case, the majority of people involved in the process answered as expected, affirming that Manel's behavior is incorrect.

6.1.3 Question 3

Question number 3 - How confident do you feel about the correct behavior of a new bug/development published in CI environment? - is the one that has more different answers. Eight persons answered positively, 50% of all the respondents being confident and 30% very confident about the correct behavior of a new bug fix/development deployed to CI. One person answered with the middle option, not showing nor confidence nor mistrust. Another person, chose option 1, not being confident at all about the new process deployment of a development/bug fix. This shows that a majority of people have a good feeling about the new process, but 10% may still show some reservations as the process was implemented recently.

To perform the planned statistical tests, the first one to realize is the Shapiro-Wilks test. The hypotheses of this test are already known from the previous sections. Using the RStudio, this test results in a p-value of 0.007736. This value is less than α for a level of confidence of 95%, meaning that the normal distribution of data is rejected and t-test cannot be used. The reference value defined to this question is 3.5, and the hypotheses to the Wilcoxon test are:

H_0: m $\leq$ 3.5

H_1: m $>$ 3.5

where m represents the mean of the sample

The p-value resulted from executing the test is 0.07893. As this value is greater than α, the null hypothesis is not rejected and with 95% of confidence the average answer is not superior to 3.5 as expected. At this phase of the project, it was expected that the people involved were more confident, but the required level of confidence is yet to be achieved.

6.1.4 Question 4

In the fourth question of the inquiry - How is the CI environment stability/behavior with the recent changes to the process? - the majority of the respondents answered with option 4 (70%). Two individuals answered with the most positive option, option 5, which means that 90% of the respondents have been satisfied about the recent stability/behavior of the CI environment. The remaining person doesn't classify it as bad or good, choosing the intermediate option.

Continuing with the statistical tests, the Shapiro-Wilks test is the starting point. The p-value obtained from this test is 0.003737 and as it is less than α, the normal distribution of data is rejected with 95% of confidence and the process will advance to the Wilcoxon test. The reference value discussed to use here is 4.3 and the average score of the answers to the inquiry should be greater than this value. This test consists on the following hypotheses:

H_0: m $\leq$ 4.3

H_1: m $>$ 4.3

where m represents the mean of the sample

In this test, it is obtained a p-value $=$ 0.878, a value inferior to the value of α. Therefore, with a confidence of 95%, the average answer to the stability/behavior of CI does not meat the reference value. As in the previous question, at this point the stability of the CI environment as not yet reach the desired level.

6.1.5 Question 5

The result of the final question of the inquiry, question 5 - These recent changes to CI process improve and accelerate the release of developments/bug fixes to version 17? -, shows that 90% of the respondents agree that the new CI process improves and accelerates the release to version 17, being that 5 respondents totally agree with that idea. One person is sill in doubt about the results of the changes made to the CI process and chose option 3.

Statistically analyzing the answers to this question, the Shapiro-Wilks test is made to evaluate the normality distribution of data. The p-value printed in RStudio when running this test is 0.008489. As this value is less than the α, the normal distribution of data is rejected and t-test cannot be executed. In this scenario, the average reference value decided to be exceeded is 3.75.

H_0: m $\leq$ 3.75

H_1: m $>$ 3.75

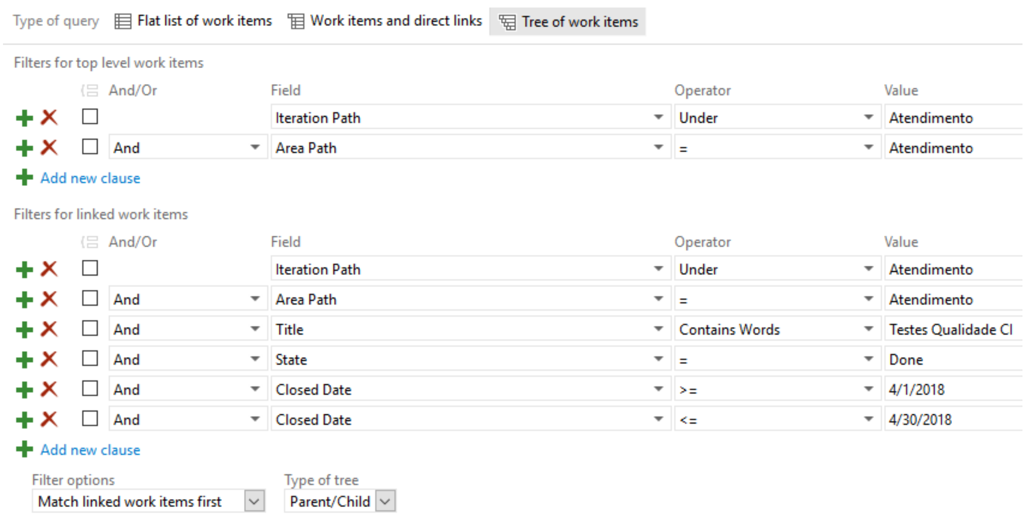
where m represents the mean of the sample

Using RStudio once more to perform this test, the resulted p-value is equal to 0.01108. This value is less than the used α=0.05, so the null hypothesis is rejected. With 95% of confidence, the average value of the answers obtained exceeds the reference value.Thus, the expected minimum of persons already agree that the new CI process improves and accelerates the releases of developments/bug fixes to version 17.

6.2 Average Time for CI Approval

As mentioned in the beginning of this chapter, this metric evaluates the time the testers team take to approve a development/bug fix published in the CI environment. It is compared the times registered in the old process and in the new one. To make that comparison, two samples of data were obtained from the TFS backlog of project *Atendimento* using queries. First, two months are chosen to be used in the evaluation of this metric; as in April of this year the solution documented in this dissertation was not being implemented yet, data from this month is chosen to represent the sample from the old CI process; in the last months the solution was implemented, so to guarantee more reliable feedback, data from September is selected to the second sample.

To obtain the data referred above, various queries were created in the backlog. As observed in Figure 6.3, in the query it is simply defined that the top level work items (general backlog items) must be from iteration and area *Atendimento* and the linked work items (general backlog items child tasks) must be from the same iteration and area, contain *Testes Qualidade CI* as title and state "Done", and their closed date (date they passed to "Done" state) must meet the interval specified. In the example query presented, it is searching for developments approved during April. To search for developments approved in September, it is only needed to change the interval of Closed Date filter to 9/1/2018 and 9/30/2018. The area path filter only obtains developments approved because the backlog is divided in two areas: the *Atendimento* area contains only new developments and one of its child areas, called *Suporte* (Support), contains bugs/issues reported internally in Glintt-HS or by its clients. Therefore, to retrieve bugs whose corrections were approved in CI environment in the chosen months, it is necessary in the TFS query to change the area path filter to *Atendimento\Suporte*. The rest of the query stays the same as shown in Figure 6.3, only changing the Closed Date to the pretended month.



Type of query: Flat list of work items | Work items and direct links | Tree of work items

Filters for top level work items

And/Or	Field	Operator	Value
	Iteration Path	Under	Atendimento
And	Area Path	=	Atendimento

Add new clause

Filters for linked work items

And/Or	Field	Operator	Value
	Iteration Path	Under	Atendimento
And	Area Path	=	Atendimento
And	Title	Contains Words	Testes Qualidade CI
And	State	=	Done
And	Closed Date	>=	4/1/2018
And	Closed Date	<=	4/30/2018

Add new clause

Filter options: Match linked work items first

Type of tree: Parent/Child

FIGURE 6.3: TFS Backlog - query for developments approved in CI in April

Running each query referred, it is presented for the month specified a list of the backlog items or bugs with their respective *Testes Qualidade CI* task indicated as "Done". Opening the information about this task, it is possible to observe its history, where it can be seen the date and time that the task state changed to "In Progress" and to "Done". This way, it is easier to obtain the time needed in hours to approve the respective development/bug fix. Observing the tasks for areas *Atendimento* and *Atendimento*\Suporte during each month defined, the obtained results are presented in Tables 6.2 and 6.3.

TABLE 6.2: Time (hours) for CI Approval in April and September.

April Old CI Process	**September New CI Process**
27.30	99.28
15.83	32.72
13.07	21.42
38.72	11.20
11.37	94.53
47.53	104.55
184	72.35
174.78	87.87
138.78	4.12
2.73	3.2

TABLE 6.3: Time (hours) for CI Approval in April and September (part 2).

April Old CI Process	**September New CI Process**
12.35	15.62
3.22	22.43
3.05	0.33
50.22	5.05
32.52	6.63
17.72	5.23
93.35	30.68
10.22	1.18
55.87	89.63
80.23	1.65
32.25	7.63
10.23	4.07

To evaluate this two samples, a paired t-test can be used. This type of test is used to evaluate differences between paired observations, being ideal to compare "before" and "after" samples. As this metric compares the results of the old and new process of CI and CD at Glintt-HS, the paired t-test seems to be the best choice of test to apply. As in the user satisfaction metric, to perform a t-test some assumptions must be verified. The first two assumptions referred earlier are also checked in this case. To perform a paired t-test, the differences for the matched-pairs must follow a normal probability distribution. Therefore, a Shapiro-Wilks test is executed, using the following hypotheses:

H_0: The differences for the matched-pairs follow a normal distribution

H_1: The differences for the matched-pairs do not follow a normal distribution

Once again, RStudio is used to calculate the differences between the matched-pairs and to run the Shapiro-Wilks test. The p-value obtained 0.7648 is greater than α=0.05, so the normal distribution of the differences for the matched-pairs is not rejected. Also, another assumption, not needed to verify in the user satisfaction tests, is the fact that there should not be no significant outliers in the differences of the matched-pairs. To check if outliers exist, a boxplot is drawn using RStudio. Figure 6.4 presents the boxplot of the differences between the two related groups.

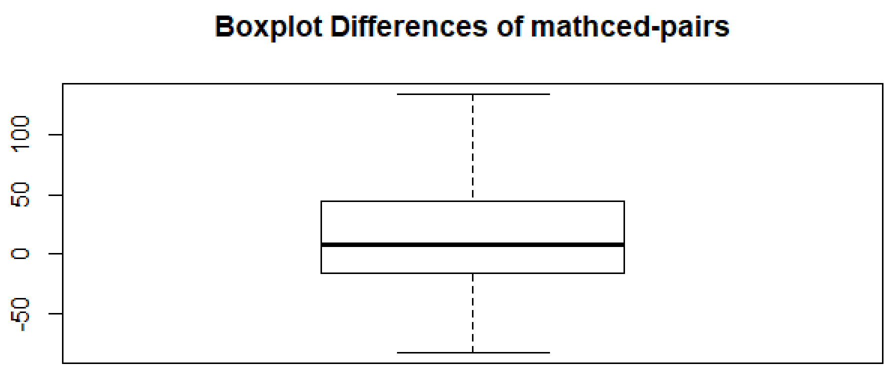


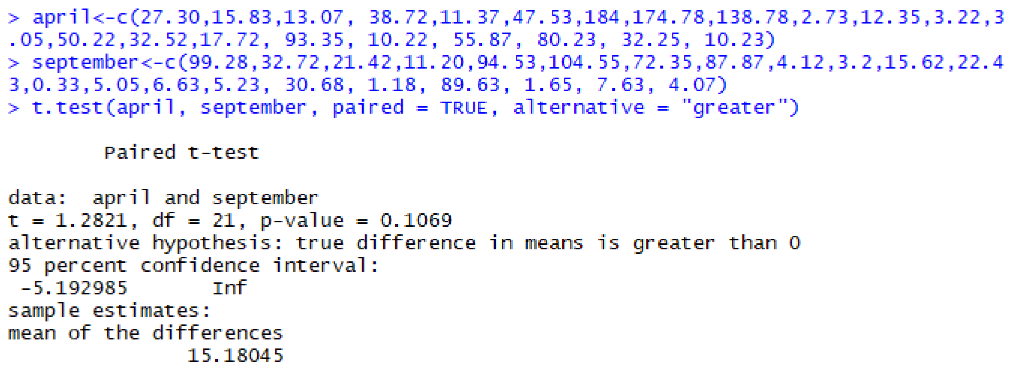
FIGURE 6.4: RStudio - boxplot of the differences between the two matched-pairs

In the previous figure, it is possible to observe that no outlier was drawn. Thus, the t-test can be used. As this test should evaluate if the average time for the new process is less than in the old, the test hypotheses are:

H_0: $m_o \leq m_n$

H_1: $m_o > m_n$

where m_o represents the mean of the old process and m_n the mean of the new process



```
> april<-c(27.30,15.83,13.07, 38.72,11.37,47.53,184,174.78,138.78,2.73,12.35,3.22,3
.05,50.22,32.52,17.72, 93.35, 10.22, 55.87, 80.23, 32.25, 10.23)
> september<-c(99.28,32.72,21.42,11.20,94.53,104.55,72.35,87.87,4.12,3.2,15.62,22.4
3,0.33,5.05,6.63,5.23, 30.68, 1.18, 89.63, 1.65, 7.63, 4.07)
> t.test(april, september, paired = TRUE, alternative = "greater")

        Paired t-test

data:  april and september
t = 1.2821, df = 21, p-value = 0.1069
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 -5.192985       Inf
sample estimates:
mean of the differences
               15.18045
```

FIGURE 6.5: RStudio - paired t-test applied for Average Time for CI Approval

Figure 6.5 presents the results of the paired t-test executed in RStudio. As the p-value is greater than α (0.05), the null hypothesis is not rejected. Therefore, with 95% of confidence, it is not possible to affirm that the average time for CI approval in the new process is less than the average time in the old one.

Chapter 7

Conclusions

This chapter describes the conclusion of the solution presented in this dissertation, including a summary of the results obtained and an explanation of future work to improve the implemented process.

7.1 Results

The solution documented in this dissertation was implemented successfully and keeps on being improved every day. The goals enumerated in the beginning of this document were all achieved with the implementation of a new CD architecture. The integration of automated tests combined with a rollback process that didn't exist results in a more improved and reliable process. Taking a look at the evaluation of the solution presented in the previous chapter, the feedback obtained by the people involved in the process is positive; the majority of the questions already have an average of answers that confirms the success of the new CI/CD process. There is an ongoing change of mentality on the company and most people agree with the use of automated tests, understand the importance of implementing tests when working on new developments or bug fixes and agrees that the improvements made help to release new versions of the application faster and with more reliability. Despite this, the confidence that changes deployed in the CI environment work correctly and the stability of this environment as not yet achieved the average levels proposed. Still, these two factors already have a very positive feedback that also reinforces the consistency of the outcome. About the factor of time for CI approval also evaluated, the tests performed indicate that it is not yet possible to confirm its improvement. Nevertheless, it can be seen in Figure 6.5 that the mean of the differences between the two analyzed groups of data is approximately 15.18. This means that subtracting the mean of the data registered in April with the mean of the records from September, the result is positive; therefore, the mean in September is less than in April. In practice, the time to approve changes published in the CI environment has decreased, but as the two related samples have relatively few records the statistical test made cannot yet confirm with 95% confidence that the process has improved.

7.2 Future Work

Every solution needs continuous evolution and maintenance, and the *Atendimento*'s CI/CD process at Glintt-HS is not different. In the previous chapters there were mentioned some designed steps that were not implemented. The first one that stands out is the lack of nonfunctional acceptance tests; these tests were approached in the technological background and in the design of the Jenkins pipeline, but during the implementation of the various groups of tests, the focus of the implementation turned to the more important and needed tests and nonfunctional tests were left behind. In the near future, it is planned the beginning of the development of these tests.

Other change to the designed solution that was explained in the implementation chapter was the trigger of the automated build. Contrary to what was thought, the build could not be triggered on each check-in, but with rolling builds. With the planned improvement of the merge process to the CI branch, this will be changed and it will allow to have a faster feedback obtained from the continuous integration of code.

Besides the future work presented before, it is planned the total migration of the Jenkins pipeline to the declarative syntax. As seen in the implementation chapter, the current pipeline code has some scripted blocks to define and reassign values to dynamic variables. With Jenkins is now possible to use Shared Libraries, a feature that allows to define variables and functions that can be imported and used along the declarative pipeline. Also, the deployment phase of the pipeline takes a long time to be completed and the deployment of assemblies and configuration files should be made in parallel. As this phase is dependent on an input step, there is some hope that in the future it will be possible to integrate an input step with parallel stages.

Bibliography

Alexandrova, Julia (Mar. 2016). *Continuous Integration with TeamCity*. URL: https://confluence.jetbrains.com/display/TCD9/Continuous+Integration+with+TeamCity.

Balliauw, Maarten (Mar. 2013). *TeamCity plugin for Visual Studio*. URL: https://blog.jetbrains.com/dotnet/2013/03/12/teamcity-plugin-for-visual-studio/.

Charlie Poole, Rob Prouse (2017). *What Is NUnit?* URL: http://nunit.org/.

CircleCI (2018). *Migrating from Jenkins to CircleCI*. URL: https://circleci.com/docs/1.0/migrating-from-jenkins/.

Cohen, Frank (2012). *Write Your First Functional Selenium Test*. URL: http://www.pushtotest.com/selenium-tutorial-for-beginners-tutorial-1.

Corgan, Scott (2016). *tap-dot*. URL: https://www.npmjs.com/package/tap-dot.

Dietrich, Erik (Feb. 2017). *Unit Test Frameworks for C#: The Pros and Cons of the Top 3*. URL: https://stackify.com/unit-test-frameworks-csharp/.

Easy, Selenium (2018). *What is Jenkins and Use of it?* URL: http://www.seleniumeasy.com/jenkins-tutorials/what-is-jenkins-and-advantages-of-jenkins-continuous-integration-tool.

Eclipse (Feb. 2017). *About Jenkins*. URL: https://wiki.eclipse.org/index.php?title=Jenkins%5C&oldid=414064.

Facebook (2018). *Jest*. URL: https://facebook.github.io/jest/.

Foundation, .NET (2017a). *About xUnit.net*. URL: https://xunit.github.io/.

– (2017b). *Getting Started with xUnit.net (desktop)*. URL: http://xunit.github.io/docs/getting-started-desktop.html.

Foundation, Apache Software (2018). *Twitter github Apache JMeter™*. URL: http://jmeter.apache.org/.

Fowler, Martin (May 2006a). *Continuous Integration*. URL: http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf.

– (Jan. 2006b). *Xunit*. URL: https://www.martinfowler.com/bliki/Xunit.html.

– (May 2013). *Continuous Delivery*. URL: https://martinfowler.com/bliki/ContinuousDelivery.html.

– (Jan. 2018). *IntegrationTest*. URL: https://martinfowler.com/bliki/IntegrationTest.html.

Gesso, Justin (Apr. 2015). *Continuous Integration Best Practices for Software Engineering*. URL: https://betsol.com/2015/04/continuous-integration-best-practices-for-software-engineering/.

Glintt – Global Intelligent Technologies, S.A. (Apr. 2018). *RELATÓRIO E CONTAS 2017*. URL: https://www.glintt.com/es/Inversores/Cuentas/Resultados%20Financeiros/Glintt_RelatorioContas2017_H.PDF.

Halliday, James (Jan. 2018). *tape*. URL: https://github.com/substack/tape.

Harding, Ben (May 2017). *Comparing Jasmine, Mocha, AVA, Tape, and Jest*. URL: https://dzone.com/articles/comparing-jasmine-mocha-ava-tape-and-jest.

Haunts, Stephen (June 2014). *Post Deployment Smoke Testing Open Source Project*. URL: https://stephenhaunts.com/projects/post-deployment-smoke-tester/.

Al-Jallad, Mohannad M. (2012). “REA Business Modeling Language: Toward a REA based Domain Specific Visual Language”. PhD thesis. KTH Royal Institute of Technology.

Jenkins (2018). *Jenkins*. URL: https://jenkins.io/.

JetBrains (2018). *Features*. URL: https://www.jetbrains.com/teamcity/features/.

Jez Humble, David Farley (2011). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional.

Koen, Peter et al. (2001). “Providing Clarity and A Common Language to the “Fuzzy Front End””. In: *Research-Technology Management* 44.2, pp. 46–55. DOI: 10.1080/08956308.2001.11671418.

Li, Gang (June 2011). *Automate web application testing with Sahi*. URL: https://www.ibm.com/developerworks/library/wa-sahi/.

Microsoft (2015). *Microsoft Build Tools 2015*. URL: https://www.microsoft.com/en-us/download/details.aspx?id=48159.

– (2018). *Team Foundation Server*. URL: https://www.visualstudio.com/tfs/.

MochaJS (Feb. 2018). *Mocha*. URL: https://mochajs.org/.

Oliveira, Jason de (Oct. 2012). *[Tutorial] Test Driven Development with Visual Studio 2012 Part4: Full cycle of TDD using Visual Studio 2012*. URL: http://www.jasondeoliveira.com/2012/10/tutorial-test-driven-development-with_25.html.

Osterwalder, Alexander (2004). *The Business Model Ontology: A proposition in a Design Science Approach*. URL: http://www.hec.unil.ch/aosterwa/PhD/Osterwalder_PhD_BM_Ontology.pdf.

Pecanac, Vladimir (Feb. 2016). *TOP 8 CONTINUOUS INTEGRATION TOOLS*. URL: https://code-maze.com/top-8-continuous-integration-tools/.

Peffers, Ken et al. (2007). “A Design Science Research Methodology for Information Systems Research”. In: *Journal of Management Information Systems* 24.3, pp. 45–78.

Phillips, Andrew (July 2014). *The Continuous Delivery Pipeline - What it is and Why it’s so Important in Developing Software*. URL: https://devops.com/continuous-delivery-pipeline/.

Pittet, Sten (2018). *Continuous integration vs. continuous delivery vs. continuous deployment*. URL: https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd.

RadView (2018). *WebLOAD Free Edition*. URL: https://www.radview.com/webload-download/.

Sandstrom, Terje (Nov. 2012). *How to manage unit tests in Visual Studio 2012 Update 1 : Part 1–Using Traits in the Unit Test Explorer*. URL: https://blogs.msdn.microsoft.com/devops/2012/11/09/how-to-manage-unit-tests-in-visual-studio-2012-update-1-part-1using-traits-in-the-unit-test-explorer/.

SeleniumHQ (2018a). *Selenium*. URL: https://github.com/SeleniumHQ/selenium.

– (2018b). *What is Selenium?* URL: http://www.seleniumhq.org/.

SmartBear (2018). *What Is Unit Testing?* URL: https://smartbear.com/learn/automated-testing/what-is-unit-testing/.

TestAnything (2018). *Test Anything Protocol*. URL: https://testanything.org/.

TytoSoftware (2018a). *Introduction - Sahi Pro*. URL: http://sahipro.com/docs/introduction/index.html.

– (2018b). *Sahi Open Source Automation Testing Tool For Web Applications*. URL: http://sahipro.com/sahi-open-source/.

Watir (2018). *Watir is...* URL: http://watir.com/.

Webpack (2018). *Concepts*. URL: https://webpack.js.org/concepts/.

White, Oliver (May 2014). *Java Tools and Technologies Landscape for 2014*. URL: `https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/12/`.

Woodall, Tony (2003). “Conceptualising 'Value for the Customer': An Attributional, Structural and Dispositional Analysis”. In: *Academy of Marketing Science Review* 2003.12.

Zinser, Jan-André (2017a). *Using mocha-webpack with jsdom*. URL: `https://zinserjan.github.io/mocha-webpack/docs/guides/jsdom.html`.

– (2017b). *Webpack configuration*. URL: `https://zinserjan.github.io/mocha-webpack/docs/installation/webpack-configuration.html`.

Appendix A

FAST Diagram

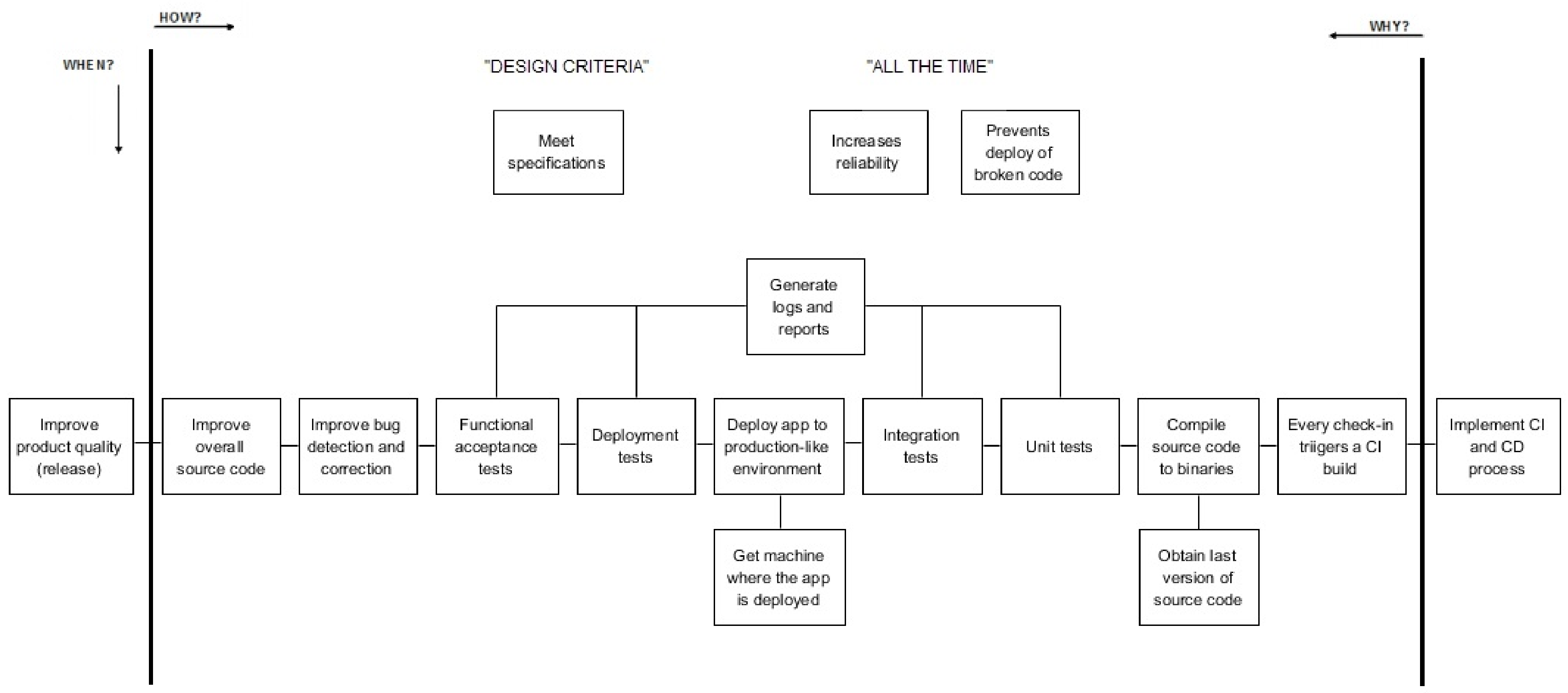

HOW?
WHEN?
"DESIGN CRITERIA"
"ALL THE TIME"
WHY?
Meet specifications
Increases reliability
Prevents deploy of broken code
Generate logs and reports
Improve product quality (release)
Improve overall source code
Improve bug detection and correction
Functional acceptance tests
Deployment tests
Deploy app to production-like environment
Integration tests
Unit tests
Compile source code to binaries
Every check-in triigers a CI build
Implement CI and CD process
Get machine where the app is deployed
Obtain last version of source code

FIGURE A.1: FAST Diagram

Appendix B

Pipeline Diagram

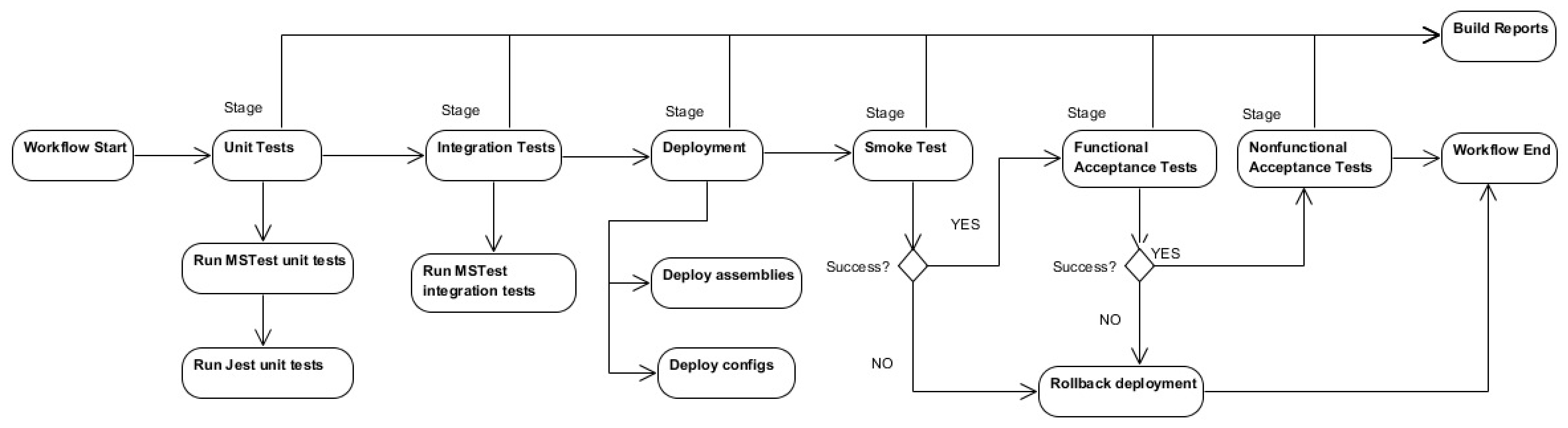
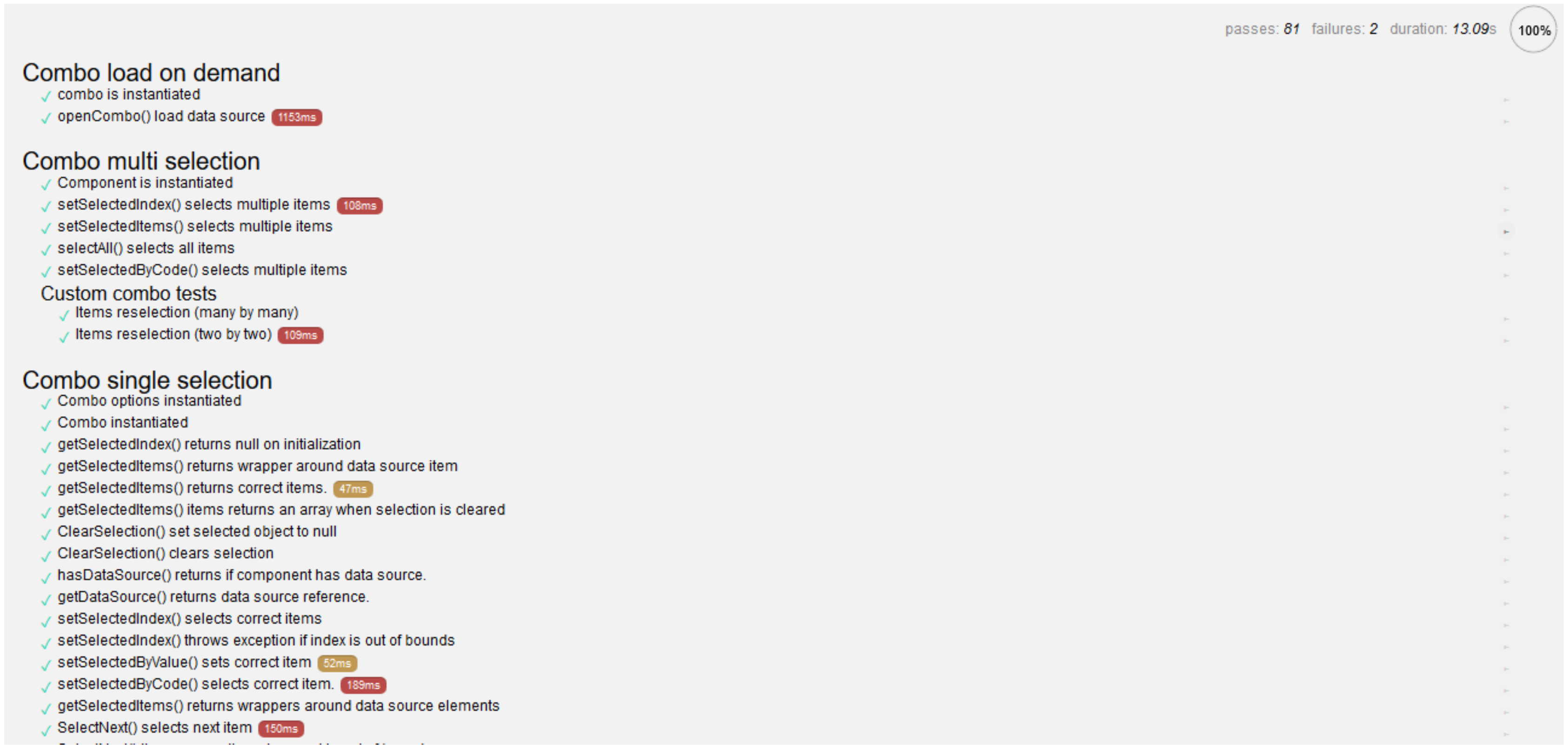


FIGURE B.1: Pipeline Diagram

Appendix C

Atendimento Mocha Tests - Results in the Browser

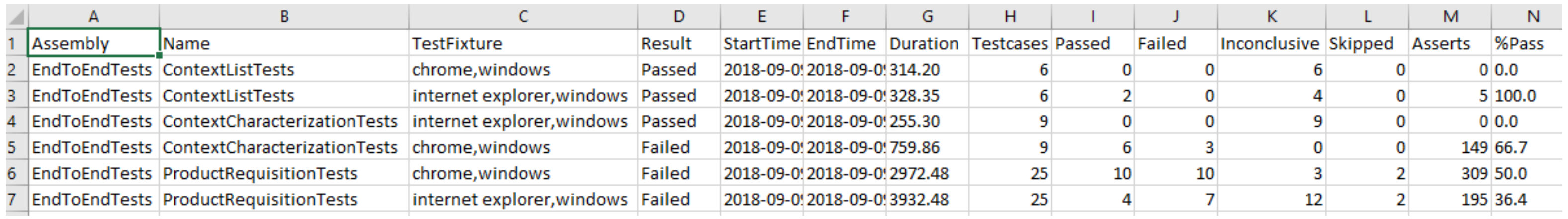


```
passes: 81  failures: 2  duration: 13.09s  100%

Combo load on demand
  ✓ combo is instantiated
  ✓ openCombo() load data source 1153ms

Combo multi selection
  ✓ Component is instantiated
  ✓ setSelectedIndex() selects multiple items 108ms
  ✓ setSelectedItems() selects multiple items
  ✓ selectAll() selects all items
  ✓ setSelectedByCode() selects multiple items
  Custom combo tests
    ✓ Items reselection (many by many)
    ✓ Items reselection (two by two) 109ms

Combo single selection
  ✓ Combo options instantiated
  ✓ Combo instantiated
  ✓ getSelectedIndex() returns null on initialization
  ✓ getSelectedItems() returns wrapper around data source item
  ✓ getSelectedItems() returns correct items. 47ms
  ✓ getSelectedItems() items returns an array when selection is cleared
  ✓ ClearSelection() set selected object to null
  ✓ ClearSelection() clears selection
  ✓ hasDataSource() returns if component has data source.
  ✓ getDataSource() returns data source reference.
  ✓ setSelectedIndex() selects correct items
  ✓ setSelectedIndex() throws exception if index is out of bounds
  ✓ setSelectedByValue() sets correct item 52ms
  ✓ setSelectedByCode() selects correct item. 189ms
  ✓ getSelectedItems() returns wrappers around data source elements
  ✓ SelectNext() selects next item 150ms
```

Figure C.1: *Atendimento* Mocha Tests - Results in the Browser

Appendix D

Atendimento Functional Acceptance Tests - CSV Report

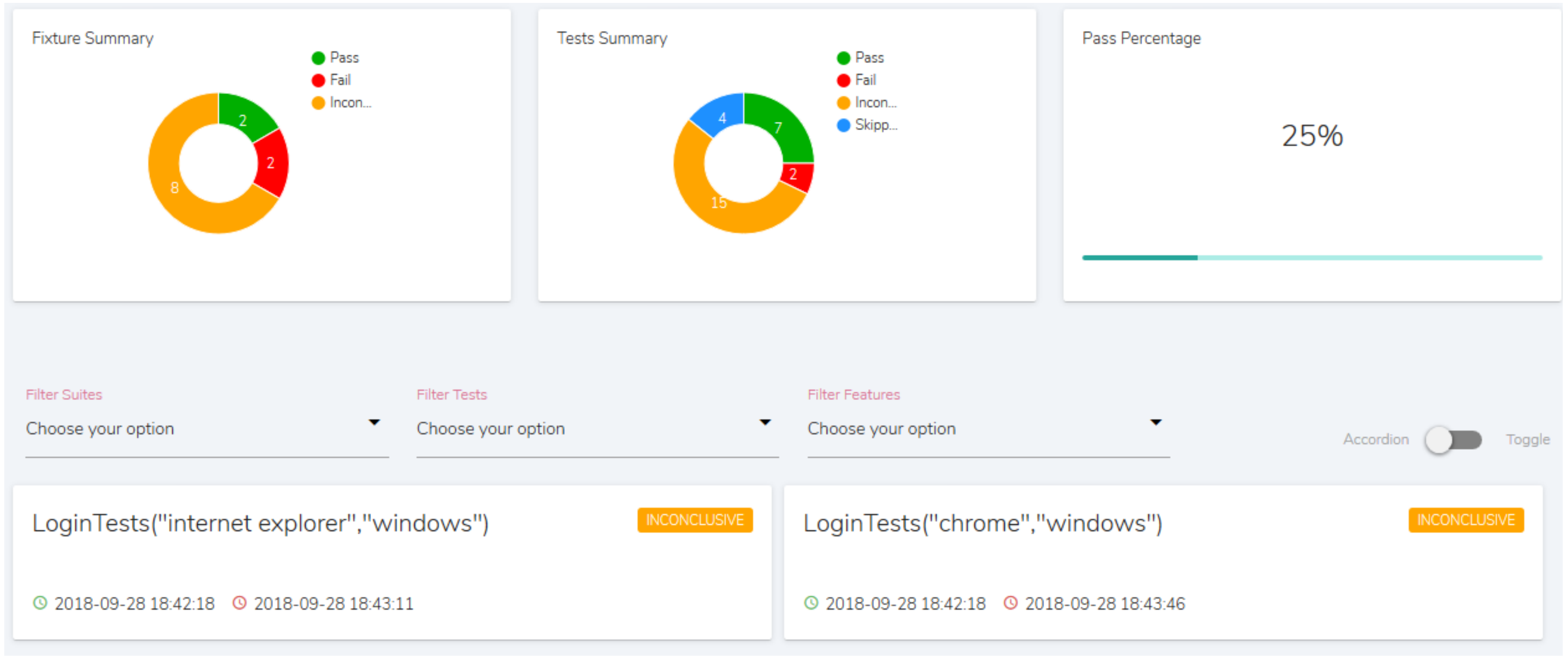


	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Assembly	Name	TestFixture	Result	StartTime	EndTime	Duration	Testcases	Passed	Failed	Inconclusive	Skipped	Asserts	%Pass
2	EndToEndTests	ContextListTests	chrome,windows	Passed	2018-09-0	2018-09-0	314.20	6	0	0	6	0	0	0.0
3	EndToEndTests	ContextListTests	internet explorer,windows	Passed	2018-09-0	2018-09-0	328.35	6	2	0	4	0	5	100.0
4	EndToEndTests	ContextCharacterizationTests	internet explorer,windows	Passed	2018-09-0	2018-09-0	255.30	9	0	0	9	0	0	0.0
5	EndToEndTests	ContextCharacterizationTests	chrome,windows	Failed	2018-09-0	2018-09-0	759.86	9	6	3	0	0	149	66.7
6	EndToEndTests	ProductRequisitionTests	chrome,windows	Failed	2018-09-0	2018-09-0	2972.48	25	10	10	3	2	309	50.0
7	EndToEndTests	ProductRequisitionTests	internet explorer,windows	Failed	2018-09-0	2018-09-0	3932.48	25	4	7	12	2	195	36.4

FIGURE D.1: *Atendimento* Functional Acceptance Tests - CSV Report

Appendix E

Atendimento Functional Acceptance Tests - HTML Report



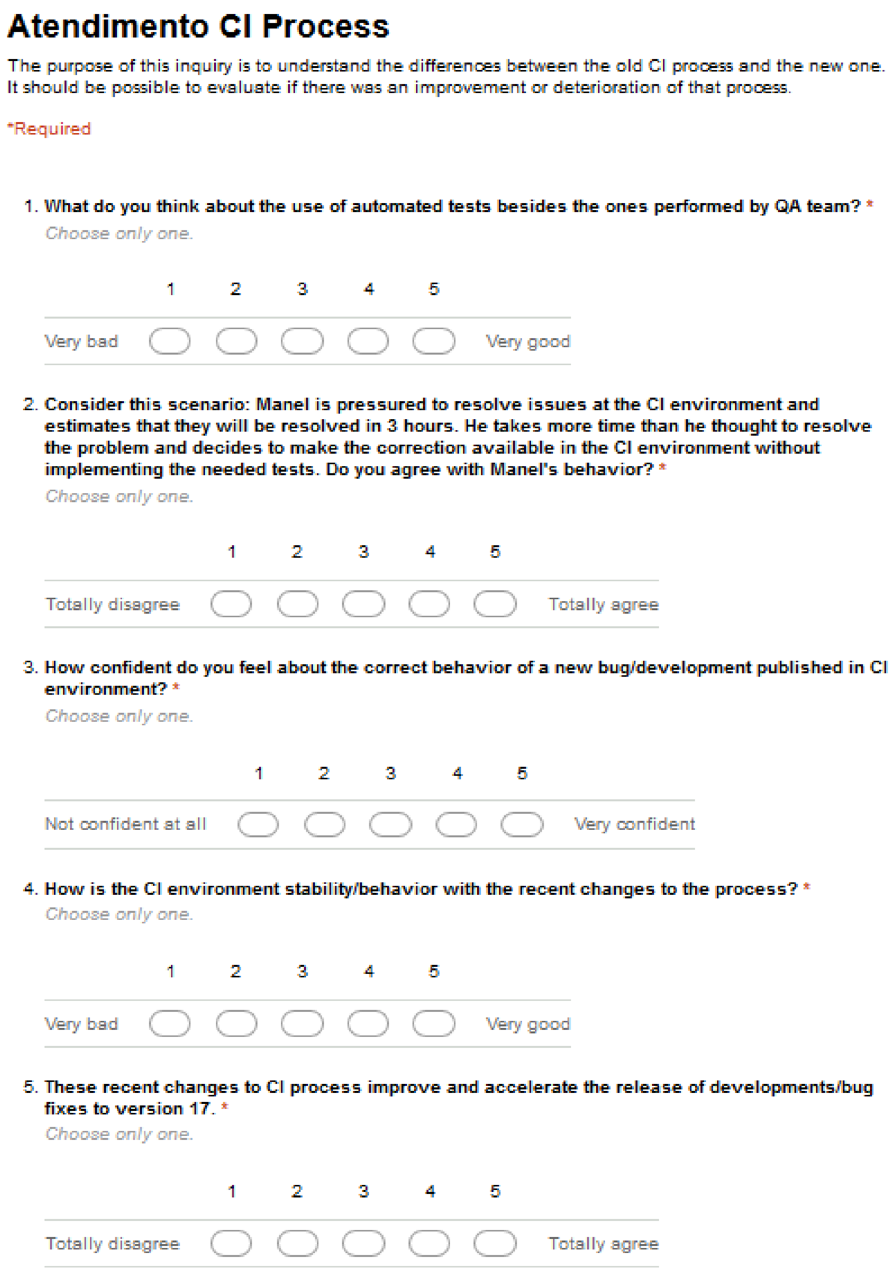
LoginTests("internet explorer","windows") INCONCLUSIVE
2018-09-28 18:42:18 2018-09-28 18:43:11

LoginTests("chrome","windows") INCONCLUSIVE
2018-09-28 18:42:18 2018-09-28 18:43:46

FIGURE E.1: *Atendimento* Functional Acceptance Tests - HTML Report

Appendix F

Evaluation - User Satisfaction Inquiry



Atendimento CI Process

The purpose of this inquiry is to understand the differences between the old CI process and the new one.
It should be possible to evaluate if there was an improvement or deterioration of that process.

*Required

1. What do you think about the use of automated tests besides the ones performed by QA team? *
Choose only one.

	1	2	3	4	5	
Very bad	○	○	○	○	○	Very good

2. Consider this scenario: Manel is pressured to resolve issues at the CI environment and estimates that they will be resolved in 3 hours. He takes more time than he thought to resolve the problem and decides to make the correction available in the CI environment without implementing the needed tests. Do you agree with Manel's behavior? *
Choose only one.

	1	2	3	4	5	
Totally disagree	○	○	○	○	○	Totally agree

3. How confident do you feel about the correct behavior of a new bug/development published in CI environment? *
Choose only one.

	1	2	3	4	5	
Not confident at all	○	○	○	○	○	Very confident

4. How is the CI environment stability/behavior with the recent changes to the process? *
Choose only one.

	1	2	3	4	5	
Very bad	○	○	○	○	○	Very good

5. These recent changes to CI process improve and accelerate the release of developments/bug fixes to version 17. *
Choose only one.

	1	2	3	4	5	
Totally disagree	○	○	○	○	○	Totally agree

FIGURE F.1: User Satisfaction Inquiry