POLITECNICO DI TORINO

Master of Science in Ingegneria Informatica (Computer Engineering)

Master Thesis

Test scripting language selection for web-based automated testing

Advisors:
PhD. Maurizio Morisio
PhD. Riccardo Coppola

Candidate:
Santiago Agudelo Ibarra

December 2018

To my beloved wife and my parents: Finally, we did it

Acknowledgements

I thank my family for the patience they have given me over the years. To my beloved wife, who has been an unconditional support in the realization of this work. And finally, but no less important, I thank my friends and colleagues for the countless collaborations to get this master's degree forward.

Summary

In the first part of this thesis, a comparison of the Selenium IDE and Selenium WebDriver automation tools (Java and C#) for different types of web applications is presented, in order to show the benefits of automated testing when each of the tools is used in different types of web applications. Next, a classification of the web applications is made according to the testing levels, giving, as a result, the separation of the applications in three groups: small, medium and large application. Finally, through the proposed methodology for the automation of the tests, the suitable automation tool can be selected according to the knowledge achieved by the tester and the type of application to be tested.

Keywords: Selenium IDE, Selenium WebDriver, Java, C#, Web application, Test automation, Test methodology.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software, as many knows, it's a key ingredient in many of the devices and systems that are in our society. It's found in smart watches, ovens, cars, DVD players, cell phones and even more complex things as airplanes, spaceships, air control systems, among others. The origin of the errors in the software began with the development of the same. This is not the case where we start with a perfect product and invent ways to spoil it. But the only way that errors enter into a program is when they are introduced by the author. Nowadays, there is no way to develop an error-free program, even if we did not discover them during development or in tests does not mean that they are not there. A quote from Alan Perlis [1], the first winner of the Turing award, describes in a better way the message in the previous paragraph:

"There are two ways to write error-free programs; only the third one works."

The previous sentence tells us that there is no way to develop error-free software. So, how do we guarantee that the produced software meets its requirements? How to guarantee that an error will not cause a failure in the program while it's in the middle of a very important operation such as a transaction in a bank's management system? We cannot guarantee that, but, with the use of tests, we can ensure a certain level of reliability. Software testing is one of the areas of software engineering that has gained most of the interest, increasing in this waythe amount of the researches being carried out. Some define the evidence as a source that provides information about the quality of the product or service. But strictly speaking, testing a program is trying to make it fail [2]. That is the main objective of the tests: find the maximum number of possible execution failures.
In literature, authors differ in the categorization of the different types of tests. Some categorizations are made depending on the size of the software to be tested, the maturity of the testing process itself, and the testing strategy chosen, among others. Moreover the number of different types of tests vary as much as the different development approaches. But, regardless of the categorization of the considered tests, any software strategy must incorporate a planning stage, that will include the design of the test cases, the execution of the tests and a evaluation of the data being collected during the execution.

1.1 Limitations of Software Testing

Any non-trivial software will have errors. It's clear that performing a few tests is not enough to prove that a system is completely error-free. There are several reasons why it's impossible to prove that. For example, consider a program that draws a quadrangle and accepts four points as input where the coordinates of the point are limited between 1 and 10. In this case, $10^4 (10 \times 10$ ways to draw a point and therefore $(10 \times 10)^2$ ways to draw a line). There are four lines, therefore $(10^4)^4 = 10^{16}$ entries of four lines are possible. Assuming that a tester tests 1000 input combinations per second and works 24 hours a day and 365 days a year. The tester could test all possible input combinations at

$$\frac{10^{16}}{10^3} = 10^{13} \; seconds$$

At 3.14×10^7 seconds per year,

$$\frac{10^{13}}{3.14 \times 10^7} = 3.171 \times 10^5 \; years$$

It would be needed to perform all possible tests. This makes it clear that the program output for all possible input combinations cannot be verified in a reasonable time. Another limitation is that the code can hide faults. Suppose that an expression is written as $x + x$, where it should be $x \times x$, using $x = 2$ as input, the correct result is produced and the tester could assume that the system contains no faults. Consider the following code fragment [3]:

```
int scale(int j) {
    j = j - 1; //should be j = j + 1
    j = j / 30000;
    return j;
}
```

From the 65,536 possible values for j only 40 values will produce an incorrect result: -30000, -29999, -27000, -26999, -24000, -23999, -21000, -20999, -18000, -17999, -15000, -14999, -12000, -11999, -9000, -8999, -6000, -5999, -3000,-2999, 2999, 3000, 5999, 6000, 8999, 9000, 11999, 12000, 14999, 15000, 17999, 18000, 20999, 21000, 23999, 24000, 26999, 27000, 29999, 30000. None of these values is a boundary of j. If all possible values except these 40 were used, then even a nearly exhaustive test would not reveal the error. Normally the tests uses the requirements as a reference point, so when the requirements are incorrect or incomplete, incorrect tests are produced. Omissions are also not revealed if implementation-based testing is used since the missing code cannot be tested. Errors in the tests can also lead to incorrect test results.

1.2 Motivation

Producing software with an insufficient level of quality can be a terrible business. According to the public Standish Group in its 2015 report (Chaos Report), 52% of software

development projects do not meet the objectives set and 19% of them are considered a failure. Only 29% achieve success. It is no longer enough for a specific application or program to be functionally complete, according to its purpose, but it must also satisfy other aspects in terms of security, performance, accessibility, maintainability, etc. Therefore, the goal of any organization that is dedicated to the software industry, should be to produce quality software, that is, create efficiently the software that fullfill the requirements.
As exposed above, software quality should be an obligatory requirement and more and more companies are considering, within the development process, quality assurance activities. These activities include a wide variety of tasks, the establishment of solid technical methods, metrics, analysis, and reports, well-planned software tests, reviews, etc. Those responsibles for all these activities aim to identify, document and communicate the possible deviations observed during the development process in order to make the appropriate decisions. They provide clarity about the state and the maturity of the product that is being developed.
One of the most important parts that make up the tasks of a quality assurance system is software testing. There are unit, integration, functional, performance, etc. tests. Some of them, by their nature, are susceptible to be automated, this means that they are executed by a machine or computer (usually called a robot) instead of the tester itself.
Currently, the automation of tests is a fundamental step to improve the quality of development. Every time the applications become more complex, the requirements are changing, more functionalities are been adding or modifying, etc., what resuts in different versions and in a reliability decrement of the quality of the final developed software. In these cases, the automation of tests becomes important and represents a clear advantage for dealing with the process of execution of the tests.
Currently, there are many tools that allow to generate and execute scripts, small programs for interacting with a given application, simulating the actions that could be performed in the application, among other functionalities. These tools help to increase productivity in terms of the number of tests performed, the scope of each of them (many combinations can be tested) and the reproducibility of the tests (we make sure that the tests are always done in the same way). These automatic tests do not replace the traditional tester (a physical person who performs the tests manually), but help to perform tedious and repetitive tasks, being necessary a more technical profile, since it requires to possess knowledge in different programming languages, databases, the architecture of web systems, etc.
Within the tests of web applications, one of the most used tools for functional testing automation is Selenium. Selenium is an open-source framework that has an important community behind it, and that allows navigation through a web application as a user would. The tool processes a script that describes the steps that must be performed during the test execution: open the browser, go to a specific URL, analyze text displayed on the screen, access a specific link, enter data, etc.
Selenium has evolved to offer a complete API for different programming languages. With the classes and methods that it provides, the scripts that represent each of the automated tests are created. Currently, the tool supports the main web browsers, such as Firefox, Chrome, Internet Explorer, Safari, etc. Within the programming languages in which one

can work with the Selenium API, better known as Selenium WebDriver, we focused our work in the Java and C# languages, since they are the two most used today, and although they are very similar, we are interested in finding the differences of each of them, to know which of them may be the best choice to perform automated tests, depending on the knowledge that the tester has in programming.
The automation tools (as Selenium IDE) use simple structures for the creation of the test cases. Nevertheless, the use of robust programming languages (such as Java and C#) for the automation of web tests, allow potentiating some functionalities to efficiently consolidate the processes of Quality Assurance. This requires, on the part of the functional analyst, the knowledge of the languages, patterns and good practices of programming. These additional requirements translate into longer testing times and execution costs. For this reason, it is necessary to know and identify the advantages and disadvantages of the different languages and choose the one that allows having a balance between the learning curve, the execution times, and the analyst programming knowledge.

1.3 Thesis objectives

1.3.1 Main objective

The main objective of this thesis is to develop a methodology for selecting the appropriate automation tool for the creation of automated test scripts using Selenium IDE or Selenium WebDriver (Java and C#).

1.3.2 Specific objectives

- Compare Selenium IDE scripts written in its own native language, with Java and C# test automation programming languages using metrics related to the learning curve, test scenarios creation time, code maintainability and execution times.

  The following tasks are defined to achieve this objective:

  - Definition of metrics to make the comparison of programming languages
  - Creation of a survey to know what is the learning curve of the programming languages.
  - Execution of the automated tests to know the execution times in each of the languages.

- Determine a metric-based criteria for the selection of the appropriate programming language for a group of applications.

  The following tasks are defined to achieve this objective:

  - Perform the categorization of the web applications within the groups.

- Implement and validate the methodology in different case studies

  The following tasks are defined to achieve this objective:

- Selection of three case studies for the implementation of the proposed automation methodology.
- Implementation of the automation of tests in Selenium IDE, Java and C#, in the case studies.

1.4 Structure of the thesis

The thesis is composed of 6 chapters. Chapter 2 gives a brief description of the definition of testing, the test strategies, types, and levels. Moreover, the definition of test automation can be found, benefits and challenges of test automation, and automation testing tools used in this thesis. And finally, the existing automated testing approaches are presented. Chapter 3 describes the analysis of the proposed methodology, beginning with a comparison of Java and C#. Then, it's shown the classification of web applications used within the methodology, and finally, in the last two sections, the proposed methodology is presented and developed. Chapter 4 presents the three selected case studies for the developing of the automated tests, through the implementation of the proposed methodology First a brief description of each of the applications is given and then it's detailed how the methodology was implemented in each of the stages. Chapter 5 describes the configuration used for each automation tool and the results with their respective analysis. Finally, Chapter 6 presents the conclusion of this document and future work.

Chapter 2

Background

In this chapter, all the theoretical framework and the related work of the thesis is shown. In section 2.1, the definitions of testing are described. In section 2.2, all the test strategies are described. In section 2.3, the testing types and its techniques are described. In section 2.4, the testing levels are described. In section 2.5, the definitions of test automation are described. In section 2.6, the considered automation testing tools are described.

2.1 Definition of Testing

The meaning adopted generally in literature for the term testing, refers to the dynamic approach. The Guide to the Software Engineering Body of Knowledge (SWEBOK) [4] defines the software test as:

"Software testing consists of the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain."

This definition mentions several aspects that are important to highlight:

- **Dynamic Verification:** Implies that the program must be executed for the input data.
- **Static Verification:** It's based on the systematic review of artifacts such as the code and the documents with the specifications and the design. Static techniques are very effective for the prevention of defects. It has been pointed out in repeated sources in literature that these techniques constitute the main source of error detection. Myers [5] indicates that from 30% to 70% of all defects found at the end of the testing process can be detected with these techniques. Other authors state that it reaches 90% [6].
- **The expected behavior:** It must be possible to decide when the observed output of the execution of the program is acceptable or not. The observed behavior can be checked against the user's expectations, against a specification or against anticipated behavior by implicit requirements or reasonable expectations.

- **A finite set of test cases:** The test should be done on a set, finite and limited, of possible test cases. A test case is a set of input values, expected results, pre-conditions and post-conditions of execution, developed with a particular objective [7].
- **Properly selected:** The selection of test cases should maximize the effectiveness of the test.
- **The infinite domain of execution:** It states that the test requirements have no limit.

The main difference between the first two techniques mentioned above is that in the static technique an analysis is made without ever executing the application, while in the dynamic technique the application has to be executed, in addition, the default cost is cheaper in static techniques. However, this does not imply replacing or eliminating dynamic techniques, which are essential for the validation of the application. In [4], it's stated that the static approach is not strictly part of the testing, although it complements, studying under the area of quality, even if the definition given for testing is generic: Testing is an activity carried out to evaluate the quality of the product and improve it, identifying defects and problems. Myers [5] defines:

- **Testing or Computer Testing:** "The process of executing a program with the intent of finding errors".
- **Software Testing:** "Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and, conversely, that it does not do anything unintended".

The guidance for the validation of software for the industry of the FDA6, V.2 [8] establishes that the Software Test implies *"running software products under known conditions with defined inputs and documented outcomes that can be compared to their predefined expectations"*. It is defined as one of the many verification activities that are carried out with the purpose of confirming that the output of the development accomplishes with its entry requirements. Among the other verification activities, it includes static approaches. It also warns that the terms Verification, Validation, and Testing have been used as equivalents in literature. In [7], testing is defined as:

"The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component."

Kaner makes an important distinction when defining testing as:

"A technical investigation done to expose quality-related information about the product under test."

Its goal should not be merely to find faults by provoking failures; the horizon must be extended, allowing the tests to serve as a learning mechanism for the system under test [9], through the deep knowledge of the application, such as the establishment of critical paths, allowed roles, functionalities more prone to fail, etc.

2.2 Test Strategies

Generally, the first step in a testing process is to create a test strategy that specifies the details of the testing procedure and help to organize all the testing process. As Shiva Kumar's says in the handbook of testing [10], the intention of a test strategy is to:

- Provide a framework and a focus for improvement efforts, and to
- Provide a means for assessing progress.

All the requirements, system design and acceptance criteria are merged into the testing strategy document. Additionally, a description of the testing is needed along with notations of objectives, scope, and other conditions. A testing strategy document should also cover the project scope that is the description of what is to be and how is to be tested, how exhaustive the test must be and how much testing is needed [10]. The test objectives, it's the list of all test in order of importance. The features and functions to be tested, if there is an exception, it should be mentioned including the reasons for exclusion. The testing approach that describes the types and levels of testing to be conducted. The testing process with a detailed description of the steps in testing is shown in figure 2.1).

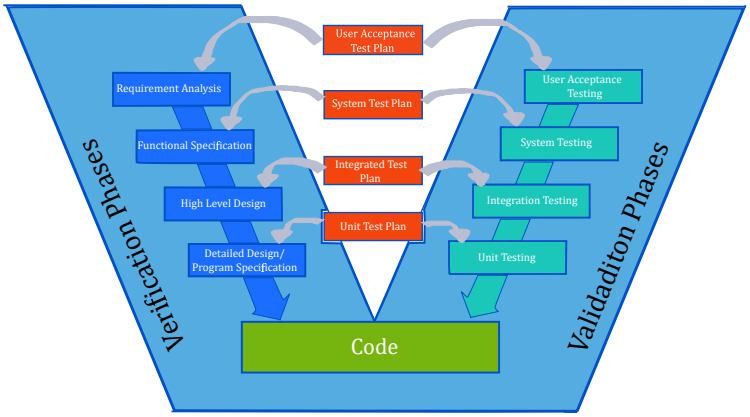

Figure 2.1: V Model.

2.3 Testing Types

Software tests can be divided into black box and white box based on the approach taken. The following sections describe the black box testing approach (section 2.3.1) and white box testing approach (section 2.3.2). In addition, some test techniques of each approach are mentioned, which are listed in table 2.1.

Test perspective	Techniques	Principle
Black box testing	Equivalent partition	The entry domain of the program under test is divided into data partitions with common characteristics (positive numbers, negative numbers, etc.).
	Analysis of limit values	The limit values (minimum and maximum), of each entry data in the program, are chosen as test data.
	Comparison test	Independent versions of the program to be tested are developed, using the same specifications. Each version is exercised with the same test cases (selected by another testing technique). If the results of the tests are different, the differences between the versions are analyzed in order to determine the failure cause.
	Test of the orthogonal table	An orthogonal table of test cases is created. In this table, the test cases are uniformly dispersed in the input domaind.
White box testing	Path testing	Each independent execution path in a program is tested at least once. A path is a way through which the execution proceeds through a function from its beginning to the end.
	Condition testing	All the logical conditions of the program under test are tested. Each clause of each condition of the program under test must be exercised with both values (true and false).
	Loop testing	The correct construction of the loops is tested (simple, nested, concatenated and unstructured loops).

Table 2.1: Black box and white box testing techniques.

2.3.1 Black Box Testing

In black box testing, the knowledge of the internal structure of the System under Test (SUT) code is not needed (see figure 2.2). The test cases are obtained from specifications and the program passes the test if the obtained output corresponds to the expected output. This type of tests is also known as functional or behavioral tests, focusing on obtaining sets of input conditions that fully exercise the functional requirements of a program [11].

Figure 2.2: Black Box Testing.

Black box testing tries to find the following categories of errors [11]:

- Incorrect or absent functions.
- Interface errors.
- Errors in data structures or access to external databases.
- Performance errors.
- Initialization and termination errors.

2.3.2 White Box Testing

In white box testing, the tester needs to know the internal structure of the software and can modify it (if required), see figure 2.3. In this test approach, also called structural tests, the test cases are obtained from the internal structure of the software code under test and the test passes only if the results are correct [11].

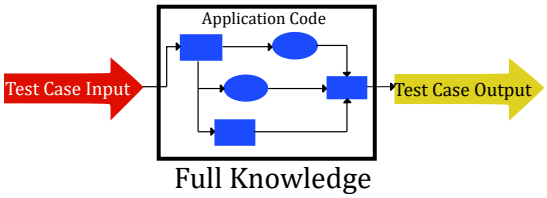

Figure 2.3: White Box Testing.

Using white box testing, the test cases that fullfill any of the following functions can be obtained [11]:

- Exercise all the independent paths of each module at least once.
- Exercise all logical decisions on their true and false paths.
- Execute all loops in their limits and with their operational limits.
- Exercise the internal data structures to ensure their validity.

2.4 Testing Levels

The testing process is involved in each stage of the software life cycle, but the tests carried out at each level of the software's development are different, since they have different objectives. Traditionally, the most common types of testing are unit testing, integration testing, system testing and acceptance testing [12] [13].

- Unit testing: evaluates the software with respect to the implementation.
- Integration testing: evaluates the software with respect to the design of the subsystems.
- System testing: evaluates the software with respect to the design of the architecture.
- Acceptance testing: evaluates the software with respect to the requirements.

Depending on the type of system to be tested, some of the given cataegorizations of the test levels or even others may be taken; it also depends on the focus of the tests that are taken. The two fundamental testing activities are the unit test (tests the parts of the system) and the system test (test the system as a whole) [14], see figure 2.4. Some levels of tests are described in the following sections. In section 2.4.1 the Unit Testing is described. In section 2.4.2 the Integration Testing is described. In section 2.4.3 the System Testing is described. In section 2.4.4 the Acceptance Testing is described.

2.4.1 Unit Testing

Unit tests, also called component tests, are responsible for testing, individually, subprograms, subroutines or procedures in a program [2]. It means, instead of testing the program as a whole, small blocks of the program are tested. The purpose of this type of tests is to compare the functioning of a unit with a certain functional specification that defines the unit. There are different types of units that can be tested [15]:

- Individual functions or methods within an object.
- Object classes that have various attributes and methods.
- Compound components formed by different objects or functions. These components have a defined interface that serves to access their functionality.

The individual functions or methods are the simplest type of unit and their tests are a set of calls to these routines with different input parameters.

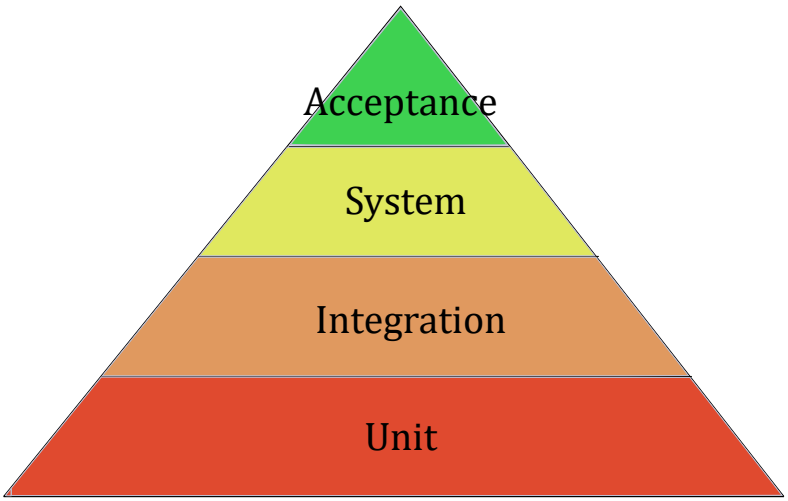

Figure 2.4: Testing Levels.

2.4.2 Integration Testing

Once the components or units were defined and tested individually, they have to be integrated to form the system. Integration tests are a technique that allows us, at the same time that we build the system with the components, to carry out tests to detect errors that arise due to integration and interaction of the individual units [15] [16]. The integration of the system involves identifying groups of components that provide some functionality of the system and integrate them by adding code to make them work together. Some possible problems that can arise from the integration of the components or modules are: loss of data in one interface, one component can have an adverse and inadvertent effect over another, global data structures can present problems, among others. The integration tests verify not only that the components work correctly together, but also that they are called correctly and they transfer the correct data at the precise time through their interfaces [15]. The integration of the components can be incremental or non-incremental. In the non-incremental approach, called big bang, all the modules are combined in advance and the entire program is tested as a whole. This type of integration has the disadvantage that many errors can be found at the same time, which is a great challenge when it comes to correcting them, since it is difficult to isolate the causes and locate where those errors come from by having the entire program under test [11]. In incremental integration, as opposed to non-incremental integration, the program is constructed by small increments or segments in which errors are easier to locate and correct [11]. There are two types of incremental integration: top-down integration and bottom-up integration.

Top-down integration

In this type of incremental integration, first the skeleton of the complete system is developed and then the components are added. In other words, the components are integrated by advancing down the control hierarchy, starting with the main component.

Then, subordinate components are added to the main component. The incorporation of the subordinate components can be done by two ways: depth-first or width-first. The top-down depth-first integration, integrates all the components of a main control path of the system structure. The selection of the main path is done arbitrarily and depends on the characteristics of the system [11]. The top-down width-first integration, integrates all the components directly subordinated to each level, moving through the structure horizontally [11].

Bottom-up integration

In the ascending integration, the system is built and tested starting with the components of the lowest levels of the program structure, for instance, the components are integrated from bottom to top following the hierarchy of control [11]. Due to the way in which the components are integrated, the required functionality of the subordinate components is always available, so it's not necessary to take care of backups.

2.4.3 System Testing

According to Sommervile [14], system tests involve integrating two or more system components and testing them as an integrated system. In an iterative development process, this type of tests is responsible for testing an increment in the system. In a waterfall process, the system testing is responsible for testing the entire system. Pressman [11] states that the test of the system (referring to a system that is incorporated to other elements such as hardware, information, etc.) is constituted by a series of tests whose objective is fundamental in the system. Each test has a different purpose, but all tests work to verify that all the elements of the system have been properly integrated and perform the appropriate functions in a correct way.

Regression Testing

Each time we add a new component as part of an integration test, the software changes. New data flow paths are established, new inputs and outputs can occur and a new control logic is invoked. In other words, the integration and testing of a new component can change the interactions of already tested components and, these changes can cause problems with components that previously worked perfectly. These problems mean that, when integrating a new component, it's necessary to re-run the tests to verify previous increments, as well as the new tests required to verify the newly added component. It's known as regression testing to the process of re-executing a subset of existing tests in order to make sure that the changes have not entered errors or some other undesired effect. Not only a regression testing have to be performed when integrating a new component, but also every time that a major change in the software is made, e.g., by correcting errors that have been discovered by a test [11] [17]. According to Pressman [11], the set of regression testing contains three different kinds of test cases:

- A representative sample of tests that exercises all the functions of the software.

- Additional tests that focus on software features that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

As the integration test progresses, the number of regression tests may grow too. For this reason, the set of regression tests should be designed to include only those tests that address one or more kinds of errors in each of the main functions of the program. It's not practical or efficient to re-run each test of each function of the program after a change [11].

2.4.4 Acceptance Testing

The customer must test the system for acceptance which is done after the defects of the system testing are corrected. In acceptance testing, the customer tests that the system works correctly and verifies that the requirements are fullfilled [18]. Automated software testing is fundamental in the field of software engineering. An organization can implement the process in order to improve the quality of software (QoS) according to the standards and in this way, the efficient of the product can be increased. The software test automation is proved to be the strongest weapon in the complex field of effective software testing [19].

2.5 Test Automation

In literature, many definitions of test automation exist, but perhaps the most fitting one is [20]:

> *"Automation is the integration of testing tools into the test environment in such a manner that the test execution, logging, and comparison of results are done with little human intervention."*

From the definition above, we can understand that the automation is not just to execute some test cases, but also a complete process that helps in the support and side activities of testing as much as possible, although the most common understanding of test automation is just related to the execution of the test cases. There are so many things that can be automated: for instance test data generation, system configuration, simulation, analysis of the results, recording activities and communicating test results [21]. Automated testing is an independent process, for that reason can cover all types of testing (functional, regression, concurrency, etc.), all testing phases and generally, if a test can be run manually then the test can be automated [22]. Automated testing differs from manual testing, for instance, in the following ways [22]:

- It improves manual testing by automating tests that are difficult to perform manually.
- It's in itself software development, it means, there are some artifacts from the application architecture that could be used when the scripts are created.

- It does not replace the need for the analytical skills of the manual testers, the knowledge of the test strategy and the understanding of the testing techniques.
- It cannot be clearly separated from manual testing. In contrast, both automated testing and manual testing are intertwined and complement each other.

Many manual tests can be converted into automated tests but they often must be adjusted to fit to automation [22]. The objective of automated testing is also defined in literature as [22]:

> *"The overall objective of AST is to design, develop, and deliver an automated test and retest capability that increases testing efficiencies; if implemented successfully, it can result in a substantial reduction in the cost, time, and resources associated with traditional test and evaluation methods and processes for software intensive systems."*

In order to run the automation tests, there are many tools that helps to automate the testing process. Nevertheless the testing tool is not the only variable for automation, because the tester also must know which is the aspect of the test to be developed and the impact of performing automation [22]. In this context, the automation tools may support various aspects of the test. Below a possible classification for these tools is given [7]:

- Administration of the tests and the testing process: tools for the administration of the tests, for the tracking of incidents, for the management of the configuration and for the administration of requirements.
- Static tests: tools to support the review process, tools for static analysis and modeling tools.
- Specification of the tests: tools for the design of the tests and for the preparation of test data.
- Execution of the tests: test case execution tools, unit testing tools, comparators, covering measurement tools, security tools.
- Performance and monitoring: dynamic analysis, load and stress, performance, and monitoring tools.

For the automation of the functional tests, the execution tools of the capture and reproduction tests are specially suitable. These tools allow the tester to capture and record tests, then edit, modify and reproduce them in different environments. Tools that record the user interface at the component level and not bitmaps are more useful. During the recording the actions taken by the tester are captured, automatically creating a script in some high-level language. Then the tester modifies the script to create a reusable and maintainable test. This script becomes the baseline and then is reproduced in a new version, against which it is compared. In general, these tools are accompanied by a comparator, which automatically compares the output at the moment of executing the script with the recorded output [23].

2.5.1 Benefits and challenges of automated software testing

With respect to the software agenda, automated tests have a fundamental advantage: time is saved, which means that it reduces the risk of delay of the programmed, improves quality and reduces short and long-term costs. Greater the automation effort, greater the advantages obtained. Although the automation tools are not going to solve all the testing problems, many benefits with proper implmentation can be achieved.
According to Berner [24], the manual execution of tests is inefficient and prone to errors, due to the execution in multiple ocassions of the same test case, allowing that the tester can inject some errors due to the repetition, while the automated execution of the test increases the efficiency in such a way that it reduces the work of the testers and it's less likely to inject errors into the input data, therefore in the test. The automated execution of test cases also helps to reduce the cost because it decreases human participation. According to Pettichord [24], Fewster [25], Kaner et al. [17], and Rice [26] between the common benefits and challenges of Automated Software Testing (AST) are the following:

Benefits:

- Run more tests more often.
- Perform tests which would be difficult or impossible to do manually.
- Better use of resources.
- Consistency and repeatability of tests.
- Reuse of tests.
- Earlier time to market.
- Increased confidence.

Challenges:

- Unrealistic expectations. Managers may believe that Automated Software Testing can solve their problems and improve quality.
- Poor testing practice.
- Expectation that automated tests will find a lot of new bugs.
- False sense of security.
- Maintenance.
- Technical problems.
- Organizational issues.

2.5.2 Approaches to Test Automation

The functional tests that are launched against the graphical user interface, historically, have been one of the areas that most have been tried to automate. Throughout the history of this part of the automation of tests, full of emotions, sorrows, and joys, we can find numerous approaches and strategies, as an example of how this area of software quality has evolved over time, always trying to save time. In 2012, Hunt [27] proposed to classify of the functional test automation approaches into generations. Currently, we can distinguish 5 generations of functional test automation strategies that are launched against a graphical user interface, which are: record and playback, scripting, data-driven scripts, keyword-driven and scriptless (see figure 2.5).

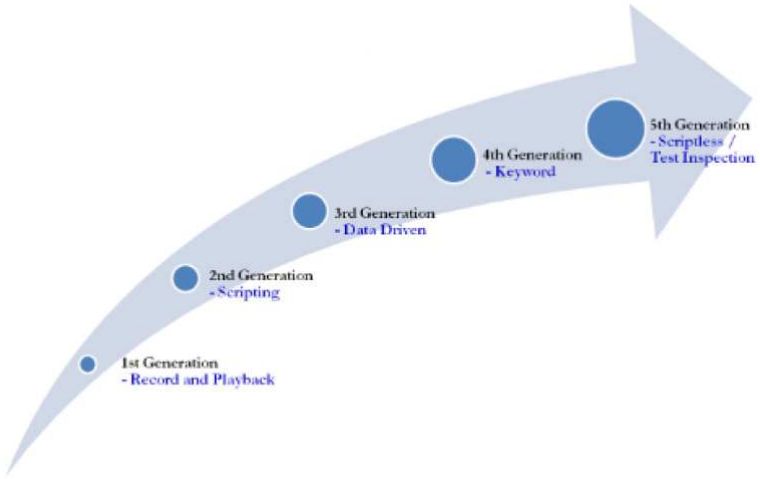

Figure 2.5: Evolution of software test automation [27].

Record and Playback

The first functional test automation tools uses this approach: record the test cases from the interactions of the tester with the graphical interface of the application to be tested. Then, the tools allow to reproduce those test cases. From these recorded actions, the tool itself generates code (so-called scripts) to rerun the test cases. As drawback of this first approach, we can highlight, on the one hand, that each automated test case includes both the actions to be taken and the test data, making difficult to reuse the test case and reducing its maintainability. If we want to automate another similar test case but with different input data, we will have to create a new case. On the other hand, many tools that follow this approach are based on identifiers or labels of the graphical interface elements. If for example we have automated a series of test cases of an application, and the name of any of the elements of the application graphical interface changes (changes the name of a button, a form, etc.), in the next version, the test case would not work, and the version must be fixed or a new one must be created. Some features of this approach are:

- An unstructured way of automation.
- Very low development costs.
- No planning is required.

Examples of tools that use this approach would be the Selenium IDE framework [28], Sikuli [29], EyeAutomate [30] or Test Complete [31].

Scripting

In this approach, scripts are programmed directly in the programming language, and that will call the execution of the test cases. This method is more robust, more flexible and allows reusing the code of test cases, but it must also have to take into account that the elaboration of scripts is more complex and requires that the team that develops the tests have a high knowledge of programming and be more specialized. Some features of this approach are:

- Helps to perform repetitive tasks and allows you to have the ability to call other common functions that have important functionalities of the business.
- Programming costs increase slightly in relation to record and playback.
- Planning is required up to a point.
- Knowledge of scripting language is required.

Tools like UISpec4J [32] would be included in this generation.

Data-Driven Test Automation

This approach separates the automation of tests into two parts: the automation script and the data (both the input and the expected output data after the execution of the test cases). The test data is stored in a separate file, which is read by the script. Each script can be used with different data, encouraging reuse and increasing test coverage. Some features of this approach are:

- The scripts are programmed in a structured way.
- The development costs required are relatively high compared to the previous approaches due to the parameterization efforts and programming.
- More planning is needed.
- The data is stored in data tables or external files (for instance, Excel, Access, SQL Server, etc.).
- The maintenance of the scripts is low.
- Recommended to be used in the testing of positive and negative data if required.

Among others, the Jameleon framework [33] supports this type of scripts.

Keyword-Driven Test Automation

This approach goes one step further than the previous one. In this approach the test data leads the testing instead of the script. Associated with the test data, keyword sequences will be used to indicate which actions must be followed in each case. When an automated test case is executed, it will read the test data and call the script according to the previous selected keywords. In the event that there is a change, it's not necessary to update the entire test case, only the elements that use the keywords should be changed. Some features of this approach are:

- The development costs are high since more efforts are needed for the planning and development of the tests.
- High programming skills are required.
- The initial planning and management efforts are very high.
- The maintenance cost is low.
- You need a framework or libraries to be able to implement it.

The Test Complete tool [31] also supports this type of testing.

Scriptless

It's based on the evolution of the automation of tests by keywords and it does not need to create a script to automate the tests. To build a test case, this method uses objects predefined by the test tool. These objects represent elements of the graphical interface of the application and simple actions. The tester has to select the objects and actions through different menus, sort them and fill their properties for each test case. As they are predefined objects, there is no need to invest time to build them therefore the time to prepare the test cases is reduced. This approach has several benefits: the test cases are reusable and the tools are easy to use since it's assumed that the tester does not see the code that is produced and does not interact with it. Also, since there is no need to program the tests, tools of this type can be used by non-technical people. Even so, we must be careful with this last point: although any person can, a priori, develop test cases using this approach, it's still necessary to have knowledge of how the application is made and what is to be obtained. Another advantage of this approach regarding to the maintainability of the test cases is that if the elements of the application change, only the data model of the used objects must be changed, keeping the initial code. As a drawback, it should be noted that these methods are limited in functionality to the objects that the tool offers by default. To solve this problem, some tools allow adding extensions, but often these extensions can be complicated and require knowledge about programming languages and about the test tool itself.
Tools such as Certify from Worksoft [34] and the Qualitia framework from Zensoft Services [35] support the scriptless approach.

2.6 Considered automation testing tools

The following automation tools are considered within this thesis:

- Selenium IDE
- Microsoft Visual Studio Community 2015 (Selenium WebDriver).
- Eclipse JEE Oxygen (Selenium WebDriver).

The tools mentioned above are described in the following sub-sections.

2.6.1 Selenium IDE

Selenium is an extension mainly of the Mozilla Firefox internet browser (see figure 2.6), but it can also work with Chrome. Selenium runs inside the browser in JavaScript and controls the browser by giving it commands. Selenium based its operation on two features [28]:

Figure 2.6: Selenium IDE workflow.

- Automation by capturing the actions: this indicates that the application allows recording the action performed by the user in contact with the browser and then reproduce it, being able to parameterize the entry data. Although the record and playback tool is an important functionality of Selenium and it's easy to use, generally, the scripts must be modified [36].
- Programming with the Selenium native language: it's also possible to program all the actions performed by the application.

However, it should be noted that the Selenium is released under the Apache 2.0 license, and thus a free and open-source project, it has language bindings for several different programming languages, like Java and C#.

2.6.2 Selenium WebDriver

Selenium WebDriver is a flexible and powerful tool that uses specific native implementations for each browser, thus avoiding the limitations that implies the use of JavaScript code to direct the actions on the System under Test (SUT). Also, it maintains a common access interface that allows a script written in a high-level language (such as Java or C#),

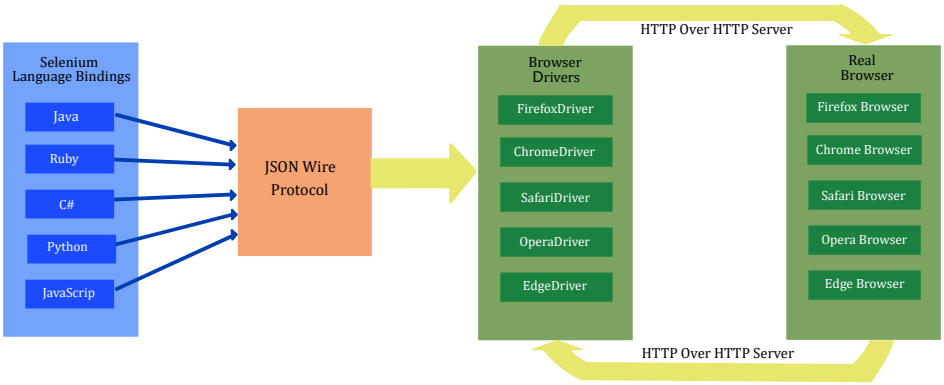

Figure 2.7: Selenium WebDriver architecture.

to simulate different actions on a page, transparently to any browser in which they are executed (see figure 2.7).
If we compare the two versions of Selenium, Selenium IDE and Selenium WebDriver, it can be observed that:

- Selenium WebDriver overcomes the limitations of the IDE when simulating the different actions that a user performs in the browser, during the execution of a certain web application. Now, Selenium is more powerful when using native commands from each browser and eliminating the use of JavaScript instructions, with the restrictions that it implies.
- WebDriver isolates the programmer from the particularities of each browser. Selenium IDE could present different behaviors depending on the browser used to execute the tests. Some tests that worked correctly on Firefox, can not do it in the same way in other browsers, and vice versa. Now, with the existing native implementation of the main browsers, API users are isolated from the particularities of the use of each browser.
- The library that offers Selenium WebDriver is more complex and more powerful. The WebDriver API is available for several high-level languages (Java, C#, Ruby, Python, among others) and is object-oriented, which facilitates its versatility and integration.
- Selenium WebDriver supports the testing of mobile applications. By using IPhone Driver / Android Driver automatic testson mobile applications can be run in different simulators/emulators.

Thanks to the above mentioned advantages, Selenium WebDriver has established itself as an industrial standard de-facto for testing web applications.

Chapter 3

Methodology

This chapter describes the proposed methodology that is performed for the creation of the automated tests in this thesis. The proposed methodology can be modified and adapted according to the type of test to be performed. In section 3.1, the comparison of programming languages is described. In section 3.2, the classification of web applications is described. And finally, in section 3.3, the methodology is presented and explained.

3.1 Comparison of programming languages

In the comparison of this section, Selenium IDE will be not taken into account in the comparison, because this is not a language itself, it's simply an IDE with its own native language, which has many disadvantages [28] compared to other programming languages more robusts as Java and C#. There is no major difference between Java and C# when comparing them at the level of automation, there are just some syntactical differences as shown in table 3.1.

Java	**C#**
WebDriver to create the browser instance.	IWebDriver to create the browser instance.
WebElement to find out the elements in the web application.	IWebElement to find out the elements in the web application.
@FindBy annotation is used in Page Objects to specify the object location strategy.	[FindBy] annotation is used in Page Objects to specify the object location strategy.

Table 3.1: Syntax differences between Java and C# when Selenium is used.

At the level of automation, there are also other differences are mentioned in table 3.2. Although Java is most widely used, it's very difficult to find a big difference to choose one or another language.

Basis for comparison	Java	C#
Online help	Plenty of websites are available for that we just need to use google.	Less online help available as compared to Java.
IDE	Eclipse is widely used for Java coding.	Visual Studio is used for C# coding.
Framework	JUnit	NUnit
Report	We just need to use ANT to generate Test Execution Report.	Manual efforts are required to create a Test Execution Report in it.

Table 3.2: Differences between Selenium WebDriver with Java and C#.

Survey

To compare the languages, it was used a survey for determining which language (between Java and C#), is more commonly used for automation in the industry, and determine if one of the two can be easier to learn with respect to the other. The main objectives of the survey was to find out what is the most common language used by QA testers when it comes to automating the tests and knowing what is the perception of them when they had to learn it. The research questions for the survey are presented in table 3.3 and the results of the survey are presented in 5.6.2. It was selected *www.surveymonkey.com* as the tool for conducting the survey. An online link was generated by this tool for accessing the questionnaire. As it is an online survey, it was tried to reach as many professionals as possible, which resulted in getting 63 respondents in total. The position and the company information obtained from the respondents is used to find the valid responses for the survey. The respondent should be a tester or similar, to make his response as a valid response.

3.2 Classification of web applications

There are many ways to classify a web application, for instance, according to its architecture, its content or its technology, nevertheless none of these classifications are useful at the time of testing. For that reason, after doing the analysis for finding a way to classify the applications, it was decided to create a classification that would serve to validate the proposed automated testing methodology. The classification is not based on the web application to which will perform the tests instead is based on existing testing levels, which are: Unit testing, integration testing, system testing and acceptance testing [37]. These levels were presented and explained in section 2.4. The tests in the fourth-level (Acceptance testing) shouldn't be automated, because the tests are performed by the user, and the user needs to see and have the control of the application, to be sure that it's working as expected. Therefore this level of testing should not be automated, since it's not performed directly by the tester, but is performed only by the user. However sometimes it can be done in the company of the tester, for getting the user perception

Number of Question	Research questions	Motivation
Question 1	Which of the following programming languages do you know?	Know which of the programming languages is best known for performing automation
Question 2	What is your cumulative programming experience in this (these) language(s)?	Know the experience of people in the use of the language
Question 3	How long have you been using an IDE for C# or Java programming?	Know the experience of people in the use of an IDE
Question 4	What framework do you use to create your automated test scripts? You can choose more than one option.	Identify the most used software automation framework when creating scripts
Question 5	What programming language do you think that you learn faster?	Know the language that is perceived to have a lower learning curve

Table 3.3: Research questions for the survey.

about the application and to know if the user finds an issue. Inside the methodology, the applications considered are divided into three categories, depending on the types of testing that are applied to them. Those applications that contain only unit testing, will be considered as small applications, because the unit tests, are the smallest tests that can be done to an application, it means, the unit tests are tests of the smallest level according to the same application. Those applications that contain only integration testing, will be considered as medium applications, because the integration tests, are the medium tests that can be done to an application, it means, the integration tests are mid-level tests with respect to the same application. And finally, those applications that contain only system testing, will be considered as large applications, because the system tests, are the large tests that can be done to an application, it means, the system tests are tests of the highest level with respect to the same application. It's important to clarify that although the highest level is the acceptance testing, these are not considered because they are usually performed by the user and not directly by the tester. In chapter 4, three applications are shown following this classification, where the small is a web application where the automation is to perform a unit testing. The medium is a web application where the automation is to perform an integration testing, and the large is a web application where the automation is to perform a regression testing (part of the system testing).

3.3 Description of the methodology

The complexity of the tasks that must be developed for an automated test to be useful is such that it deserves to be treated as a project in itself. In this section, the methodological

process followed to perform automated tests is presented, which can be modified and adapted according to the type of automated test to be performed. The process is divided in five sequential stages, with the peculiarity that the fourth stage is iterative. These stages and their main activities are shown in figure 3.1 and they are described in the following subsections.

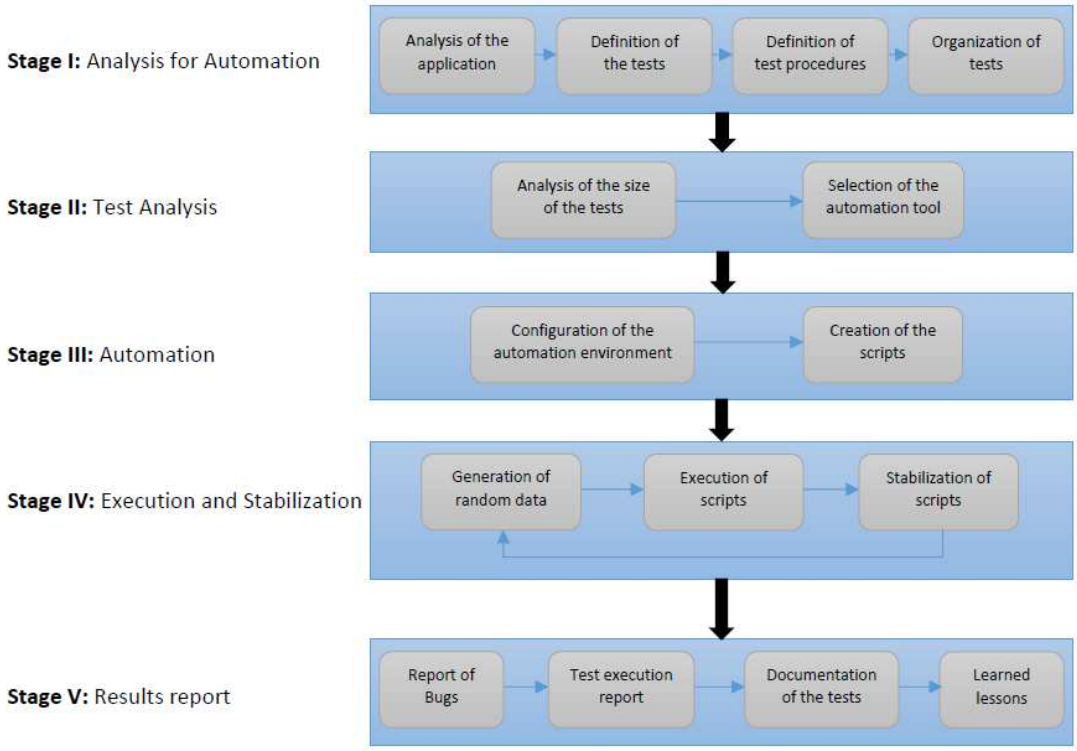

Figure 3.1: Stages and main activities of the proposed methodology.

In the following subsections, the main activities of the five phases in the proposed methodology are presented. The section 3.3.1 describes the stage I, named analysis for automation. The section 3.3.2 describes the stage II, named test analysis. The section 3.3.3 describes the stage III, named automation. The section 3.3.4 describes the stage IV, named execution and stabilization, and finally, the section 3.3.5 describes the stage V, named results report.

3.3.1 Stage I: Analysis for automation

In stage I, the following activities are described: analysis of the application, the definition of the tests, definition of the test procedures, and organization of the tests.

Analysis of the application

It's necessary to evaluate if the effort invested in the automation of the tests is profitable in the face of the benefit obtained. It's not advisable to carry out automation when changes are foreseen since the life of the automated test is very short. The objective when automating a test is that it has a life of several executions so that the time spent in automation is profitable compared to the time gained when executing the tests manually. In cases in which the execution of the tests extends a lot in time, and the tests to be automated are few, the time invested in the automation of them will not be recovered. It is possible that the tests to be performed in an application need to be executed on different deployment environments (different browsers, application servers, software versions). In this case, even if the number of tests is reduced (and it can be thought that it is not optimal to automate them), its execution must be repeated several times so it can compensate for its automation. So taking into account these points and once seen if automation is recommended, the next step is to define the tests to automate.

Definition of the tests

Test cases cost time and money, the set of test cases that many testers choose are those that meet coverage goals, such as source code coverage, code declarations or ticket coverage. In this phase, the testers are more interested in the execution paths, sequences of code declarations that represent a flow of the application, looking for the set of scenarios that will guarantee to find most of the errors. In addition, the test scenarios can ensure that the software works according to the specified requirements.

Definition of the tests procedures

The objective of this activity is to define the suites and scripts that will make up the automated tests. From the functional cycles and functionalities selected for the tests, the suites and scripts, that will execute them, are specified. Defining the suites implies defining the scripts that compose them and specifying possible execution dependencies between them. For each script the scenarios to be performed and their verifications must be defined.

Organization of tests

After defining the test scenarios and test procedures, it must be defined the order of execution of the tests; this is very important in the automation, because in this methodology, all the test scenarios will be executed sequentially, it means, using the same driver (without having to sign off, close the driver, open the driver, sign in), in this way the time execution is shorter and the next test scenario only start its own test, from the endpoint of execution of the previous test, minimizing the execution time and resources. The test may lose independence in this way, whence all the scenarios must be defined very well, in order to be clear about each of them.

3.3.2 Stage II. Test analysis

In stage II, are described the following activities: analysis of the size of the tests, and definition of the automation tool.

Analysis of the size of the tests

The size of the tests is decided according to the classification presented in section 3.2.

Selection of the automation tool

The main idea of this thesis is compare some of the test automation programming languages to be able to define the correct automation tool according to the application to be automated, therefore, for the application of the methodology, the automation of the applications will be carried out in the three languages that have been mentioned in the course of this thesis. Which are: Selenium IDE, Selenium Java WebDriver and Selenium C# WebDriver. The results are shown in table 5.4.

3.3.3 Stage III. Automation

In stage III, are described the following activities: configuration of the automation environment, and creation of the scripts.

Configuration of the automation environment

The objective is to configure the environment of the application that is going to be tested in order to execute the suites correctly and document this configuration. In this activity, all the data that the applications used in order to execute correctly the suites, must be configured. It's important to document this configuration in sufficient detail.

Creation of the scripts

The objective of this activity is to create the suites together with their corresponding scripts. This activity consists in assembling each one of the suites with the corresponding functional cycle to be automated. The tests are recorded or encoded, obtaining, as a result, the scripts that make up the suites. The correct operation of each script is verified.

3.3.4 Stage IV. Execution and stabilization

In stage IV, are described the following activities: generation of random data, execution of scripts, and stabilization of scripts.

Generation of random data

The objective of this activity is to generate the data for the execution of the tests. If it's a small application, the data should be generated using the black box testing techniques [38]. If it's a medium application, the tester must analyze and decide the best option

to choose, either using black box techniques or generating data randomly. If it's a large application the most advisable thing is to generate the data randomly. When data is generated randomly, it's taken from the application to be tested, so it must be verified that there is enough data within the application to be able to perform the tests.

Execution of scripts

The objective of this activity is to execute a complete test of the suites corresponding to the functional cycle and verify its correct operation. This activity also consists in carrying out the complete test of the functional cycle in the environment prepared for that purpose. In case of a malfunction, the necessary adjustments must be made. The behavior of the suites as a whole is verified.

Stabilization of scripts

The objective of this activity is to verify the correct behavior of the scripts in the testing environment and prepare the suites and scripts generated for the validation. In this activity, scripts are tested and eventually adjusted for proper operation in the testing environment. It's validated by comparing the tests that are expected to be automated with the tests that the scripts perform. To do a correct stabilization, the scripts must be executed several times in order to verify that the scripts are able to run more than once.

3.3.5 Stage V. Results report

In stage V, are described the following activities: report of bugs, test execution report, documentation of the tests, and learned lessons.

Report of bugs

In this section, the most common types of defects in programming will be presented. It will show a classification according to the degree of severity and the classification that will be used in the report of bugs. The severity of a bug can be determined based on the damage caused to the operation of the system. It's important to mention that the degree of severity of a bug varies depending on the software type where it is presented. For example, a catastrophic defect for a nuclear system means that the failure can result in deaths or environmental damage; while a catastrophic defect for a database system means that the failure can cause loss of valuable data. That is why there is not a standard that defines a classification of this type, but in each system is determined the degree of severity of the defects based on the context in which it's applied. An example of this classification of bugs according to their degree of severity is shown below [39]:

- **Catastrophic:** bugs that can cause several serious damages. Example: security problems.
- **Major:** bugs that can cause serious consequences such as the loss of important data.

- **Minor:** bugs that can cause small or insignificant consequences. Example: display the results in a different format than expected.
- **No effect:** Bugs of this type may not cause an impediment in the execution of the system, but may lead to a different and generally avoidable interpretation. Example: simple typographical errors in the documentation.

The bugs found during the execution of the automation test must be reported for correction and to follow a good practice of the QA process, the following fields are present:

- An identification number
- A clear, concrete and short title
- Explain clearly what the problem is, what was the expected result vs the obtained result
- Reporter
- Assigned user
- Version
- The environment where the bug was found
- Component/module over which the fault was detected
- Platform and Operating System
- Browser
- Priority of the correction
- The severity of the bug
- If it's necessary to attach evidence (screenshots, logs, documents, etc.) or any complimentary material that helps the developer to solve the bug
- State in which the bug is located, to indicate if it can be reviewed by the developer, has already been solved or has not yet been reviewed

Test execution report

A final report of the project must be made including the methodology followed throughout the project, the number of automated test scenarios, the executions carried out, the percentage of coverage of the tests, the improvements made on the system and the errors found during the execution of tests. In addition, a compilation of all the information and test artifacts created throughout the project must be carried out.

Documentation of the tests

Documentation of the tests carried out must be done, even if it's short, it must indicate the new functionalities added to the project, with a little explanation of how the funcionalities works and it's useful, some images can be inserted to explain better the changes, like a user documentation.

Learned lessons

Finally, it is good to analyze internally within the team that has made the tests, what were the lessons learned, the mistakes made during the execution of the project and all the information that may be relevant for the improvement of the testing process.

Chapter 4

Description of the case studies

In the following subsections, the application of the proposed methodology will be analyzed in three different case studies categorized as small, medium or large. The criteria for making such a classification was described in section 3.2. An explanation of each of the case studies can be found, in order to have a better understanding of the operation of the applications and the automation of the tests carried out.

4.1 Small application

The tests that must be done in the small application named Scheduler, will be only unit tests. The test consist in the creation of a new button, which opens a new screen to add tasks ($TS1_s$) and it can be seen in figure 4.1. The most important thing in the creation of the tasks is the output format, where the two valid formats are XML and JSON. In addition, the execution frequency of the task also can be chosen as follows: Perform the task only once or do it within a specic time period. The file is generated as a result of the data typed in the form, and this data is taken directly from the database and sent according to the delivery mechanism selected by the user. The email can be selected to send the information directly to a provided mail address, or save the file in an FTP server, where all the necessary data will have to be given so that the application can perform the automatic login to the server and store the file.

4.2 Medium application

Redmine is a tool for project management that among other funcionalities, allows users of different projects to track and organize them. It's also possible to optimize the operation of the application by adding functionalities to it. It includes an incident tracking system with bug tracking. The figure 4.2, shows a little overview of the application. In this application, we are only interested in the integration of the following tabs:

- Login ($TS1_m$): It's used to initialize the session into the application.
- Overview ($TS2_m$): As its name says, this tab is responsible to show an overview of

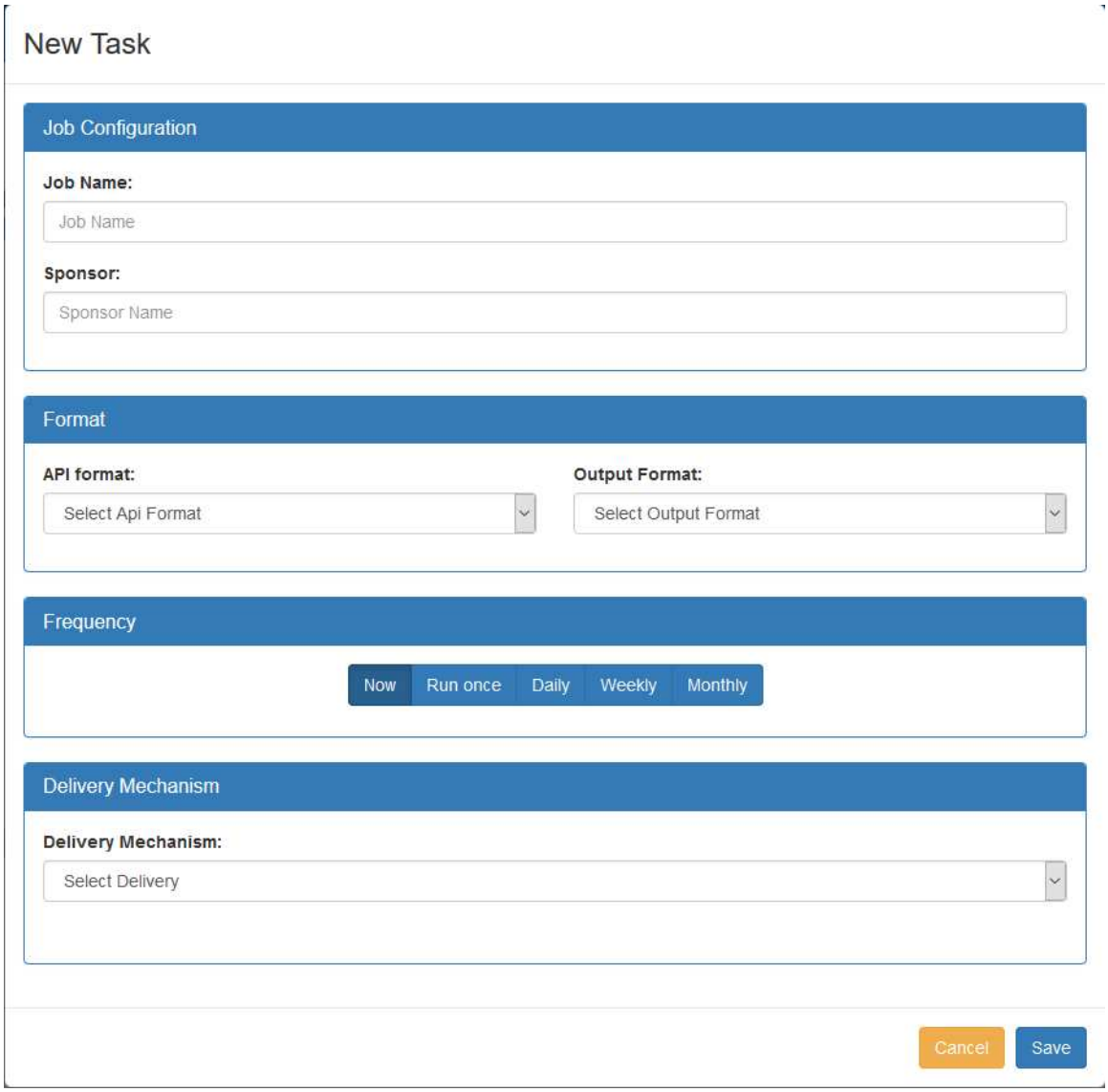

Figure 4.1: Add task functionality for the small application.

all the tickets created for the project, for the QA process, only the first three are used; which are bug, task, and fix.

- New ticket ($TS3_m$): In this tab, we can create a new ticket (same of a bug for QA process) and add all the important information to be able to reproduce and solve it.
- Search ticket ($TS4_m$): It's a functionality to search for all the tickets created in the application. It's possible to search using the ticket number, the description, the subject or the tracker.

- Time tracking ($TS5_m$): This tab allows (by default) to see a table with all the tickets assigned to a user, and if a specific ticket is chosen, the spent time in it during a given day can be added.

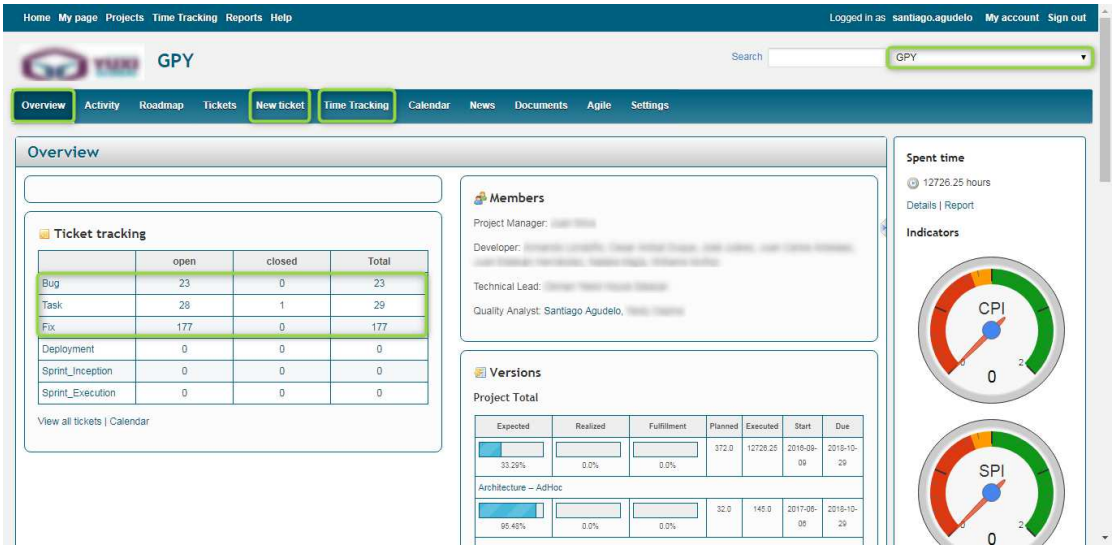

Figure 4.2: Overview functionality for the medium application.

The followed process to do the automation of the test scenarios was in a first time to take the data of the tickets in the overview tab. Next create a new ticket where the tracker is equal to bug, task, or fix. And finally, look for the ticket created and add some spent time to that ticket.

4.3 Large application

Edere is an application to create medical studies through a workflow. The idea is to create from scratch a new workflow with several stages, these stages are the workflow that the study should continue in order to be completed, but not all the users can edit the study, that is why users must be created, add roles, pages, and functions, to determine the functionalities to which a user has access (security tab). If you do not want to create the workflow from scratch, there is the possibility to make an import of one or more workflows using the import tab. In addition, the user can search the studies applying different types of filters using the Browse & Search tab. Additionally, in the dashboard tab (see figure 4.3), by default the user can see all the medical studies that are assigned to him, and then he can choose one to follow the whole process and finish the workflow; this is only done if the user has the necessary permissions to add the corresponding information, and perform the approval of each of the workflow tasks. The description of each tab is explained below:

- Login ($TS1_l$): It's used to initialize the session into the application.

- Security ($TS2_l$): This tab allows to create and manage all the permissions of the application users.
- Workflow ($TS3_l$): It's a functionality to create and manage the progress of the medical studies.
- Metadata ($TS4_l$): Allows to create and manage the functions and sets inside the application. The functions and sets can be added or attached to a workflow.
- Import ($TS5_l$): As its name says, the functionality of this tab consists in import external workflows within the application, since they contain a file with the correct extension.
- Browse & Search ($TS6_l$): It's used to search the studies inside the application, allowing the user to use different filters in the search.
- Dashboard ($TS7_l$): This tab is the first screen that the user see when the application is opened and allows to have an overview of all the projects, these are separated in two different tables, the one lets to see the projects assigned to the current user and the second to see all the other projects.

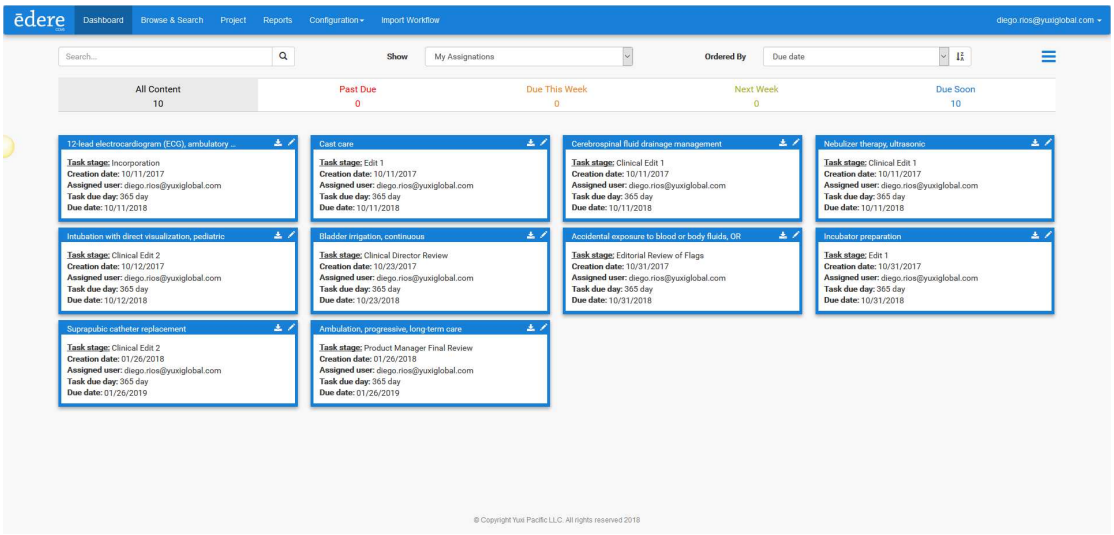

Figure 4.3: Dashboard functionality for the large application.

4.4 Implementation of the methodology in the selected applications

Following the methodology, the automation tests for the three applications are analyzed according to each proposed stage, as is shown below.

4.4.1 Stage I: Analysis for automation

In the following subsections, it's described the analysis for automation made for each application.

Small Application

There were some changes in the sprint for this application, but the majority of the tests were minors or did not suggest furher benefits, therefore during the analysis, the changes were discarded for automation, and just the add task funcionality was defined. This task is useful because a file is generated automatically, and although the interface to create the task is not likely or is not expected to change in time, the file structure will change, adding new features inside the file, for that reason, the filemust be generated several times in upcoming changes, as long as we review the structure of the file, and it is a work that is done manually.

Medium Application

The goal with this application is to test all the functionalities within some tabs, it means, the integration of the different modules of the tabs, for which integration tests will be carried out. The tests consists in the life cycle of a ticket. It was decided to automate these tests because this are the most important features in the application and other changes can impact their behavior and furthermore each time that a regresson test is done, it's likely that these tests must be executed. The procedure to be followed in is basically to create the ticket, search the ticket and then insert some work time to the ticket, it means that the number of the ticket that is created automatically by the application, it's very important to perform the other tasks.

Large Application

In this project, there was a document for the regression testing previously created between the client and the tester. Initially the test scenarios and procedures were defined and the regression was performed manually. It was decided to automate these tests because the regression had to be performed every two or three months, and the test was almost always the same. Having the regression testing in an automated way will help to reduce the execution time and to perform the tests several times, if necessary.

4.4.2 Stage II: Test analysis

As can be seen from the description of the applications, the tests performed for the small application are unit testing, for the medium application are Integration Testing between the modules of the application, and for the large application the performed tests are Regression Testing. As already mentioned, the tools used to perform the automation, were Selenium IDE, Selenium Java and Selenium C#, for getting some metrics to be able to decide which is the best automation tool to use, depending on the type of application.

4.4.3 Stage III: Automation

For the three applications, a test enviroment used only by the testers was selected. Also, the creation of the scripts was performed using the same logical sequence, but obviously, the syntaxes and the commands changed depending of the language. For Selenium IDE some considerations were taken: additional extensions and a Javascript code were used in order to implement decisions (if/else) and to get the execution time, respectively. Also, it was used another extension to store all the constants into variables at the beginning and just to use the variables to type the data. In this context, only one script was created for the small appplication in Selenium IDE, five scripts for the medium application, and seven scripts for the large application. Regarding to the languages Java and C#, it was necessary to create a project using Eclipse and Visual Studio respectively, in this case the scrips created increased because three classes were necessary to manage the driver, the test suite and other additional methods. To point out, it was implemented the page object pattern, whereby each script in Selenium IDE, will have an additional class in these two languages, just to implement the pattern, it means, a total of five classes were created for the small application, thirteen classes for the medium application, and seventeen classes for the large application.

4.4.4 Stage IV: Execution and stabilization

All the three automation techniques (in the three applications) were executed using its own test environment and were executed several times to be sure that the automation was stable, all the elements were reached with the locators inside the automation and the logical implemented was correct for the tests performed. During the execution, all the bugs that were found, were immediately reported using the test rail application.

4.4.5 Stage V: Results report

In this stage, four reports are generated, where each one had a different objective. The first one is the report of bugs used for indicating the bugs found. For the small application none bug was encountered. For the medium application one bug was encountered, and for the large application five bugs were encountered. The second one is the report of test execution that indicates all the summary of the test performed, in this report is also included the test performed manually. For the small application, there are only two additional test scenarios that are “delete” and “edit” a task. For the medium application, there are another additional test scenarios that will not be explained in this thesis. Additionally, in this document, it’s included the finding and suggestions encountered during these tests, where a finding is defined as an error that was inside the application but was no part of the tests performed, and a suggestion, is just an improvement or a change that can be done to the application, but it’s not mandatory because the user can coexist with the error. For the small application, the tests are performed during a project that is working using the Scrum framework, so it’s necessary to update all the documentation related to the test performed, such documentation is more like a user documentation, where the new funcionalties and how they work, are inidicated. For the medium and large applications, the documentation of the projects is updated indicating the changes and

reporting the scripts created, and giving the path where these scripts can be reached. Finally, in the fourth report, for the small application, the lessons learned during the project were shared with the team. In order to improve the testing process, an email was sent to the participants of the project, for recording the lessons learned and implementing them during the next sprint. For the medium and large applications, there was not a document for the lessons learned.

Chapter 5

Results

Before the execution of the scripts, the enviroment tests must be set up according to the automation tool to be used for the tests and the programming language. Therefore, in the first part of this chapter the relevant aspects for the configuration of the enviroment tests are given, including a brief description of the automation tools, the requiered extensions and framewoks, in order to introduce the reader and allow him to have a better understanding of their functionalities. In the second part of this chapter, the results of the execution of automation and the analysis of the results for each tool are shown. The tools used for automating the test scenarios were: Selenium IDE, Selenium java WebDriver, and Selenium C# WebDriver.

5.1 Selenium IDE

As explained in the section 2.6, Selenium IDE is an extension of the Internet browser "Mozilla Firefox". This is a software for automation of processes or tasks for the internet browser, it's based on the capture of actions and programming with the native language of Selenium. As it's described in chapter 4, it was used to create the automated test scripts for the three applications each one with a different complexity grade. Implementing automated tests for the small application in Selenium IDE (see Figure 5.1) was a quite easy if know the logic of decision structure (if/else), but the implementation of the other two test suites was more difficult.
The logic of if/else decision structure can be added to Selenium with an add-on. Also, it's important to say that Selenium executes its commands one after the other and it's no possible to call a method or another instruction in a different file. Thereby, the implementation of the test scenarios for all the applications was done in the IDE, nevertheless there were some limitations or difficulties that increased the creation time of automation scripts. Additionally, the test scenarios were created to be executed sequentially, i.e., the execution of test cases is based on the outcome of the previous one whose execution has already been completed. In order to accomplish the last feature in Selenium IDE, it must execute the test using the "Play entire test suite" button or execute each test case separetly, but it implies going manually to the screen where this test case starts.

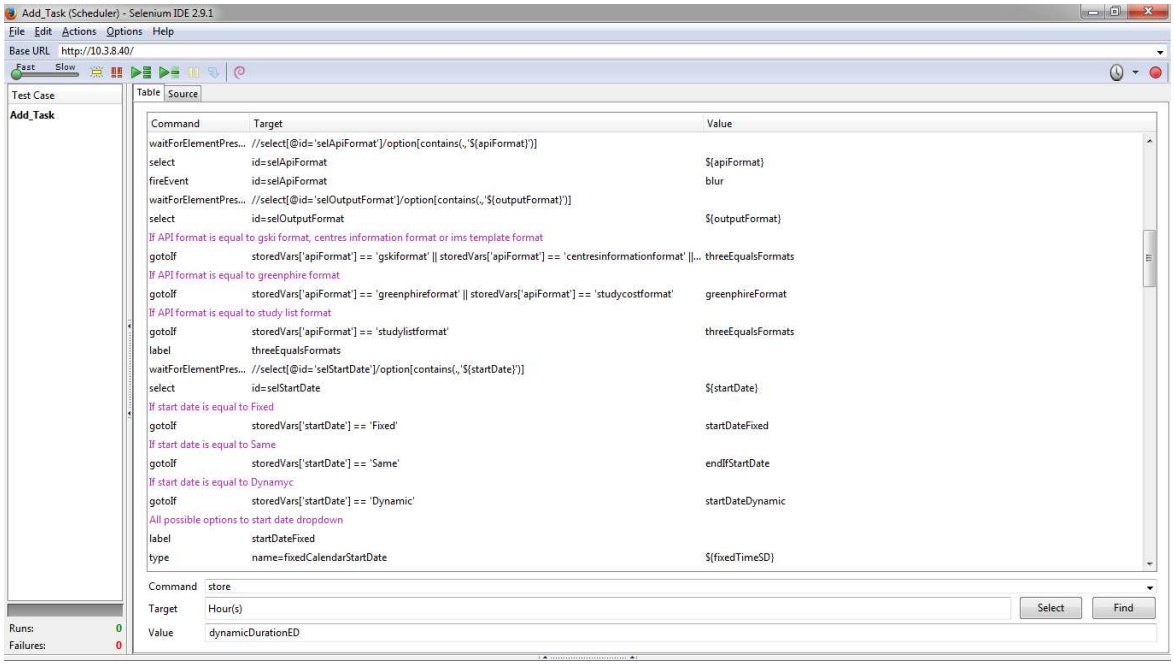

Figure 5.1: Script in Selenium IDE for the small application.

The software specifications of the Selenium tools are detailed below:

Feature	**Description**
Browser	Mozilla Firefox 52.0.1.
Selenium IDE	2.9.1

5.1.1 Add-ons

There are some add-ons that can be installed in Firefox, in order to do easier the automation, the most important and useful extensions are detailed below.

- **Firebug 2.0.19:** Firebug allows to analyze the HTML code of the page that the browser is showing at that moment, allowing to locate each of the elements of the web page, selecting them with the mouse cursor.

- **Firepath 0.9.7.1.1:** Firepath is an add-on that adds an important function to Firebug, allowing to locate elements in a web page through its XPath route.

- **SelBlocks 2.1.1:** SelBlocks allows to use structure commands in Selenium IDE, such as if, else, while, etc.

- **Low Control 1.0.4.1:** Low Control allows to use jump commands in Selenium IDE, such as goto, label, gotoIf, etc.

- **Stored Variables 2.0.1:** Stored Variables allows to create and manage variables in Selenium IDE.

From the add-ons mentioned above, Firebug and Firepath add-ons are also useful for automation using Selenium WebDriver.

5.2 Java Selenium WebDriver

In figure 5.2 the class diagram of the medium application can be seen. This diagram shows how this case study was implemented using Java Selenium WebDriver.
The software specifications for the Java Selenium WebDriver and some tools used, are detailed below.

Feature	Description
Driver	GeckoDriver 0.18.0
Java IDE	Eclipse Oxygen.3a Release (4.7.3a)
Maven	3.3.9
Junit	4.0
Selenium WebDriver	3.8.1

5.2.1 Maven

Maven is mainly a compilation tool for java projects that can generate different types of binaries. Normally, the default output of a Java project is a JAR file. Maven provides plugins and phases of the lifecycle to generate different types of binary artifacts for Java projects. A JAR project combines all source classes together with the required project resources in a single field. This JAR file can be distributed for using it elsewhere [40]. The idea above described can be complicated to understand but Maven makes the work easier, since it provides a *pom.xml* file where the dependencies can be added. The dependencies are all external JARs that are need to run the project, and adding these dependencies into the file, maven add automatically the JARs to the project, for in the opposite way, if the depency need to be removed, Maven also remove the JARs automatically. Maven also has some advantages when a project need to be moved (imported) from one computer to another. In this case after creating a new maven project an importing it, maven will add automatically all the JARs from the dependencies. For the reasons mentioned above, Maven is used as a tool for adding the dependencies into the automation projects created in this thesis. For a better understanding of the tool and the use of dependencies, in figure 5.3 an example is given.

5.2.2 JUnit

JUnit is a framework for writing and executing automated tests for Java applications. Among its advantages, JUnit allows to insert assertions (Similar to Selenium IDE), which facilitate the comparison of one value with another using a condition, for veryfing the result. Also, the scenarios or test cases in suites can be grouped in order to have a

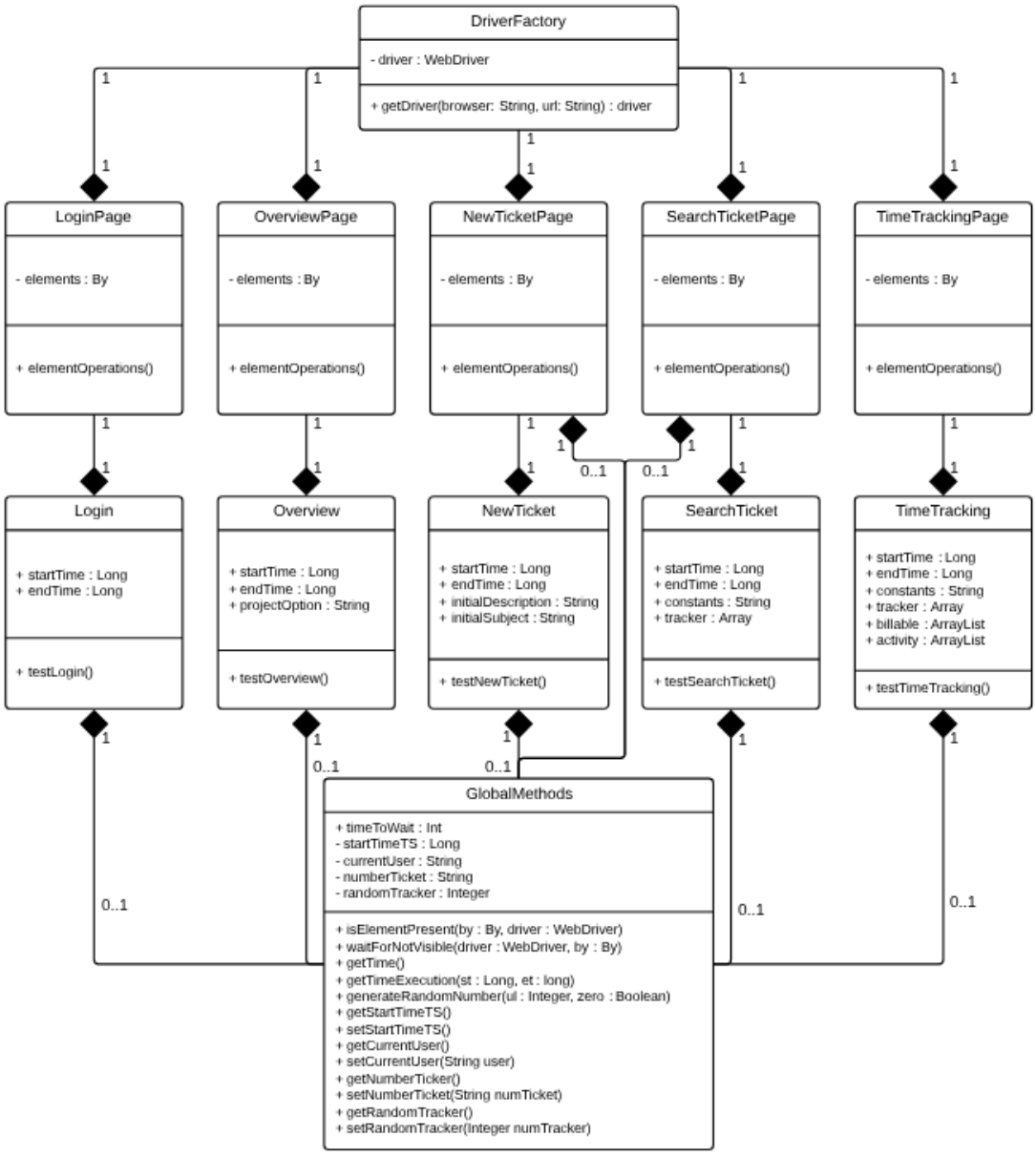

Figure 5.2: Class diagram of the medium application in Java.

better organization of the tests [41]. In figure 5.4, the structure and the characteristics of the *TestSuite.java* class for the medium application are described followed by a brief description of each one. This class allows to run the complete test suite using the JUnit Framework.

- **RunWith:** It's an annotation where the *Suite.class* is indicated, this main class

RedMine/pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>Redmine</groupId>
    <artifactId>Redmine</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>Redmine</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.seleniumhq.selenium</groupId>
            <artifactId>selenium-java</artifactId>
            <version>3.8.1</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Figure 5.3: Pom.xml for the medium application.

contains the classes inside the suite.

- **SuiteClasses:** Add a reference to JUnit test classes where are indicated all the classes that will be executed. The classes should have a Test annotation.
- **Test:** It's an annotation in a method, all the code inside this method will be executed as a JUnit test.
- **BeforeClass:** It's a setUp() function that is executed at the beginning of all test cases.
- **AfterClass:** It's a tearDown() function that is executed after all test cases have finished.

5.3 C# Selenium WebDriver

In figure 5.5 the class diagram of the large application can be seen. This diagram shows how this case study was implemented using C# Selenium WebDriver.

TestSuite.java

```
1  package testSuite;
2
3  import java.net.MalformedURLException;
18
19 @RunWith(Suite.class)
20 @SuiteClasses({
21     Login.class,
22     Overview.class,
23     NewTicket.class,
24     SearchTicket.class,
25     TimeTracking.class,
26 })
27
28 public class TestSuite {
29     public static WebDriver driver;
30     protected static DriverFactory driverFactory;
31     protected static String baseUrl;
32     protected static String browserName;
33
34     @BeforeClass
35     public static void setUpClass() throws MalformedURLException {
36         driverFactory = new DriverFactory();
37         baseUrl = "http://redmine.yuxipacificmed.com/";
38         browserName = "Chrome";
39         driver = driverFactory.getDriver(browserName, baseUrl);
40     }
41
42     @AfterClass
43     public static void setDownClass() {
44          driver.quit();
45     }
46 }
```

Figure 5.4: Test Suite in Java for the medium application.

The software specifications for the C# Selenium WebDriver and some tools used, are detailed below.

Feature	**Description**
Driver:	GeckoDriver 0.18.0
Visual Studio:	Microsoft Visual Studio Community 2015. Version 14.0.25431.01 Update 3.
Nunit:	3.10.1
Selenium WebDriver:	3.14.0

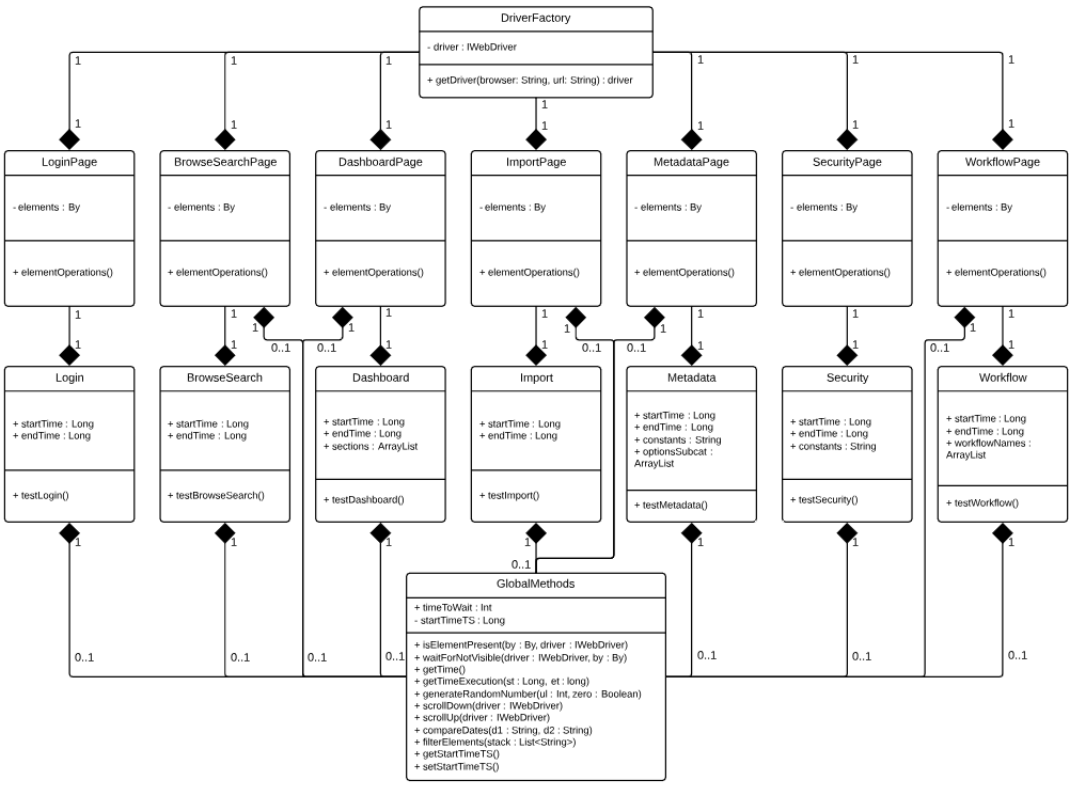

Figure 5.5: Class diagram of the large application in C#.

5.3.1 NuGet Package

NuGet is an extension of Visual Studio that makes it easier to add, remove and update references to libraries and tools in Visual Studio projects that use the .NET Framework. When a library or tool is added, NuGet copies the files to the solution and makes automatically makes the changes that are necessary for the project, such as adding references and changing the file *app.config* or *web.config*. When a library is deleted, NuGet deletes the files and reverses the changes it made to the project. NuGet provides a quick and easy way to add features to an existing application as long as these features are integrated into source code control. In figure 5.6 an example of the use of references in NuGet is shown.

5.3.2 NUnit

NUnit is an open source framework for writing and executing automated test cases for supporting all .NET languages. In order to develop automated tests using NUnit, a framework and text editor are required. Nevertheless, the text editor can be replaced by an integrated development environment, for a more convenient work.
In figure 5.7, the structure and the characteristics of the *TestSuite.cs* class for the large application are described followed by a brief description of each one. This class allows to

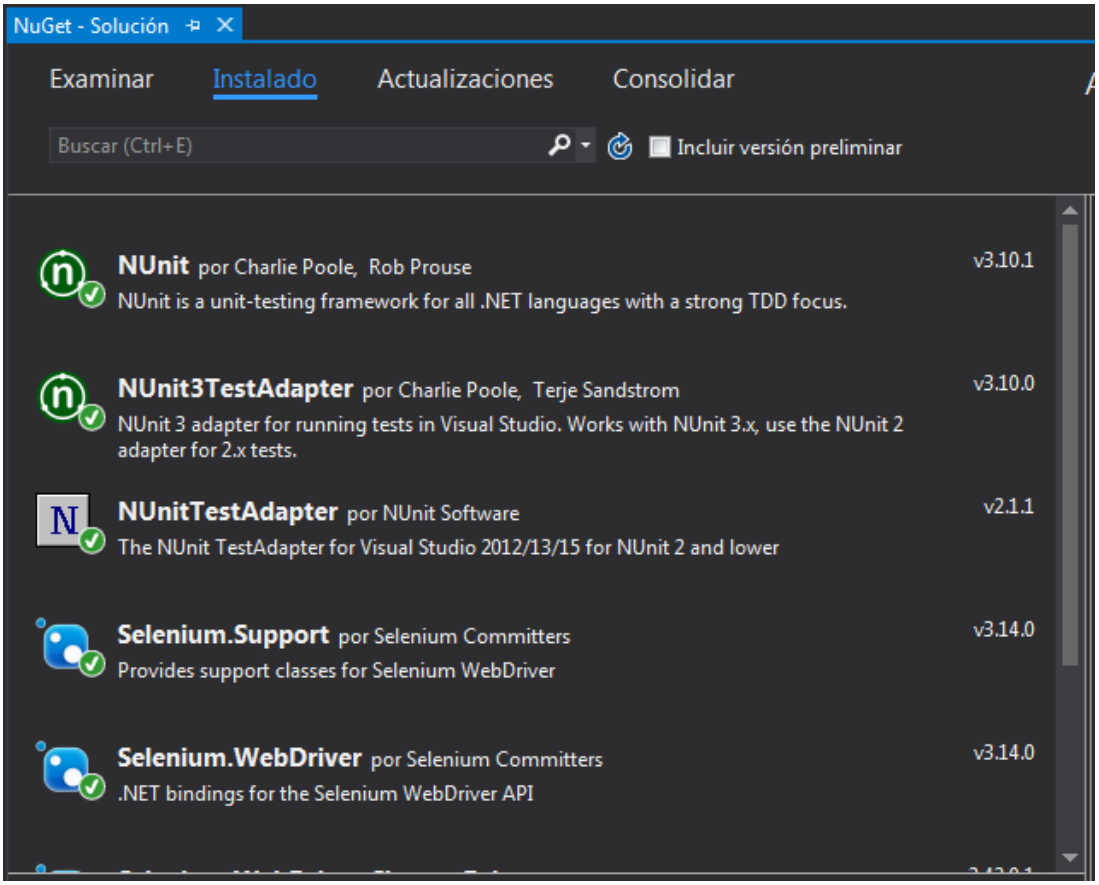

Figure 5.6: NuGet Package for the three applications.

run the complete test suite using the NUnit Framework.

- **[TestFixture()]:** It's an attribute where you indicate the class that contains the tests.
- **[OneTimeSetUp]:** It's an attribute to identify a function that is executed once in the beginning of all test cases.
- **[Test]:** It's an attribute in a method inside a Test Fixture class, all inside this method will be executed as a NUnit test.
- **[OneTimeTearDown]:** It's an attribute to identify a function that is executed once after all test cases have finished.

testSuite.cs

Scheduler

Scheduler.src.test.testSuite.TestSuite

```
1   using OpenQA.Selenium;
2   using NUnit.Framework;
3   using Scheduler.src.main.utility;
4   using Scheduler.src.test.test;
5   using System;
6
7   namespace Scheduler.src.test.testSuite {
8
9       [TestFixture()]
10      public class TestSuite {
11          public static IWebDriver driver;
12          private static DriverFactory driverFactory;
13          protected static String baseUrl;
14          protected static String browserName;
15
16          [OneTimeSetUp]
17          public void Initialize() {
18              driverFactory = new DriverFactory();
19              baseUrl = "http://10.3.8.40/WebSiteNewUI/";
20              browserName = "Chrome";
21              driver = driverFactory.getDriver(browserName, baseUrl);
22          }
23
24          [Test]
25          public void AddTask() {
26              AddTask addTask = new AddTask();
27              addTask.testAddTask();
28          }
29
30          [OneTimeTearDown]
31          public void endTest() {
32              driver.Quit();
33          }
```

100 %

Figure 5.7: Test Suite in C# for the small application.

5.4 Page Object pattern

Page Object is a design pattern that is very popular in test automation thanks to the possibility of reducing the code duplication and to enhance the test maintenance. A Page Object is an independent object-oriented class that serves as an interface to a page of the test. In this context, the test uses the methods of the interface whenever they need to interact with the UI of that page. Since the UI changes are located just in one place, commonly in the page, the tests don't need to be changed [28]. When Page Object pattern is used, all Page Object can be encapsulated into functional classes that will contain all

elements and functions which the Page Object is providing. Selenium handles the locating of the elements what allows to perform the actions with the found elements. A web element is an interface that provides all actions that the user can do with elements. This interface is provided by Selenium being the most useful one the function click. Web element is a very general unit and it can represent any element of DOM (Document Object Model): Selenium also provides other interfaces for special HTML elements, like inputs, selects, etc. It's important to know that the role of Selenium WebDriver in functional testing is to perform the user interaction with the tested application. As a summary, these are the best practices for working with Page Objects [42]:

- The HTML structure of the page is defined only in one place; so the change in the HTML code will affect the page class of the test.
- Page Objects expose services to developers, in other words, provide methods to interact with the page.
- These methods should return again other Page Objects. This encourages the test developers to interact rather with the services than with the implementation. As a result, they are able to control which tests will fail, it means better maintainability.
- Page Objects do not have to represent the whole website, they can be just part of it.

5.5 Test execution

In order to judge the effectiveness and efficiency of the current testing practices, they need to be quantified and measured. To do this, a relevant set of metrics should be collected during the test execution. These metrics are going to be used as an indication of the current quality of the software product. The table 5.1 contains the selected metrics that are measured during the test execution.
There are some other metrics to take into account, but those metrics aren't measured in the execution time, so they will be shown during the analysis of the results. The browser used to do the test execution was Mozilla Firefox in order to be able to compare the IDE results with the WebDriver results. The automation for the three applications were performed by using three different tools: Selenium IDE, Selenium Java WebDriver and Selenium C# WebDriver.

The characteristics of the computer were all tests were performed are shown below:

Feature	**Description**
Operating System:	Windows 7 Home Premium. Service Pack 1.
System type:	64-bit Operating System
Processor:	Intel Core i7-2670QM 2.20GHz
Memory RAM:	6 GB

Concept	Description	Motivation
Execution time	Time that the tool takes to execute a test, this is important because there are different tools and each tool can have different requirements to execute a test.	Since large test suites can take significant time to execute, each test case should be measured in terms of time. As mentioned by Hayes this can also be useful in order to find performance problems [43].
Number of errors	Number of errors found during the execution.	Testing is about find errors and correct them and generally, the quality of a project is measure having into account the number of errors.
Successful execution	The fact that the execution has finished completely and all the test cases are in state "Passed".	This is important because sometimes the automated tests are not completely stables, it means, that even the execution finished successfully, it can be failed during another execution.

Table 5.1: Measures to take during test execution.

5.6 Analysis of the results

Three Test Suites were created during this thesis. Each Test Suite contained its own script and each script is equivalent to a tab of the application, each tab is composed by a series of test cases, which are shown in table 5.2 and the name of each test is indicated in Appendix A.5. Following the directives of the proposed methodology for the automation of the tests, some metrics are considered as criteria for the selection of the best automation tool.

5.6.1 Environment installation and configuration complexity

Selenium IDE is quite easy to install and configure because it works as an extension of Mozilla Firefox and in case to need an extension, it can be installed in the same way as the IDE. After that, the IDE is configured to start the automation. Selenium Java WebDriver work just installing an IDE that enables to codify in Java. For the tests, Eclipse and JUnit were used. The first one only needs to be downloaded without any additional configuration in its IDE while the second one is configured using Maven, in order to facilites the addition of the diferent required dependencies. Selenium C# WebDriver works just with installing the Visual Studio IDE. It takes some time to install it. In this case, the references as added as Nunit. After that, C# is configured to start the automation. As noticed, the

Test Suite	Test Script	Test Case
Small Application	AddTask	1
Medium Application	Login	1
	Overview	2
	NewTicket	2
	SearchTicket	1
	TimeTracking	3
Large Application	Login	1
	Security	11
	Workflow	3
	Metadata	7
	Import	1
	Browse & Search	23
	Dashboard	35

Table 5.2: Test Cases by script.

installation and configuration of the enviroments do not represent a hard task for the tester, specially Selenium IDE, since it works as a Firefox extension and takes less than five minutes to be ready, just having a Firefox browser in your machine, so you don't need to install other different tools.

5.6.2 Learning curve

When it comes to choose a tool to perfom the automation of the tests, some key factors must be taken into account even more when the automators may not have a deeper knowledge in programming languages or in some cases, they have the basis knowledge of object-oriented programming. One of these key factors is the learning curve of the tool, which in the projects means a saving of time and resources during the implementation. To estimante the learning curve, a survey was conducted to investigate (i) what is the most common language used by automation testers when it comes to automating the tests and (ii) what is their perception during the learning process. In this section, the analyisis of the survey results provide the first key factor for the selection of the automation tool. The learning curve of Selenium IDE was not taken into account for the survey, because it's not a programming language and even it's easy to learn.

Question 1. Which of the following programming languages do you know?

The 32% of the respondents knew Java, the 28% knew C# and the 40% answered that they knew both languages (see figure 5.8). This trend is similar to the one presented by the IEEE in its top programming languages review [44], according to IEEE, Java ist more popular than C# and maybe this is because the former one is an open-source multiplatform, while the latter requieres the IDE of Visual Studio, which works mainly on Windows operating systems. For other operating systems as Linux, other IDEs as MonoDevelop could be used.

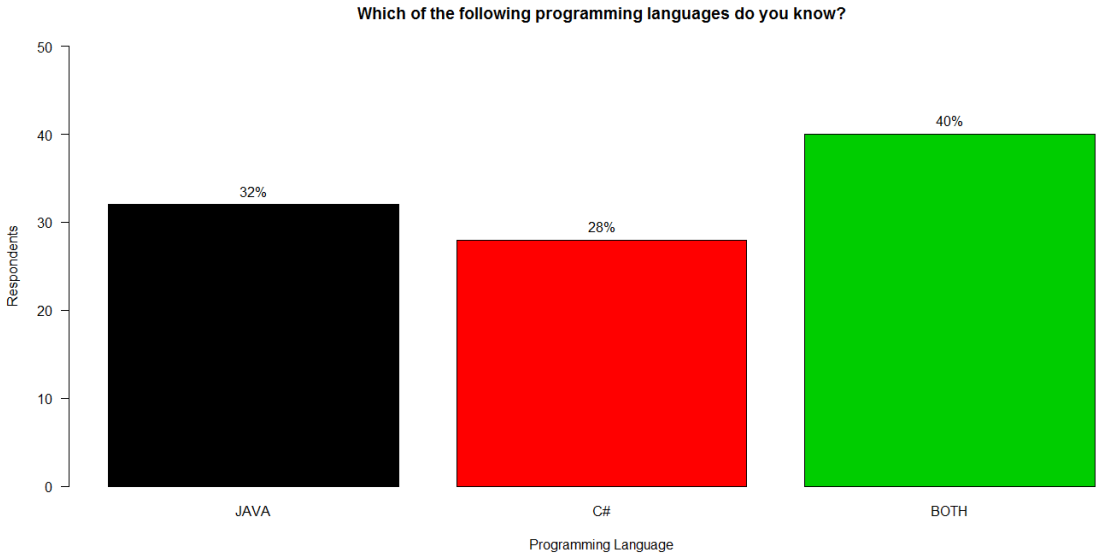

Figure 5.8: Answers of respondents to Question 1.

Question 2. What is your cumulative programming experience in this (these) language(s)?

36% of the respondents answered that they have less than six months of expertise in the programming language, the 34% have at least one year of expertise and the last 30% have more than two years of knowledge of the programming language (see figure 5.9).

Question 3. How long have you been using an IDE for C# or Java programming?

More than half of the respondents (56%) has less than six months working on an IDE. The answers of the remaining 44% of the responders are equally divided: 22% of the responders works on an IDE since one year while the other works since two years as showed in figure 5.10.

Question 4. What framework do you use to create your automated test scripts? You can choose more than one option.

According to the survey, the programming language more used to automation testing with 29% is C#. Java is used by 23% of the asked people while Selenium IDE is used by 21%. The other 27% percent of respondents are used another programming language to create their scripts (see figure 5.11). To be noticed, this question revealed that although Java is better known, automators use more C# for the creation of scripts.

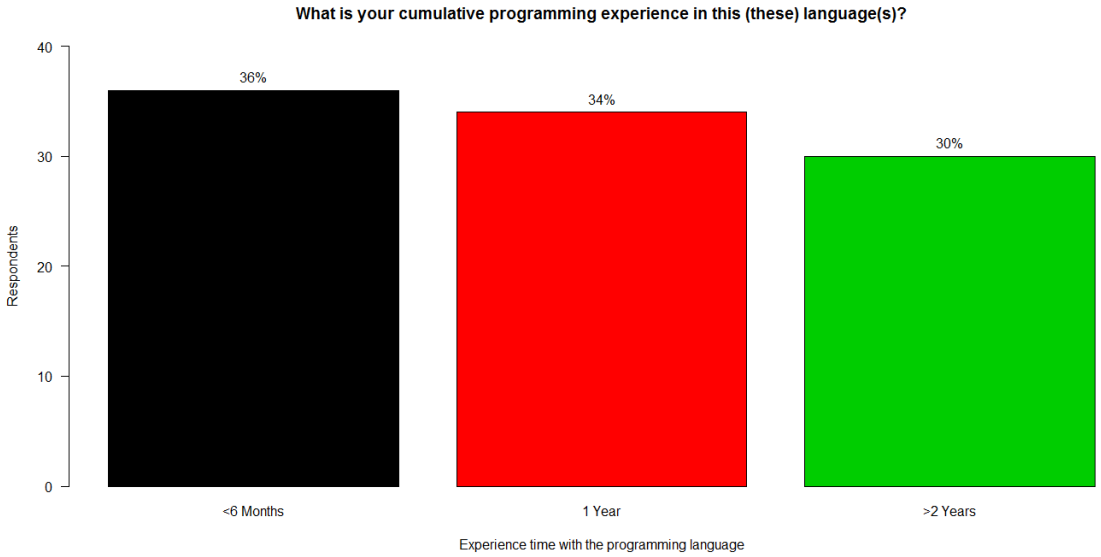

Figure 5.9: Answers of respondents to Question 2.

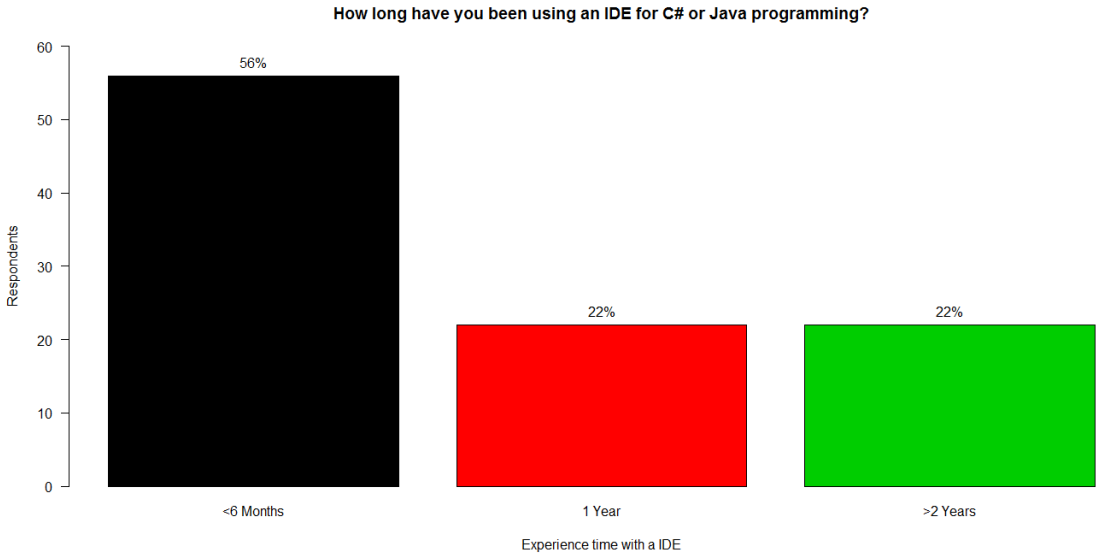

Figure 5.10: Answers of respondents to Question 3.

Question 5. What programming language do you think that you learn faster?

48% of the respondents answered that C# is easier to learn than Java. 40% answered that Java is easier to learn than C#. And just the 12% agreed that both have the same

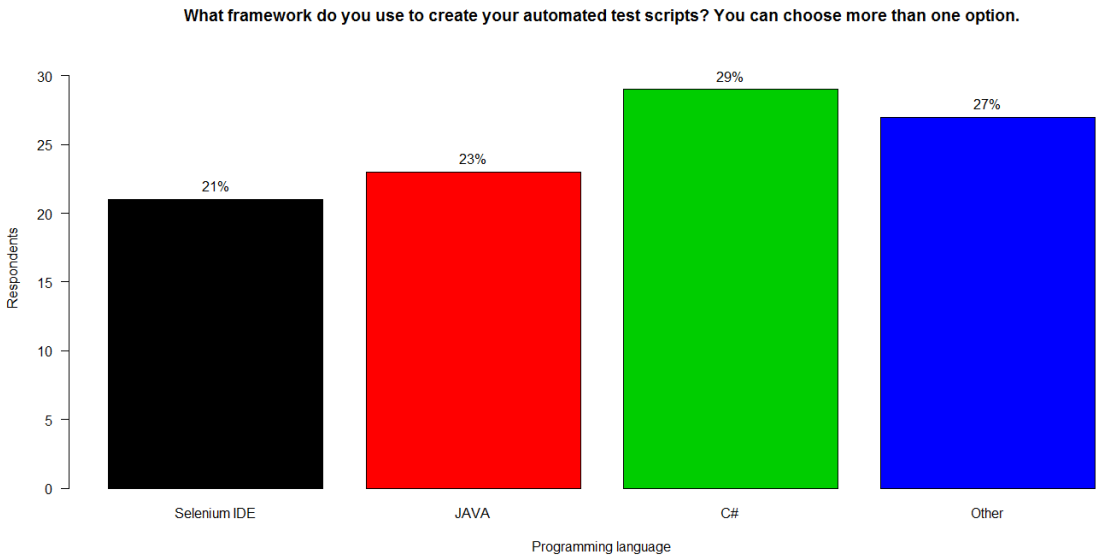

Figure 5.11: Answers of respondents to Question 4.

complexity level (see figure 5.12).

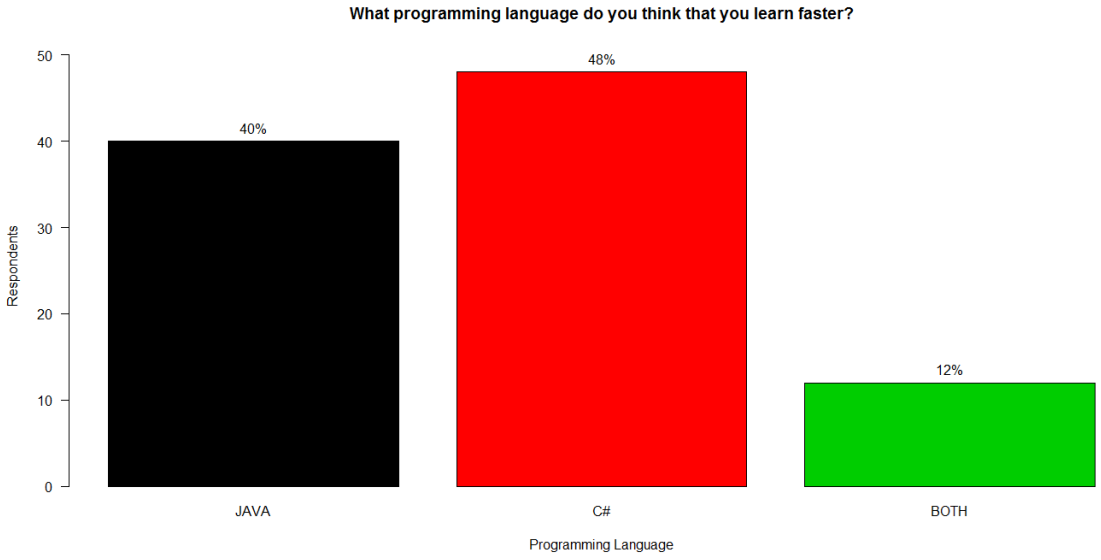

Figure 5.12: Answers of respondents to Question 5.

Although Java seems to be the most popular language within the QAs, the majority of them use C# (Question 4) motivated in most of the cases, in one hand, by the easier

high-level syntax and in the other hand by the fact that a QA is not used to programming (Question 5).

5.6.3 Creation time of the test scenarios

A second key factor is the creation time of the test scenarios. After create all them for each application, the following can be concluded for this factor: Time taken to create the test scripts on Selenium Java and C# are bigger than Selenium IDE for all cases. This is not a surprise since in Selenium IDE only the script need to be created and executed. For Java and C#, Page Object pattern was used and additional classes were created.
In figure 5.13 the creation time of the test cases $TS1_s$ to $TS2_s$ for the small application is showed. As a remark $TS1_s$ was measured without taking into account the required time for the tests management (creation of the Test Suite, drivers and additional required methods for the performance of the tests). As a consequence, an additional Test Scenario was included named $TS2_s$, for measuring exclusively the spent time in the tests management. In this context, $TS2_s$ is not a test, it allows to compare Selenium IDE with Java and C# in terms of the creation time of the required Tests Suite, drivers, and additional methods. As can be observed in figure 5.13, the creation time assigned to Selenium IDE is equal to zero (the IDE manage automatically the execution of the scripts), while the times for Java and C# are different to zero and similars.
Some conclusions can be taken out from the creation time: The time spent for the creation of the tests scripts for the small application using Selenium IDE is equivalent to one working day while using Java and C# took 2 and 2.1 working days, respectively. A working day is here understood as a eight hours period. Regarding the tests management, although it can be handled in a single file, it's not recommendable, since what is sought is to have a maintainable code, therefore, the classes must be independent in order to better manage the test. The above tell us that the final times depends strongly on the management stage even more than on the time spent in the scripts.
In figure 5.14, the creation time of the test cases $TS1_m$ to $TS6_m$ for the medium application are shown. It can be observed that the difference between Java and C# is still very small. For $TS1_m$ and $TS4_m$, the creation time of the scripts are very similar for the three tools due to the simplicity of the test to be performed. It means complex structures within the code are not required. Regarding $TS2_m$, the test to be performed is more complex, a cycle with several nested ifs had to be created and all the functionality of all possible paths must be verified. These actions are hard to control in Selenium IDE, hence the creation times in this tool were bigger than in Java and C#, with which the same actions can be handled easily. As a remark, $TS1_m$ to $TS5_m$ were measured without taking into account the required time for the tests management (creation of the Test Suite, drivers and additional required methods for the performance of the tests). As a consequence, an additional Test Scenario was included called $TS6_m$, for measuring exclusively the spent time in the tests management. In this context, $TS6_m$ in the medium application is equivalent to $TS2_s$ in the small application. In addition, as some tests have a higher complexity, it's observed that the creation time of the scripts using Selenium IDE is not longer the half of the spent times in Java and C#. However, the creation time of the Selenium IDE (9.1 working days) scripts is still smaller than Java and C# (11.2 and 11.3 working days

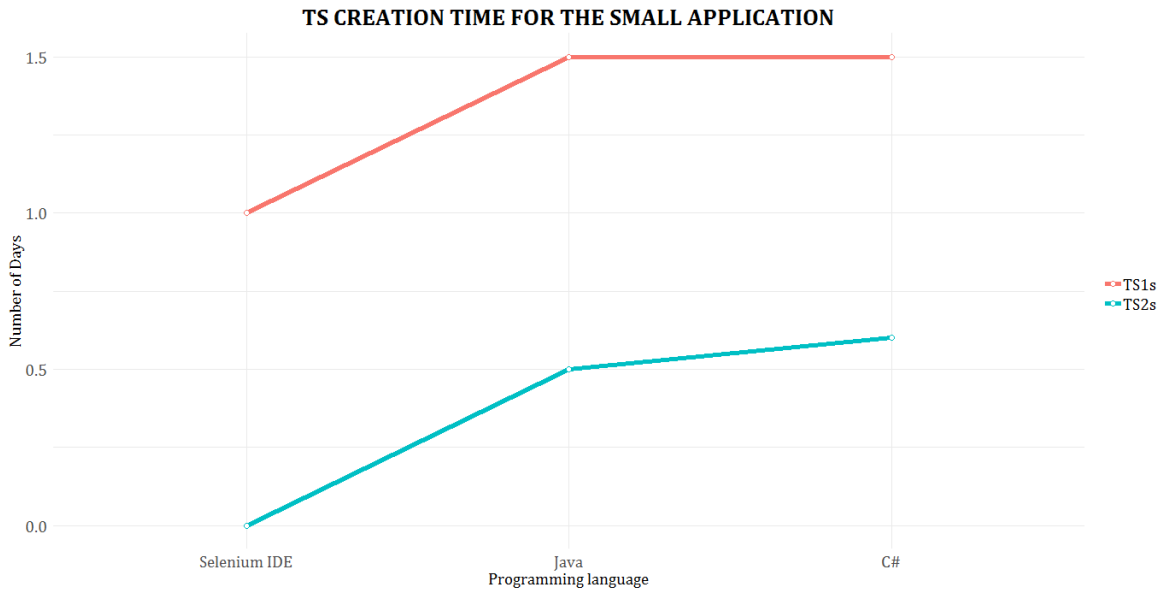

Figure 5.13: Comparison of the creation time for the small application.

respectively). To highlight, the change in the code is a hard task in Selenium IDE since it is difficult to find the line, even by the person who created the script, neverthless in Java and C#, thanks to Page Object pattern, the script can be changed easily. This matter is going to be analyzed deeply in another section.

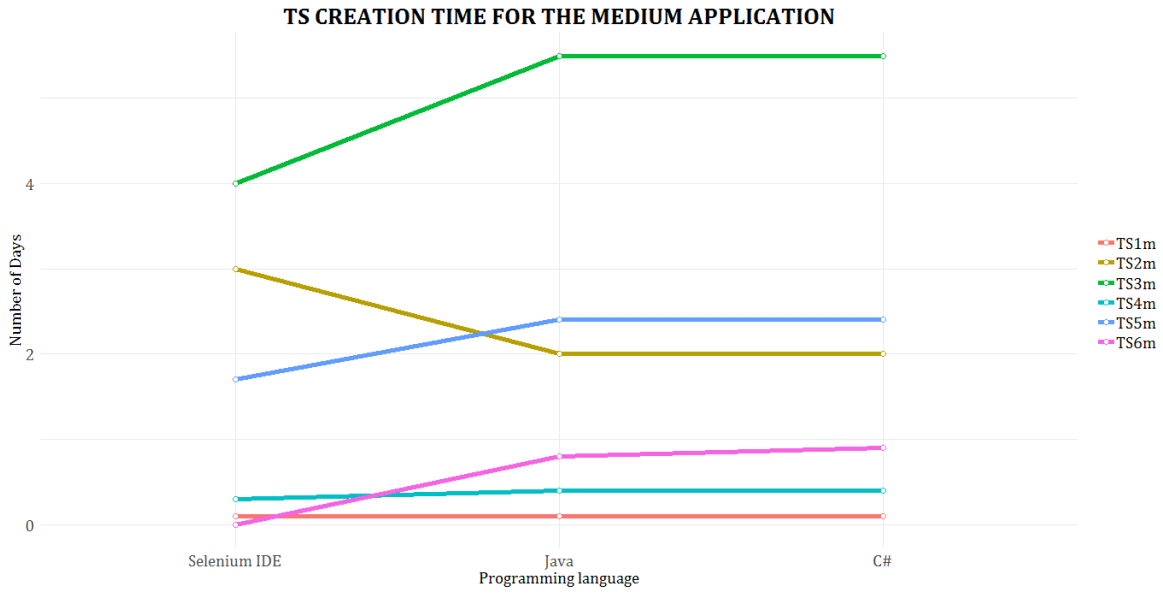

Figure 5.14: Comparison of the creation time for the medium application.

In figure 5.15, the creation time of the test cases $TS1_l$ to $TS8_l$ for the large application are shown. It can be observed that the difference between Java and C# is still very small. For $TS1_l$ and $TS5_l$, the creation time of the scripts are very similar for the three tools due to the simplicity of the test to be performed. It means complex structures within the code are not required. As a remark, $TS1_l$ to $TS7_l$ were measured without taking into account the required time for the tests management (creation of the Test Suite, drivers and additional required methods for the performance of the tests). As a consequence, an additional Test Scenario was included called $TS8_l$, for measuring exclusively the spent time in the tests management. In this context, $TS8_l$ in the large application is equivalent to $TS2_s$ in the small application. In addition, as some tests have a higher complexity, it's observed that the creation time of the scripts using Selenium IDE is not longer the half of the spent times in Java and C#. However the creation time of the Selenium IDE (24.8 working days) scripts is still smaller than Java and C# (42.2 and 42.3 working days respectively).

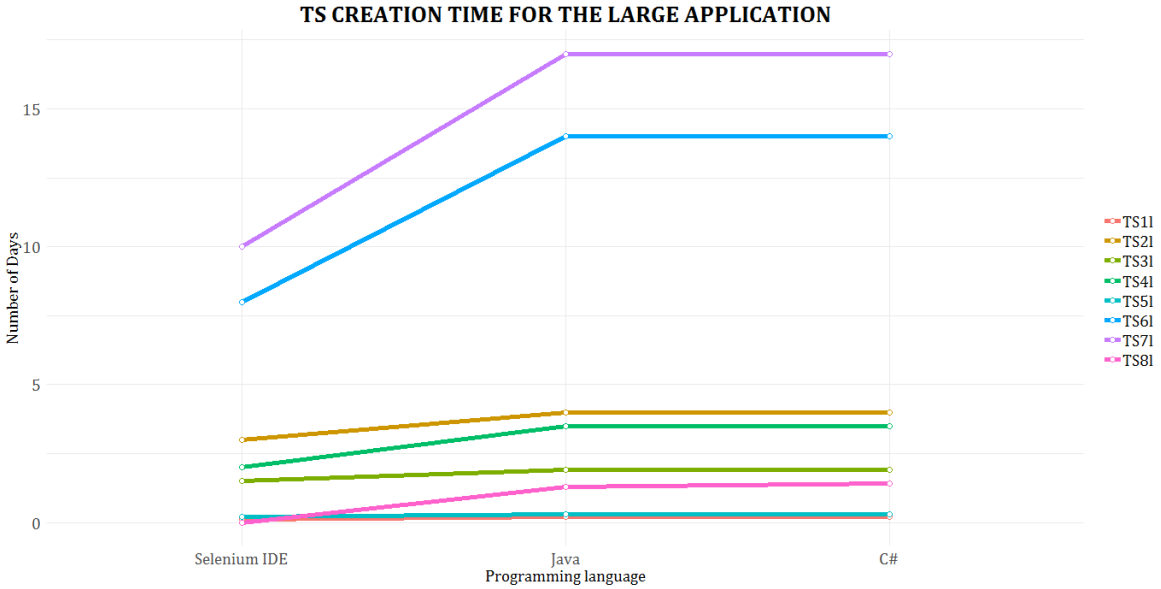

Figure 5.15: Comparison of the creation time for the large application.

5.6.4 Lines of code

The third key factor is relating with the total number of code lines written for the creation of the scripts. Since Page Object pattern is used, for each test script created in Selenium WebDriver a total of two classes for each language were generated, one for executing the test and another for creating the page objects to reach the web page elements and that's the main reason for the results. Additionaly, other classes were created for managing the driver and the execution of the test scenarios. In this context, it is expected, as it can be observed in the following figures (5.16, 5.17, and 5.18) that the lines number of the

scripts codified in Java and C# is always bigger than the number of lines in Selenium IDE. Comparing the lines of code of Java and C#, the number of lines is similar within the three applications, nevertheless the lines number of the scripts in C# is smaller than Java for the small and large application. The difference is however negligible. In figure 5.16, the lines number of the test scripts created for the small application using the three tools are exposed. Regarding $TS1_s$, the corresponding number of lines for Selenium IDE is almost 2.5 smaller in comparison to Java and C#. Here we can point an important observation out: Although the difference in the lines number in Selenium IDE is significant with respect to Java and C#, the same difference (2.5 times) was not observed in the creation time of the scripts, this fact can be explained as follows; in one hand, for creating a complex code structure in Selenium IDE such as if/else and loops, the complexity level is higher, on the other hand, the programming languages, use brackets, indenters increasing the number of code lines. Selenium IDE does not allow to leave white spaces within commands, which makes the code hard to understand and to require an addition time for analysing it. Moreover, the use of Page Object pattern increases two times the number of script lines for Java and C# with respect of Selenium IDE scripts. In this context, $TS2_s$ is not a test, the scripts represents the code lines codified for the test management (creation of the Test Suite, drivers and additional required methods for the performance of the tests). Since Selenium IDE does it automatically, the number of lines is equal to zero, while the number of lines for Java and C# are different to zero and similars. However, these times are negligible.

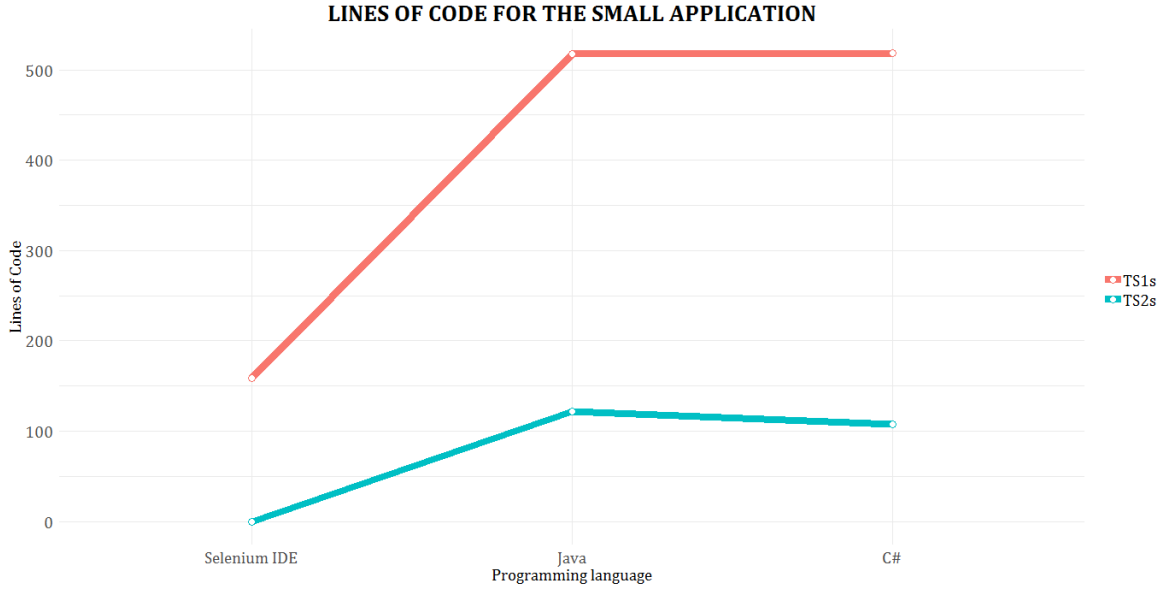

Figure 5.16: Comparison of the lines of code for the small application.

In figure 5.17, the lines number of the test scripts created for the medium application using the three tools are exposed. Regarding $TS4_m$, the corresponding number of lines for Selenium IDE is almost 8 times smaller in comparison to Java and C#. Here we

can point an important observation out: Although the difference in the lines number in Selenium IDE is significant with respect to Java and C#, the same difference (8 times) was not observed in the creation time of the scripts, this fact can be explained as follows; in one hand, for creating a complex code structure in Selenium IDE such as if/else and loops, the complexity level is higher, on the other hand, the programming languages, use brackets, indenters increasing the number of code lines. Selenium IDE does not allow to leave white spaces within commands, which makes the code hard to understand and to require an addition time for analysing it. Moreover, the use of Page Object pattern increases two times the number of script lines for Java and C# with respect of Selenium IDE scripts. In this context, $TS6_m$ is not a test, the scripts represents the code lines codified for the test management. $TS6_m$ in the medium application is equivalent to $TS2_s$ in the small application. Since Selenium IDE does it automatically, the number of lines is equal to zero, while the number of lines for Java and C# are different to zero and similars. However, these times are negligible.

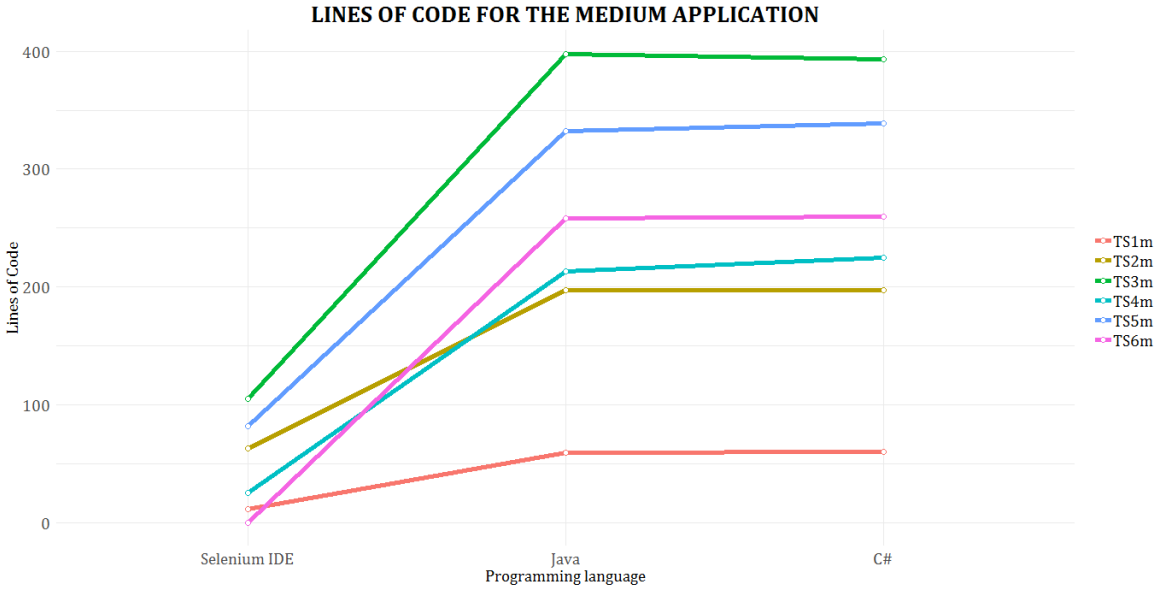

Figure 5.17: Comparison of the lines of code for the medium application.

In figure 5.18, the lines number of the test scripts created for the large application using the three tools are exposed. To be noticed, the difference between the large and medium applications is not so evident, this fact also can be easily explained as follows: Java and C# are programming languages and Page Object pattern was used, therefore it was possible reuse the common methods invoking them in the respective object class, meaning that a method can be called several times while it implemation was made only once. However, in Selenium IDE, it was not possible, so the commands had to be written again. In this context, $TS8_l$ is not a test, the scripts represents the code lines codified for the test management. $TS8_l$ in the large application is equivalent to $TS2_s$ in the small application. Since Selenium IDE does it automatically, the number of lines is equal to zero, while the

number of lines for Java and C# are different to zero and similars. However, these times are negligible.

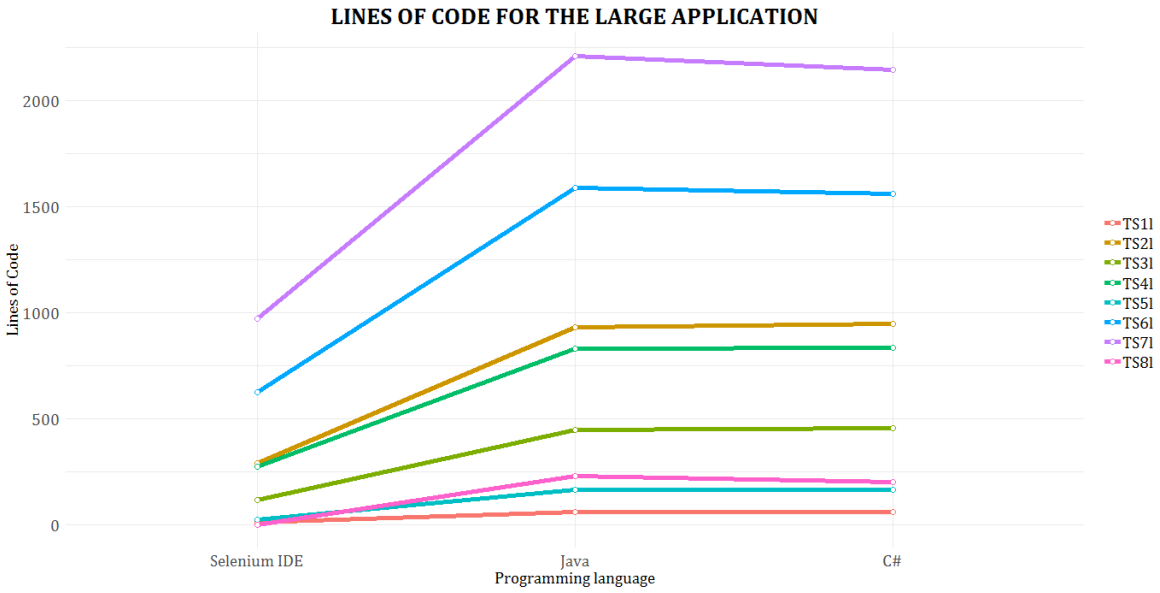

Figure 5.18: Comparison of the lines of code for the large application.

5.6.5 Execution time

Another key factor is the execution time. It was measured in such a way that can truly represent the time to complete the entire test suite. So the following considerations were taken into account: The test suites for each tool were run five times and all the execution time were taken right after to had turned the computer on to guarantee that the application to be tested was the only programm demanding resources. In the following figures (5.19, 5.20, and 5.21), it can be observed that the there is a remarkable difference between the tools. For Selenium IDE, the execution of the test cases always demand more time. Meanwhile, for Java and C#, the test cases programmed using the second language spent less time during their execution. For the medium and large application, the trend is the same; the execution times are shorter in C# than in Java. Also can be concluded that as the application grows, the gap in the execution time will increase.
In figure 5.19, the measured times (seconds) for each of the five executions for the small application are shown. In average, the execution time of the test suites in Selenium IDE, Java and C# are 32.46, 9.56, and 9.31 seconds, respectively. The difference between the automation tools in this case is negligible.
The figure 5.20 shows the results for the medium application. In average, the execution time of the test suites in Selenium IDE, Java and C# are 105.88, 76.81, and 71.14 seconds, respectively. Althoug Selenium IDE is still slower than the other two tools, in this application a difference between C# and Java could be noted. C# has a shorter average

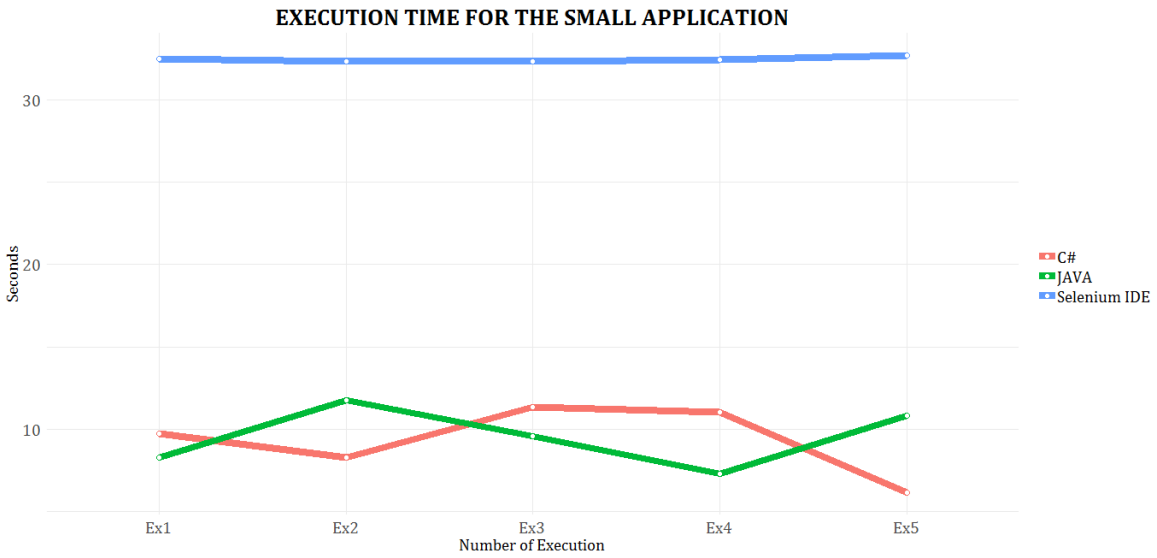

Figure 5.19: Comparison of the lines of code for the small application.

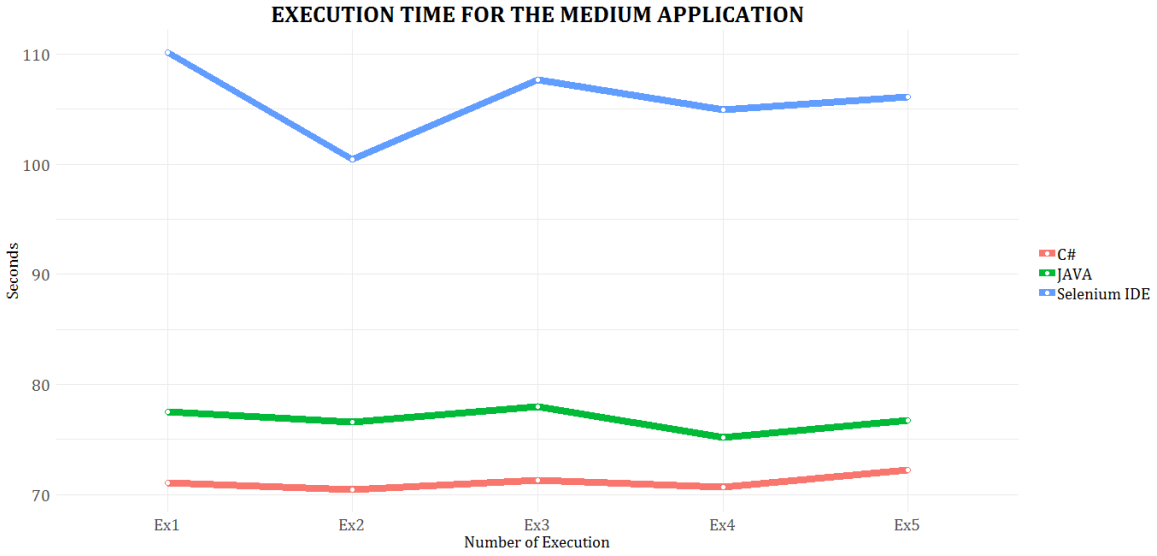

Figure 5.20: Comparison of the lines of code for the medium application.

execution time.
Finally, In figure 5.21, the execution times for each execution of the test suite of the large application are shown. In average, the execution time of the test suites in Selenium IDE, Java and C are 1656.06, 1356.46, and 1231.05 seconds, respectively. Once again C# seems to be faster than the other tools and suitable for large projects. The registered difference increased to 125 seconds (a little more than two minutes).

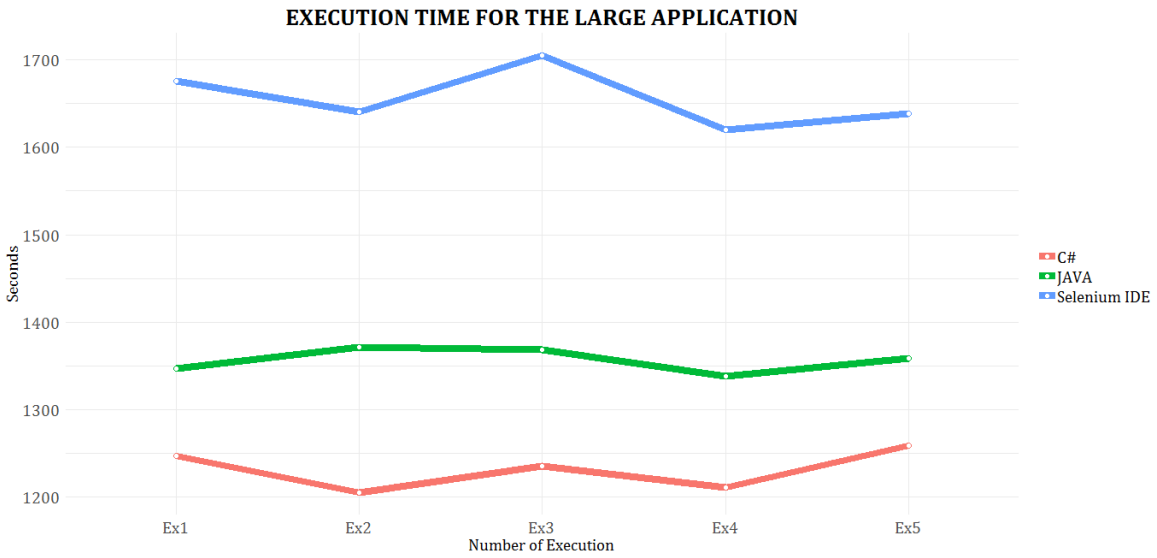

Figure 5.21: Comparison of the lines of code for the large application.

5.6.6 Number of errors

A further key factor is the number of errors. Since the same logic and methodology were followed for automating the tests in the three tools, the same number of errors were found for each of the automation tools used in each of the applications. For a better understanding refer to the table 5.3.

	Selenium IDE	**Selenium Java**	**Selenium C#**
Small application	0	0	0
Medium application	1	1	1
Large application	5	5	5

Table 5.3: Number of errors for tool/application.

5.6.7 Successful execution

The successful execution of test is also considered as a key factor. Sometimes, the execution can fail, because the locators are very weak, it means that with the same locator it's not able to find the element in different executions, so it is important to be sure that a strong locator was created and it's advisable always try to find the elements using the locators in the order presented below [28]:

1. Id
2. ClassName

3. LinkText
4. PartialLinkText
5. CssSelector
6. Xpath

In the execution, it was found that the main error was related to the not found locators. The rules were followed as described above and it was found that if the locator failed, it failed in all the tools and a new locator must be found. In general, the three tools showed an "Element locator not found" through a message or an exception. After solving these problems, the applications were executing successfully in the three tools.

5.6.8 Code maintainability

The last key factor is related with the code maintainability defined as the effort required to do a change in the scripts. How it was explained in the Line of Codes and Creation time of the test scenarios subsections, for Java and C# was used the Page Object pattern. From the Selenium documentation [28], it's known that the patterns help to enhance the test maintenance, because the UI changes are just located in one place, doing the maintainability of the code very fast and inexpensive. As it was mentioned in the last sections, java and C# use Page Object pattern while in Selenium IDE it's not possible to use it, therefore the scripts were created using the Selenium IDE editor, which makes a change in the UI difficult to find and perform (more expensive). On the other hand, there are situations in which a method can be used several times during the execution of the test, that is, the functionality is encapsulated in a single method, and when there is a change that affects this method, the change would only have to be done once, whereas in Selenium IDE, the change would have to be made every time over the lines of the scripts that are going to use it, as it happened in the implementation of the large application, where was possible to reuse several methods, it means, the larger the application, the more possibilities that part of the code can be reused.

5.7 Selection of the automation tool

The different key factors included in the last sections will enable to select the suitable tool for automating the tests of a project with specific requirements. In this section, the testers can find a suggestion of the best tool for each one according to the relevance of each factor in the projects. First some limitations and advantages of each tool are giving, in order to reflect the experience of managing each one. Selenium IDE, for instance, is easy to install and the automation scripts are easily created, even extra extensions can be added for having robust scripts like with the programming languages. Nevertheless it has some limitations when it comes to code maintainability and execution times. This last aspect is essential when choosing Selenium IDE for small applications without and expected growth in the future, otherwise, there could be a re-process in the migration process. Regarding to Java anc C#, they are very similar programming languages, both allow to use patterns

or other libraries to have an organized code, easy to execute and maintain. However, from the proposed key factors, there are some differences that can help to a QA to decide what language to choose. Mainly, it was observed that based on the testers experience, C# seems to have an easier syntax and based on the recorded execution times, C# is able to perform the execution in less time even when the application becomes larger. In table 5.4, it's presented in detail what tool should be used in relation with the size of the application (creation time, line of codes, execution time, and code maintainability) and the learning curve of the tester. From the table the following remarks can be exposed:

- If the tester or project fullfills the following requirements: No basic knowledge of object-oriented programming, it's a small application (small in the context of this thesis), the tester should use Selenium IDE, in this way the tester will learn faster about test automation and to identify the elements of a web page. There is however one risk, if the project grows, automation must be migrated to a programming language. On the contrary, if the tester has the basic knowledge of object-oriented programming, the most advisable thing is to use a language such as Java or C#, in this way the tester can take advantage of his knowledge and make an automation that can be very useful towards the future, a more maintainable and easier to understand for an external person to the project, and although the creation time of the scripts is greater, the real benefits will be seen when making a change within the code or the application grow.

- If the tester or project fullfills the following requirements: No basic knowledge of object-oriented programming, it's a medium application (medium in the context of this thesis), the tester should use a language like C#, since it's perceived to be much easier to learn and the times of execution are a bit smaller, on the contrary, if he possesses the basic knowledge of object-oriented programming, it would be better to perform the automation in the language that the person feels most comfortable with, or if there is some automation previously done in one of these two languages, the ideal would be to use that language, since the code can be reused, for example for the creation of the driver and the structure of the Test Suite management.

- If the tester or project fullfills the following requirements: No basic knowledge of object-oriented programming, it's a medium application (medium in the context of this thesis), the tester should use a language like C#, since it's perceived to be much easier to learn and execution times are somewhat smaller, and in comparison with Java, in C# less code line could be needed. On the contrary, if he possesses the basic knowledge of object-oriented programming, it would be better to perform the automation in the language that he feels most comfortable, or if there is some automation previously done in one of these two languages, the ideal would be to use that language, since the code can be reused, for example, for the creation of the driver and the structure of the Test Suite management.

	Selenium IDE Knowledge	Java Knowledge	C# Knowledge	Language to use
Small Application	✓	✓	✓	C#
	✓	✓	✗	Java
	✓	✗	✓	C#
	✗	✓	✓	C#
	✓	✗	✗	Selenium IDE
	✗	✓	✗	Java
	✗	✗	✓	C#
	✗	✗	✗	Selenium IDE
Medium Application	✓	✓	✓	C#
	✓	✓	✗	Java
	✓	✗	✓	C#
	✗	✓	✓	C#
	✓	✗	✗	C#
	✗	✓	✗	Java
	✗	✗	✓	C#
	✗	✗	✗	C#
Large Application	✓	✓	✓	C#
	✓	✓	✗	Java
	✓	✗	✓	C#
	✗	✓	✓	C#
	✓	✗	✗	C#
	✗	✓	✗	Java
	✗	✗	✓	C#
	✗	✗	✗	C#

Table 5.4: Automation tool to use according to the application size and the programming language knowledge.

Chapter 6

Conclusions and Future Work

The most relevant conclusions obtained in this thesis regarding the research carried out, the methodology used to solve the problem and the results obtained are listed below.

- The proposed methodology allowed to classify, according to the testing level and in a general way, the applications subject to testing in three types of applications: Small applications where unit testing is carried out, medium applications where integration testing is carried out, and large applications where system testing (regression testing) is carried out.

- Regarding the automation tool, the majority of people knows Java, but C# is more used and is perceived as easier to learn. Through a survey, an experimental learning curve for the two programming languages was built. The survey reveled that although Java is the most popular language, the tester preferred to uses C# in their projects, motivated mainly for the fact, that C# has an syntax and structure easier to be learnt and understood.

- Contrary to what many people think, Selenium IDE (plus other extensions) can be used to create complex scripts, using advanced programming structures such as if/else or loops. As a drawback, using Selenium IDE can represent a higher complexity when it comes to the creation, understanding and management of the scripts, even in small applications.

- Regarding the creation time it was observed for the three case studies chosen for automation revealed important differences in two of the three tools. It was observed that the spent time to create the test scripts on Selenium Java and C# is bigger than the spent one in Selenium IDE in all the case studies. This makes sense, because in Selenium IDE, the script is only created and executed, while in Java and C#, the Page Object pattern was used, increasing the time due to its developing. In Selenium Java and Selenium C# the creation time of the scripts is practically the same, being the difference negligible.

- Regarding the lines of code it was observed for the three case studies chosen for automation revealed important differences in two of the three tools. It was observed

that the number of code lines used to create the test scripts are bigger in Java and C# in comparison with Selenium IDE. As in the creation time of the scripts, the Page Object pattern also affects the lines of code that should be written in the tests, because for each test script created in Selenium IDE, we have two classes created in the other two languages, one for executing the test and the other for creating the objects to reach the web page elements.

- Regarding the execution time it was observed for the three case studies chosen for automation revealed important differences in the three tools. Initially it was observed how the execution time is greater for the implementation made with Selenium IDE, however, as the size of the application increased, it was also observed that among the other two languages, there is also a difference in the time of execution, being in all cases greater the time for Selenium IDE. The execution with Java showed that it was faster with respect to Selenium IDE, and finally, the execution time with C# was the fastest. The execution time plays an important role as a selection factor specially when the application size grows.

The following points describe some proposals for future work:

- Conduct the comparison of test automation tools for applications on mobile devices; tools based on Selenium web drivers such as AndroidDriver, iOSDriver, Selendroid or Appium.
- Make the comparison using more programming languages supported by Selenium web driver, such as Ruby or Python.
- Include other tools than Selenium web driver, to try to make a more complete comparison between the main tools of test automation.

Appendix A

Automation Scripts

For the automation of the test scenarios, three different tools were used in each of the three case studies: Small, medium and large applications. In this section, one script of the medium application is shown in order to recognize the differences of each automation tool. Inside the script named *NewTicket*, there are two test cases that allow to create a ticket and verify that the ticket was created correctly.

A.1 Selenium IDE

In Selenium IDE, we just have one script and it's shown below.

#	Command	Target	Value
1	storeEval	new Date().getTime()	startTimeNewTicket
2	clickAndWait	link=New ticket	
3	storeEval	['Bug', 'Task', 'Fix']	tracker
4	storeEval	new Array();	priorityArray
5	storeEval	new Array();	versionArray
6	storeEval	new Array();	typeArray
7	storeEval	new Array();	ticketTypeArray
8	storeEval	storedVars['tracker'].length	sizeTracker
9	store	javascript{Math.floor(Math.random()*storedVars['sizeTracker']);}	randomTracker
10	storeEval	storedVars['tracker'][storedVars['randomTracker']]	trackerLabel
11	select	//*[@id='issue_tracker_id']	${trackerLabel}
12	storeEval	new Date().getTime()	dateTime
13	storeEval	"Subject field test Automation-".concat(storedVars['dateTime'])	subject
14	storeEval	"Description field test Automation-".concat(storedVars['dateTime'])	description
15	storeXpathCount	//*[@id='issue_priority_id']/option	numOptionsPriority
16	store	1	i
17	while	storedVars['i'] <= storedVars['numOptionsPriority']	
18	storeValue	//*[@id='issue_priority_id']/option[${i}]	priorityAux
19	storeEval	storedVars['priorityArray'].push(storedVars['priorityAux'])	
20	storeEval	${i} + 1	i
21	endWhile		
22	store	javascript{Math.floor(Math.random()*storedVars['numOptionsPriority']);}	randomPriority
23	storeEval	storedVars['priorityArray'][storedVars['randomPriority']]	priority
24	storeXpathCount	//*[@id='issue_fixed_version_id']/option	numOptionsVersion
25	store	1	i
26	while	storedVars['i'] <= storedVars['numOptionsVersion']	
27	storeValue	//*[@id='issue_fixed_version_id']/option[${i}]	versionAux
28	storeEval	storedVars['versionArray'].push(storedVars['versionAux'])	
29	storeEval	${i} + 1	i
30	endWhile		
31	store	javascript{Math.floor(Math.random()*storedVars['numOptionsVersion']);}	randomVersion
32	storeEval	storedVars['versionArray'][storedVars['randomVersion']]	version
33	gotoIf	storedVars['trackerLabel'] == "Task"	endIfType
34	storeXpathCount	//*[@id='issue_priority_id']/option	numOptionsType
35	store	1	i
36	while	storedVars['i'] <= storedVars['numOptionsType']	
37	storeValue	//*[@id='issue_custom_field_values_1']/option[${i}]	typeAux

38	storeEval	storedVars['typeArray'].push(storedVars['typeAux'])	
39	storeEval	${i} + 1	i
40	endWhile		
41	store	javascript{Math.floor(Math.random()*storedVars['numOptionsType']);}	randomType
42	storeEval	storedVars['typeArray'][storedVars['randomType']]	type
43	label	endIfType	
44	gotoIf	storedVars['trackerLabel'] == "Bug" || storedVars['trackerLabel'] == "Task"	endIfTicketType
45	storeXpathCount	//*[@id='issue_custom_field_values_8']/option	numOptionsTicketType
46	store	1	i
47	while	[c]@1@storedVars['i'] <= storedVars['numOptionsTicketType']	
48	storeValue	//*[@id='issue_custom_field_values_8']/option[${i}]	ticketTypeAux
49	gotoIf	storedVars['ticketTypeAux'] == "" || storedVars['ticketTypeAux'] == "null"	endIfTicketTypeAux
50	storeEval	storedVars['ticketTypeArray'].push(storedVars['ticketTypeAux'])	
51	label	endIfTicketTypeAux	
52	storeEval	${i} + 1	i
53	endWhile		
54	storeEval	${numOptionsTicketType} - 1	numOptionsTicketType
55	store	javascript{Math.floor(Math.random()*storedVars['numOptionsTicketType']);}	randomTicketType
56	storeEval	storedVars['ticketTypeArray'][storedVars['randomTicketType']]	ticketType
57	label	endIfTicketType	
58	type	id=issue_subject	${subject}
59	type	id=issue_description	${description}
60	select	id=issue_status_id	index=0
61	select	id=issue_priority_id	value=${priority}
62	storeSelectedLabel	id=issue_priority_id	priority
63	select	id=issue_assigned_to_id	${currentUser}
64	select	id=issue_fixed_version_id	value=${version}
65	storeSelectedLabel	id=issue_fixed_version_id	version
66	gotoIf	storedVars['trackerLabel'] == "Task"	endIfSelectType
67	select	id=issue_custom_field_values_1	${type}
68	label	endIfSelectType	
69	storeValue	//*[@id='issue_start_date']	startDate
70	storeEval	d = new Date(storedVars['startDate']); d.setMonth(d.getMonth() + 1); d = d.getFullYear() + '-' + ('0' + (d.getMonth() + 1)).slice(-2) + '-' + ('0' + d.getDate()).slice(-2);	dueDate
71	type	//*[@id='issue_due_date']	${dueDate}
72	store	javascript{Math.floor(Math.random() * 9) + 1;}	randomHours
73	type	id=issue_estimated_hours	${randomHours}
74	gotoIf	storedVars['trackerLabel'] == "Bug" || storedVars['trackerLabel'] == "Task"	endIfSelectTicketType
75	select	//*[@id='issue_custom_field_values_8']	${ticketType}
76	label	endIfSelectTicketType	
77	click	//*['watchers_inputs']/label[contains(.,'${currentUser}')]/input	
78	clickAndWait	css=#issue-form > input[name="commit"]	
79	//CHECK NEW TICKET		
80	verifyAttribute	id=flash_notice@class	flash notice
81	storeText	//*[@id='flash_notice']	numberTicketCreatedAux
82	storeEval	storedVars['numberTicketCreatedAux'].split(" ");	ticketArrayAux
83	storeEval	storedVars['ticketArrayAux'][1].split('#');	ticketArray
84	storeEval	storedVars['ticketArray'][1]	numberTicketCreated
85	verifyText	//*[@id='content']/h2	${trackerLabel} #${numberTicketCreated}
86	verifyText	//div[@class='subject']/div/h3	${subject}
87	verifyText	//*[@id='content']/div[3]/p/a[1]	${currentUser}
88	verifyText	//div[@class='status attribute']/div[2]	New
89	verifyText	//div[@class='priority attribute']/div[2]	${priority}
90	verifyText	//div[@class='assigned-to attribute']/div[2]/a	${currentUser}
91	verifyText	//div[@class='fixed-version attribute']/div[2]/a	${version}
92	gotoIf	storedVars['trackerLabel'] == "Task"	endIfVerifyType
93	verifyText	//div[@class='cf_1 attribute']/div[2]	${type}
94	label	endIfVerifyType	
95	verifyText	//div[@class='start-date attribute']/div[2]	${startDate}
96	verifyText	//div[@class='due-date attribute']/div[2]	${dueDate}
97	verifyText	//div[@class='estimated-hours attribute']/div[2]	${randomHours}.00 h
98	gotoIf	storedVars['trackerLabel'] == "Bug" || storedVars['trackerLabel'] == "Task"	endIfVerifyTicketType
99	verifyText	//*[@id='content']/div[3]/div[2]/div[2]/div[2]/div/div[2]	${ticketType}
100	label	endIfVerifyTicketType	
101	verifyText	//div[@class='description']/div[2]/p	${description}
102	storeEval	new Date().getTime()	endTimeNewTicket
103	storeEval	(${endTimeNewTicket} - ${startTimeNewTicket}) / 1000	scriptExecutionTimeNT
104	echo	${scriptExecutionTimeNT} seconds on New Ticket tab	

A.2 Selenium Java WebDriver

In Java, there are two classes, the first one is named *NewTicket* and is the test to be executed using Java programming language.

```
1  package test;
2
3  import java.text.DateFormat;
4  import java.text.SimpleDateFormat;
5  import java.util.ArrayList;
6  import java.util.Calendar;
7  import java.util.Date;
8  import org.junit.Test;
9  import pageObjects.NewTicketPage;
10 import utility.GlobalMethods;
11
12 public class NewTicket {
13     private GlobalMethods global;
14     private NewTicketPage newTicketPage;
15
16     @Test
17     public void testNewTicket() throws Exception {
18         global = new GlobalMethods();
19         newTicketPage = new NewTicketPage();
20         Long startTimeNewTicket = global.getTime();
21
22         //Data Entry
23         final String initialDescription = "Description field test Automation-";
24         final String initialSubject = "Subject field test Automation-";
25         String currentUser = global.getCurrentUser();
26
27         //Variables
28         String[] tracker = {"Bug","Task","Fix"};
29         ArrayList<String> priorityArray = new ArrayList<String>();
30         ArrayList<String> versionArray = new ArrayList<String>();
31         ArrayList<String> typeArray = new ArrayList<String>();
32
33         // CREATE NEW TICKET
34         newTicketPage.clickNewTicketLink();
35         Integer sizeTracker = newTicketPage.getTrackerSize(tracker);
36         Integer randomTracker = global.generateRandomNumber(sizeTracker, true);
37         global.setTrackerLabel(tracker[randomTracker]);
38         newTicketPage.selTracker(tracker[randomTracker]);
39         String dateTime = String.valueOf(global.getTime());
40         String subject = initialSubject + dateTime;
41         String description = initialDescription + dateTime;
42         Integer numOptionsPriority = newTicketPage.getNumOptionsPriority();
43
44         for(int i = 1; i <= numOptionsPriority; i++) {
45             String priorityAux = newTicketPage.getOptionPriority(i);
46             priorityArray.add(priorityAux);
47         }
48         Integer randomPriority = global.generateRandomNumber(numOptionsPriority, true);
49         String priority = priorityArray.get(randomPriority);
50         Integer numOptionsVersion = newTicketPage.getNumOptionsVersion();
51
52         for(int i = 1; i <= numOptionsVersion; i++) {
53             String versionAux = "";
54             try {
55                 versionAux = newTicketPage.getOptionFixedVersion(i);
56             }
57             catch(org.openqa.selenium.StaleElementReferenceException ex) {
58                 versionAux = newTicketPage.getOptionFixedVersion(i);
59             }
60             versionArray.add(versionAux);
61         }
62         Integer randomVersion = global.generateRandomNumber(numOptionsVersion, true);
63         String version = versionArray.get(randomVersion);
64         String type = "";
65
66         if(tracker[randomTracker] == "Bug" || tracker[randomTracker] == "Fix") {
67             Integer numOptionsType = newTicketPage.getNumOptionsPriority();
68
69             for(int i = 1; i <= numOptionsType; i++) {
70                 String typeAux = newTicketPage.getOptionType(i);
71                 typeArray.add(typeAux);
72             }
73             Integer randomType = global.generateRandomNumber(numOptionsType, true);
74             type = typeArray.get(randomType);
75         }
76         newTicketPage.txtIssueSubject(subject);
```

```
77          newTicketPage.txtDescription(description);
78          newTicketPage.selIssueStatus();
79          priority  = newTicketPage.getPrioritySelected();
80          newTicketPage.selIssueAssignedTo(currentUser);
81          newTicketPage.selIssueFixedVersion(version);
82
83          if(tracker[randomTracker] == "Bug" || tracker[randomTracker] == "Fix") {
84              newTicketPage.selIssueType(type);
85          }
86          String startDate = newTicketPage.getIssueStartDate();
87          Date date = new Date();
88          Calendar calendar = Calendar.getInstance();
89          calendar.setTime(date);
90          calendar.add(Calendar.MONTH, 1);
91          date = calendar.getTime();
92          DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
93          String dueDate = dateFormat.format(date);
94          Integer randomHours = global.generateRandomNumber(9, false);
95          newTicketPage.txtIssueEstimatedHours(String.valueOf(randomHours));
96          newTicketPage.clickSelectUser(currentUser);
97          newTicketPage.clickCreateissue();
98
99          // CHECK NEW TICKET
100         newTicketPage.verifyIssueCreation();
101         String numberTicketCreatedAux = newTicketPage.getNumberTicketCreatedAux();
102         String[] ticketArrayAux = numberTicketCreatedAux.split(" ");
103         String[] ticketArray = ticketArrayAux[1].split("#");
104         String numberTicketCreated = ticketArray[1];
105         newTicketPage.verifyTicketCreated(tracker[randomTracker], numberTicketCreated);
106         global.setNumberTicketCreated(numberTicketCreated);
107         newTicketPage.verifySubject(subject);
108         global.setSubject(subject);
109         newTicketPage.verifyCurrentUser(currentUser);
110         newTicketPage.verifyNewWord();
111         newTicketPage.verifyPriority(priority);
112         global.setPriority(priority);
113         newTicketPage.verifyCurrentUserTable(currentUser);
114         newTicketPage.waitForVersion();
115         newTicketPage.verifyVersion(version);
116         global.setVersion(version);
117
118         if(tracker[randomTracker] == "Bug" || tracker[randomTracker] == "Fix") {
119             newTicketPage.verifyType(type);
120             global.setType(type);
121         }
122         newTicketPage.verifyStartDate(startDate);
123         global.setStartDate(startDate);
124         newTicketPage.verifyDueDate(dueDate);
125         global.setDueDate(dueDate);
126         newTicketPage.verifyRandomHours(String.valueOf(randomHours));
127         global.setRandomHours(String.valueOf(randomHours));
128         global.setRandomTracker(randomTracker);
129         newTicketPage.verifyDescription(description);
130         Long endTimeNewTicket = global.getTime();
131         System.out.println("New Ticket execution time was: " + global.getTimeExecution(startTimeNewTicket,
                endTimeNewTicket));
132     }
133 }
```

The second one is named *NewTicketPage* and consists in the UI elements with their locators and the actions (methods) performed that are called from the *NewTicket* class using Java programming language, it means, it's the implementation of the Page Object Pattern.

```
1   package pageObjects;
2
3   import static org.junit.Assert.assertEquals;
4   import static org.junit.Assert.fail;
5   import org.openqa.selenium.By;
6   import org.openqa.selenium.WebDriver;
7   import org.openqa.selenium.support.ui.Select;
8   import testSuite.TestSuite;
9
10  public class NewTicketPage {
11      private WebDriver driver = TestSuite.driver;
12      private StringBuffer verificationErrors = new StringBuffer();
13
14      private By newTicketLink = By.linkText("New ticket");
15      private By trackerSel = By.xpath("//*[@id='issue_tracker_id']");
```

```
16      private By versionOptions = By.xpath("//*[@id='issue_fixed_version_id']/option");
17      private By priorityOptions = By.xpath("//*[@id='issue_priority_id']/option");
18      private By ticketTypeOptions = By.xpath("//*[@id='issue_custom_field_values_8']/option");
19      private By issueSubject = By.id("issue_subject");
20      private By issueDescription = By.id("issue_description");
21      private By issueStatus = By.id("issue_status_id");
22      private By issueAssignedTo = By.id("issue_assigned_to_id");
23      private By issueFixedVersion = By.id("issue_fixed_version_id");
24      private By issueType = By.id("issue_custom_field_values_1");
25      private By issueStartDate = By.xpath("//*[@id='issue_start_date']");
26      private By issueDueDate = By.xpath("//*[@id='issue_due_date']");
27      private By issueEstimatedHours = By.id("issue_estimated_hours");
28      private By createIssue = By.cssSelector("#issue-form > input[name=\"commit\"]");
29      private By confirmationMessage = By.id("flash_notice");
30      private By numberTicketField = By.xpath("//*[@id='content']/h2");
31      private By subjectField = By.xpath("//div[@class='subject']/div/h3");
32      private By currentUserField = By.xpath("//*[@id='content']/div[3]/p/a[1]");
33      private By newField = By.xpath("//div[@class='status attribute']/div[2]");
34      private By priorityField = By.xpath("//div[@class='priority attribute']/div[2]");
35      private By currentUserTableField = By.xpath("//div[@class='assigned-to attribute']/div[2]/a");
36      private By versionField = By.xpath("//div[@class='fixed-version attribute']/div[2]/a");
37      private By typeField = By.xpath("//div[@class='cf_1 attribute']/div[2]");
38      private By startDateField = By.xpath("//div[@class='start-date attribute']/div[2]");
39      private By dueDateField = By.xpath("//div[@class='due-date attribute']/div[2]");
40      private By randomHoursField = By.xpath("//div[@class='estimated-hours attribute']/div[2]");
41      private By descriptionField = By.xpath("//div[@class='description']/div[2]/p");
42
43
44      public void clickNewTicketLink() {
45          driver.findElement(newTicketLink).click();
46      }
47
48      public int getTrackerSize(String[] tracker) {
49          return tracker.length;
50      }
51
52      public void selTracker(String trackerLabel) {
53          new Select(driver.findElement(trackerSel)).selectByVisibleText(trackerLabel);
54      }
55
56      public String getOptionPriority(Integer numOption) {
57          return driver.findElement(By.xpath("//*[@id='issue_priority_id']/option[" + numOption + "]")).getAttribute("value");
58      }
59
60      public int getNumOptionsVersion() {
61          String dueDate = dateFormat.format(date);
62      }
63
64      public String getOptionFixedVersion(Integer numOption) {
65          return driver.findElement(By.xpath("//*[@id='issue_fixed_version_id']/option[" + numOption +
                "]")).getAttribute("value");
66      }
67
68      public int getNumOptionsPriority() {
69          return (Integer) driver.findElements(priorityOptions).size();
70      }
71
72      public String getOptionType(Integer numOption) {
73          return driver.findElement(By.xpath("//*[@id='issue_custom_field_values_1']/option[" + numOption +
                "]")).getAttribute("value");
74      }
75
76      public int getNumOptionsTicketType() {
77          return (Integer) driver.findElements(ticketTypeOptions).size();
78      }
79
80      public String getOptionTicketType(Integer numOption) {
81          return driver.findElement(By.xpath("//*[@id='issue_custom_field_values_8']/option[" + numOption +
                "]")).getAttribute("value");
82      }
83
84      public void txtIssueSubject(String subject) {
85          driver.findElement(issueSubject).clear();
86          driver.findElement(issueSubject).sendKeys(subject);
87      }
88
89      public void txtDescription(String description) {
90          driver.findElement(issueDescription).clear();
91          driver.findElement(issueDescription).sendKeys(description);
92      }
93
94      public void selIssueStatus() {
95          new Select(driver.findElement(issueStatus)).selectByIndex(0);
96      }
97
98      public String getPrioritySelected() {
```

```
99          return driver.findElement(issueStatus).getText();
100     }
101
102     public void selIssueAssignedTo(String currentUser) {
103         new Select(driver.findElement(issueAssignedTo)).selectByVisibleText(currentUser);
104     }
105
106     public void selIssueFixedVersion(String version) {
107         new Select(driver.findElement(issueFixedVersion)).selectByValue(version);
108     }
109
110     public void selIssueType(String type) {
111         new Select(driver.findElement(issueType)).selectByVisibleText(type);
112     }
113
114     public String getIssueStartDate() {
115         return driver.findElement(issueStartDate).getAttribute("value");
116     }
117
118     public void txtIssueDueDate(String dueDate) {
119         driver.findElement(issueDueDate).clear();
120         driver.findElement(issueDueDate).sendKeys(dueDate);
121     }
122
123     public void txtIssueEstimatedHours(String randomHours) {
124         driver.findElement(issueEstimatedHours).clear();
125         driver.findElement(issueEstimatedHours).sendKeys(randomHours);
126     }
127
128     public void clickSelectUser(String currentUser) {
129         driver.findElement(By.xpath("//*['watchers_inputs']/label[contains(.,'" + currentUser + "')]/input")).click();
130     }
131
132     public void clickCreateissue() {
133         driver.findElement(createIssue).click();
134     }
135
136     public void verifyIssueCreation() {
137         try {
138             assertEquals("flash notice", driver.findElement(confirmationMessage).getAttribute("class"));
139         } catch (Error e) {
140             verificationErrors.append(e.toString());
141         }
142     }
143
144     public String getNumberTicketCreatedAux() {
145         return driver.findElement(confirmationMessage).getText();
146     }
147
148     public void verifyTicketCreated(String trackerLabel, String numberTicketCreated) {
149         try {
150             assertEquals(trackerLabel + " #" + numberTicketCreated, driver.findElement(numberTicketField).getText());
151         } catch (Error e) {
152             verificationErrors.append(e.toString());
153         }
154     }
155
156     public void verifySubject(String subject) {
157         try {
158             assertEquals(subject, driver.findElement(subjectField).getText());
159         } catch (Error e) {
160             verificationErrors.append(e.toString());
161         }
162     }
163
164     public void verifyCurrentUser(String currentUser) {
165         try {
166             assertEquals(currentUser, driver.findElement(currentUserField).getText());
167         } catch (Error e) {
168             verificationErrors.append(e.toString());
169         }
170     }
171
172     public void verifyNewWord() {
173         try {
174             assertEquals("New", driver.findElement(newField).getText());
175         } catch (Error e) {
176             verificationErrors.append(e.toString());
177         }
178     }
179
180     public void verifyPriority(String priority) {
181         try {
182             assertEquals(priority, driver.findElement(priorityField).getText());
183         } catch (Error e) {
184             verificationErrors.append(e.toString());
```

```
185         }
186     }
187
188     public void verifyCurrentUserTable(String currentUser) {
189         try {
190             assertEquals(currentUser, driver.findElement(currentUserTableField).getText());
191         } catch (Error e) {
192             verificationErrors .append(e.toString());
193         }
194     }
195
196     public void waitForVersion() throws InterruptedException {
197         for (int second = 0;; second++) {
198             if (second >= 60) fail("timeout");
199             try { if (driver.findElement(versionField).isDisplayed())
200                 break; } catch (Exception e) {}
201             Thread.sleep(1000);
202         }
203     }
204
205     public void verifyVersion(String version) {
206         try {
207             assertEquals(version, driver.findElement(versionField).getText());
208         } catch (Error e) {
209             verificationErrors .append(e.toString());
210         }
211     }
212
213     public void verifyType(String type) {
214         try {
215             assertEquals(type, driver.findElement(typeField).getText());
216         } catch (Error e) {
217             verificationErrors .append(e.toString());
218         }
219     }
220
221     public void verifyStartDate(String startDate) {
222         try {
223             assertEquals(startDate, driver.findElement(startDateField).getText());
224         } catch (Error e) {
225             verificationErrors .append(e.toString());
226         }
227     }
228
229     public void verifyDueDate(String dueDate) {
230         try {
231             assertEquals(dueDate, driver.findElement(dueDateField).getText());
232         } catch (Error e) {
233             verificationErrors .append(e.toString());
234         }
235     }
236
237     public void verifyRandomHours(String randomHours) {
238         try {
239             assertEquals(randomHours + ".00 h", driver.findElement(randomHoursField).getText());
240         } catch (Error e) {
241             verificationErrors .append(e.toString());
242         }
243     }
244
245     public void verifyDescription(String description) {
246         try {
247             assertEquals(assertEquals(description, driver.findElement(descriptionField).getText());
248         } catch (Error e) {
249             verificationErrors .append(e.toString());
250         }
251     }
252
253     public void verifyErrors() {
254         String verificationErrorString = verificationErrors.toString();
255
256         if (!"".equals( verificationErrorString )) {
257             fail ( verificationErrorString );
258         }
259     }
260 }
```

A.3 Selenium C# WebDriver

In C#, there are two classes, the first one is named *NewTicket* and is the test to be executed using C# programming language.

```
using RedMine.src.main.utility;
using RedMine.src.main.pageObjects;
using System;
using System.Collections.Generic;
using OpenQA.Selenium;

namespace RedMine.src.test.test {
    public class NewTicket {
        private GlobalMethods global;
        private NewTicketPage newTicketPage;

        public void testNewTicket() {
            global = new GlobalMethods();
            newTicketPage = new NewTicketPage();
            long startTimeNewTicket = global.getTime();

            //Data Entry
            String initialDescription = "Description field test Automation-";
            String initialSubject = "Subject field test Automation-";
            String currentUser = global.getCurrentUser();

            //Variables
            String[] tracker = { "Bug", "Task", "Fix" };
            List<String> priorityArray = new List<String>();
            List<String> versionArray = new List<String>();
            List<String> typeArray = new List<String>();


            // CREATE NEW TICKET
            newTicketPage.clickNewTicketLink();
            int sizeTracker = newTicketPage.getTrackerSize(tracker);
            int randomTracker = global.generateRandomNumber(sizeTracker, true);
            global.setTrackerLabel(tracker[randomTracker]);
            newTicketPage.selTracker(tracker[randomTracker]);
            String dateTime = global.getTime().ToString();
            String subject = initialSubject + dateTime;
            String description = initialDescription + dateTime;
            int numOptionsPriority = newTicketPage.getNumOptionsPriority();

            for(int i = 1; i <= numOptionsPriority; i++) {
                String priorityAUx = newTicketPage.getOptionPriority(i);
                priorityArray.Add(priorityAUx);
            }
            int randomPriority = global.generateRandomNumber(numOptionsPriority, true);
            String priority = priorityArray[randomPriority].ToString();
            int numOptionsVersion = newTicketPage.getNumOptionsVersion();

            for(int i = 1; i <= numOptionsVersion; i++) {
                String versionAux = "";

                try {
                    versionAux = newTicketPage.getOptionFixedVersion(i);
                }
                catch (StaleElementReferenceException e) {
                    versionAux = newTicketPage.getOptionFixedVersion(i);
                }
                versionArray.Add(versionAux);
            }
            int randomVersion = global.generateRandomNumber(numOptionsVersion, true);
            String version = versionArray[randomVersion].ToString();
            String type = "";

            if(tracker[randomTracker] == "Bug" || tracker[randomTracker] == "Fix") {
                int numOptionsType = newTicketPage.getNumOptionsPriority();

                for(int i = 1; i <= numOptionsType; i++) {
                    String typeAux = newTicketPage.getOptionType(i);
                    typeArray.Add(typeAux);
                }
                int randomType = global.generateRandomNumber(numOptionsType, true);
                type = typeArray[randomType].ToString();
            }
            newTicketPage.txtIssueSubject(subject);
            newTicketPage.txtDescription(description);
            newTicketPage.selIssueStatus();
            priority = newTicketPage.getPrioritySelected();
```

```
77              newTicketPage.selIssueAssignedTo(currentUser);
78              newTicketPage.selIssueFixedVersion(version);
79
80              if(tracker[randomTracker] == "Bug" || tracker[randomTracker] == "Fix") {
81                  newTicketPage.selIssueType(type);
82              }
83              String startDate = newTicketPage.getIssueStartDate();
84              String dueDate = DateTime.Now.AddMonths(1).ToString("yyyy-MM-dd");
85              newTicketPage.txtIssueDueDate(dueDate);
86              int randomHours = global.generateRandomNumber(9, false);
87              newTicketPage.txtIssueEstimatedHours(randomHours.ToString());
88              newTicketPage.clickSelectUser(currentUser);
89              newTicketPage.clickCreateissue();
90
91              // CHECK NEW TICKET
92              newTicketPage.verifyIssueCreation();
93              String numberTicketCreatedAux = newTicketPage.getNumberTicketCreatedAux();
94              String[] ticketArrayAux = numberTicketCreatedAux.Split(' ');
95              String[] ticketArray = ticketArrayAux[1].Split('#');
96              String numberTicketCreated = ticketArray[1];
97              newTicketPage.verifyTicketCreated(tracker[randomTracker], numberTicketCreated);
98              global.setNumberTicketCreated(numberTicketCreated);
99              newTicketPage.verifySubject(subject);
100             global.setSubject(subject);
101             newTicketPage.verifyCurrentUser(currentUser);
102             newTicketPage.verifyNewWord();
103             newTicketPage.verifyPriority(priority);
104             global.setPriority(priority);
105             newTicketPage.verifyCurrentUserTable(currentUser);
106             newTicketPage.verifyVersion(version);
107             global.setVersion(version);
108
109             if(tracker[randomTracker] == "Bug" || tracker[randomTracker] == "Fix") {
110                 newTicketPage.verifyType(type);
111                 global.setType(type);
112             }
113             newTicketPage.verifyStartDate(startDate);
114             global.setStartDate(startDate);
115             newTicketPage.verifyDueDate(dueDate);
116             global.setDueDate(dueDate);
117             newTicketPage.verifyRandomHours(randomHours.ToString());
118             global.setRandomHours(randomHours.ToString());
119             global.setRandomTracker(randomTracker);
120             newTicketPage.verifyDescription(description);
121             long endTimeNewTicket = global.getTime();
122             Console.WriteLine("New Ticket execution time was: " + global.getTimeExecution(startTimeNewTicket,
                    endTimeNewTicket));
123         }
124     }
125 }
```

The second one is named *NewTicketPage* and consists in the UI elements with their locators and the actions (methods) performed that are called from the *NewTicket* class using C# programming language, it means, it's the implementation of the Page Object Pattern.

```
1  using OpenQA.Selenium;
2  using RedMine.src.test.testSuite;
3  using System;
4  using OpenQA.Selenium.Support.UI;
5  using System.Diagnostics;
6  using System.Text;
7  using NUnit.Framework;
8
9  namespace RedMine.src.main.pageObjects {
10     class NewTicketPage {
11         private IWebDriver driver = TestSuite.driver;
12         private StringBuilder verificationErrors = new StringBuilder();
13
14         private By newTicketLink = By.LinkText("New ticket");
15         private By trackerSel = By.XPath("//*[@id='issue_tracker_id']");
16         private By versionOptions = By.XPath("//*[@id='issue_fixed_version_id']/option");
17         private By priorityOptions = By.XPath("//*[@id='issue_priority_id']/option");
18         private By ticketTypeOptions = By.XPath("//*[@id='issue_custom_field_values_8']/option");
19         private By issueSubject = By.Id("issue_subject");
20         private By issueDescription = By.Id("issue_description");
21         private By issueStatus = By.Id("issue_status_id");
22         private By issueAssignedTo = By.Id("issue_assigned_to_id");
23         private By issueFixedVersion = By.Id("issue_fixed_version_id");
```

```
24          private By issueType = By.Id("issue_custom_field_values_1");
25          private By issueStartDate = By.XPath("//*[@id='issue_start_date']");
26          private By issueDueDate = By.XPath("//*[@id='issue_due_date']");
27          private By issueEstimatedHours = By.Id("issue_estimated_hours");
28          private By createIssue = By.CssSelector("#issue-form > input[name=\"commit\"]");
29          private By confirmationMessage = By.Id("flash_notice");
30          private By numberTicketField = By.XPath("//*[@id='content']/h2");
31          private By subjectField = By.XPath("//div[@class='subject']/div/h3");
32          private By currentUserField = By.XPath("//*[@id='content']/div[3]/p/a[1]");
33          private By newField = By.XPath("//div[@class='status attribute']/div[2]");
34          private By priorityField = By.XPath("//div[@class='priority attribute']/div[2]");
35          private By currentUserTableField = By.XPath("//div[@class='assigned-to attribute']/div[2]/a");
36          private By versionField = By.XPath("//div[@class='fixed-version attribute']/div[2]/a");
37          private By typeField = By.XPath("//div[@class='cf_1 attribute']/div[2]");
38          private By startDateField = By.XPath("//div[@class='start-date attribute']/div[2]");
39          private By dueDateField = By.XPath("//div[@class='due-date attribute']/div[2]");
40          private By randomHoursField = By.XPath("//div[@class='estimated-hours attribute']/div[2]");
41          private By ticketTypeField = By.XPath("//*[@id='content']/div[3]/div[2]/div[2]/div[2]/div/div[2]");
42          private By descriptionField = By.XPath("//div[@class='description']/div[2]/p");
43
44
45          public void clickNewTicketLink() {
46              driver.FindElement(newTicketLink).Click();
47          }
48
49          public int getTrackerSize(String[] tracker) {
50              return tracker.Length;
51          }
52
53          public void selTracker(String trackerLabel) {
54              new SelectElement(driver.FindElement(trackerSel)).SelectByText(trackerLabel);
55          }
56
57          public String getOptionPriority(int numOption) {
58              return driver.FindElement(By.XPath("//*[@id='issue_priority_id']/option[" + numOption +
                        "]")).GetAttribute("value");
59          }
60
61          public int getNumOptionsVersion() {
62              return (int)driver.FindElements(versionOptions).Count;
63          }
64
65          public String getOptionFixedVersion(int numOption) {
66              return driver.FindElement(By.XPath("//*[@id='issue_fixed_version_id']/option[" + numOption +
                        "]")).GetAttribute("value");
67          }
68
69          public int getNumOptionsPriority() {
70              return (int)driver.FindElements(priorityOptions).Count;
71          }
72
73          public String getOptionType(int numOption) {
74              return driver.FindElement(By.XPath("//*[@id='issue_custom_field_values_1']/option[" + numOption +
                        "]")).GetAttribute("value");
75          }
76
77          public int getNumOptionsTicketType() {
78              return (int)driver.FindElements(ticketTypeOptions).Count;
79          }
80
81          public String getOptionTicketType(int numOption) {
82              return driver.FindElement(By.XPath("//*[@id='issue_custom_field_values_8']/option[" + numOption +
                        "]")).GetAttribute("value");
83          }
84
85          public void txtIssueSubject(String subject) {
86              driver.FindElement(issueSubject).Clear();
87              driver.FindElement(issueSubject).SendKeys(subject);
88          }
89
90          public void txtDescription(String description) {
91              driver.FindElement(issueDescription).Clear();
92              driver.FindElement(issueDescription).SendKeys(description);
93          }
94
95          public void selIssueStatus() {
96              new SelectElement(driver.FindElement(issueStatus)).SelectByIndex(0);
97          }
98
99          public String getPrioritySelected() {
100             return driver.FindElement(issueStatus).Text;
101         }
102
103         public void selIssueAssignedTo(String currentUser) {
104             new SelectElement(driver.FindElement(issueAssignedTo)).SelectByText(currentUser);
105         }
```

```
106
107         public void selIssueFixedVersion(String version) {
108             driver.FindElement(issueFixedVersion)).SelectByValue(version);
109         }
110
111         public void selIssueType(String type) {
112             new SelectElement(driver.FindElement(issueType)).SelectByText(type);
113         }
114
115         public String getIssueStartDate() {
116             return driver.FindElement(issueStartDate).GetAttribute("value");
117         }
118
119         public void txtIssueDueDate(String dueDate) {
120             driver.FindElement(issueDueDate).Clear();
121             driver.FindElement(issueDueDate).SendKeys(dueDate);
122         }
123
124         public void txtIssueEstimatedHours(String randomHours) {
125             driver.FindElement(issueEstimatedHours).Clear();
126             driver.FindElement(issueEstimatedHours).SendKeys(randomHours);
127         }
128
129         public void clickSelectUser(String currentUser) {
130             driver.FindElement(By.XPath("//*['watchers_inputs']/label[contains(.,'" + currentUser + "')]/input")).Click();
131         }
132
133         public void clickCreateissue() {
134             driver.FindElement(createIssue).Click();
135         }
136
137         public void verifyIssueCreation() {
138             try {
139                 Assert.AreEqual("flash notice", driver.FindElement(confirmationMessage).GetAttribute("class"));
140             }
141             catch (Exception e) {
142                 verificationErrors.Append(e.ToString());
143             }
144         }
145
146         public String getNumberTicketCreatedAux() {
147             return driver.FindElement(confirmationMessage).Text;
148         }
149
150         public void verifyTicketCreated(String trackerLabel, String numberTicketCreated) {
151             try {
152                 Assert.AreEqual(trackerLabel + " #" + numberTicketCreated, driver.FindElement(numberTicketField).Text);
153             }
154             catch (Exception e) {
155                 verificationErrors.Append(e.ToString());
156             }
157         }
158
159         public void verifySubject(String subject) {
160             try {
161                 Assert.AreEqual(subject, driver.FindElement(subjectField).Text);
162             }
163             catch (Exception e) {
164                 verificationErrors.Append(e.ToString());
165             }
166         }
167
168         public void verifyCurrentUser(String currentUser) {
169             try {
170                 Assert.AreEqual(currentUser, driver.FindElement(currentUserField).Text);
171             }
172             catch (Exception e) {
173                 verificationErrors.Append(e.ToString());
174             }
175         }
176
177         public void verifyNewWord() {
178             try {
179                 Assert.AreEqual("New", driver.FindElement(newField).Text);
180             }
181             catch (Exception e) {
182                 verificationErrors.Append(e.ToString());
183             }
184         }
185
186         public void verifyPriority(String priority) {
187             try {
188                 Assert.AreEqual(priority, driver.FindElement(priorityField).Text);
189             }
190             catch (Exception e) {
191                 verificationErrors.Append(e.ToString());
```

```
192             }
193         }
194
195         public void verifyCurrentUserTable(String currentUser) {
196             try {
197                 Assert.AreEqual(currentUser, driver.FindElement(currentUserTableField).Text);
198             }
199             catch (Exception e) {
200                 verificationErrors.Append(e.ToString());
201             }
202         }
203
204         public void verifyVersion(String version) {
205             try {
206                 Assert.AreEqual(version, driver.FindElement(versionField).Text);
207             }
208             catch (Exception e) {
209                 verificationErrors.Append(e.ToString());
210             }
211         }
212
213         public void verifyType(String type) {
214             try {
215                 Assert.AreEqual(type, driver.FindElement(typeField).Text);
216             }
217             catch (Exception e) {
218                 verificationErrors.Append(e.ToString());
219             }
220         }
221
222         public void verifyStartDate(String startDate) {
223             try {
224                 Assert.AreEqual(startDate, driver.FindElement(startDateField).Text);
225             }
226             catch (Exception e) {
227                 verificationErrors.Append(e.ToString());
228             }
229         }
230
231         public void verifyDueDate(String dueDate) {
232             try {
233                  Assert.AreEqual(dueDate, driver.FindElement(dueDateField).Text);
234             }
235             catch (Exception e) {
236                 verificationErrors.Append(e.ToString());
237             }
238         }
239
240         public void verifyRandomHours(String randomHours) {
241             try {
242                 Assert.AreEqual(randomHours + ".00 h", driver.FindElement(randomHoursField).Text);
243             }
244             catch (Exception e) {
245                 verificationErrors.Append(e.ToString());
246             }
247         }
248
249         public void verifyDescription(String description) {
250             try {
251                 Assert.AreEqual(description, driver.FindElement(descriptionField).Text);
252             }
253             catch (Exception e) {
254                 verificationErrors.Append(e.ToString());
255             }
256         }
257
258         public void verifyErrors() {
259             String verificationErrorString = verificationErrors.ToString();
260
261             if (!"".Equals(verificationErrorString)) {
262                 Debug.Fail(verificationErrorString);
263             }
264         }
265     }
266 }
```

The red lines indicate only a syntax difference, mainly between Java and C#, because for the native language of Selenium IDE, the syntax is different from any programming language. A real difference in the code can be seen in line 70 of Selenium IDE, 87 to 93 of Java (*NewTicket.java* file), and 84 of C# (*NewTicket.cs* file). In these lines of code, the described functionality consist in adding a month to the current date. It can be noted

that such addition is easier to be done in C#, while in Java several lines of code are needed, and in Selenium IDE, JavaScript language is used so the logic must be written in one line, what makes it more difficult.

A.4 Scripts

In the following link, all the scripts created during this thesis can be found, for the small, medium and large application in the three automation tools, Selenium IDE, Java and C#. They are going to be available at least until 31/12/2019, after that the scripts can be requested to santip176@hotmail.com.

https://www.dropbox.com/s/0obdi2go111ebgs/Automation%20Scripts.zip?dl=0

A.5 Description of the test cases

In the tables A.2, A.3, and A.4, the name of all the test cases that were automated in this thesis are shown. In the table A.2, the test case for the small application is shown. In the table A.3. the test cases for the medium application are shown, and in the table A.4, the test cases for the large application are shown.

Test Script	**Test Case**
AddTask	Add new task

Table A.2: Test Case for the small application.

Test Script	**Test Case**
Login	Login
Overview	Get the current user
	Tickets overview
NewTicket	Create a new ticket
	Verify the creation of the new ticket
SearchTicket	Search a ticket
TimeTracking	Add a new time
	Verify the time added
	Verify the total time

Table A.3: Test Cases for the medium application.

Test Script	**Test Case**
Login	Login
	Add a new role
	Delete a role
	Add a new user

	Assign role to user
	Modify a user
	Delete a user
	Add pages to role
	Remove pages to role
	Add functions to role
	Remove functions to role
Workflow	Delete workflow
	Search workflow
	Publish workflow
Metadata	Create metadata numeric type
	Create metadata multiple choice type
	Create metadata multiple choice with dependencies type
	Delete a metadata
	Create a set
	Add a metadata to a set
	Delete a set
Import	Import file with workflows
Browse & Search	Search using special characters
	Search using blank space
	Search a project with the complete name
	Search a project with a word of the project
	Search a project with a word of the procedure
	Search a procedure with the complete name
	Search a procedure with a word of the procedure
	Filter by update year
	Filter by update month
	Filter by discipline
	Filter by category
	Filter by subcategory
	Filter by product year
	Export project through the export button
	Export project through the export AC button
	See properties to procedure
	See properties to image
	Export project through the export icon
	Export project through the export AC icon
	Filter by show only my content
	Filter by completed workflow
	See all pages through pagination
	See more data through items dropdown
	Filter by product title of the section one
	Filter by project title of the section one
	Filter by current workflow task stage of the section one

	Filter by product title of the section two
	Filter by project title of the section two
	Filter by current workflow task stage of the section two
	Filter by current assigned user of the section two
	Pass procedures to next step
	Edit file through task properties
	Download file through task properties
	Replace file through task properties
	Reassign task
	Search the project by partial name of the project
	Search the project by complete name of the project
	Search the project by partial task stage
	Search the project by complete task stage
	Search the project by creation date
	Search the project by assigned user
	Search the project by task due day
	Search the project by due date
	Search the project filtering by show all assignments
	Search the project filtering by show my assignations
	Search using special characters
	Search using blank space
	Order by due date
	Order by project name
	Order by task stage
	Order by creation date
	Order by assigned user
	Order by task due day
	Filter by all content
	Filter by pass due
	Filter by due this week
	Filter by next week
	Filter by due soon
	Edit the project from option pencil
	Validate the option edit pencil
	Download file through button windows main
	Assign the task of the option task stage
	Assign the project of the option assigned user

Table A.4: Test Cases for the large application.

Bibliography

[1] Alan J. Perlis. “Epigrams on Programming”. In: (1982).

[2] Bertrand Meyer. “Seven Principles of Software Testing”. In: *Computer* 41 (2008), pp. 99–101. ISSN: 0018-9162.

[3] Robert V. Binder. *Testing object-oriented systems: models, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-80938-9.

[4] Pierre Bourque, R. E Fairley, and IEEE Computer Society. *Guide to the software engineering body of knowledge*. 3rd ed. 2014. ISBN: 0-7695-5166-1.

[5] Glenford J. Myers, Tom Badgett, and Corey Sandler. *The Art of Software Testing*. 2nd. John Wiley & Sons, Inc., 2004. ISBN: 0-471-46912-2.

[6] Robert L. Glass. *Facts and Fallacies of Software Engineering”*. 1st. Addison Wesley, 2002. ISBN: 0-321-11742-5.

[7] *IEEE Standard Glossary of Software Enginnering Terminology*. Institute of Electrical and Electronics Engineers, 1990. ISBN: 1-55937-067-X.

[8] “General Principles of Software Validation; Final Guidance for Industry and FDA Staff”. In: (2002).

[9] Cem Kaner. “The Ongoing Revolution in Software Testing”. In: (2004).

[10] Shiva Kumar. “An Effective Handbook for Implementing Software Test Strategies”. In: (2001).

[11] Roger S. Pressman. *Software engineering: a practitioner’s approach*. 7th ed. New York: McGraw-Hill Higher Education, 2010. ISBN: 978-0-07-337597-7.

[12] Stefan J. Galler and Bernhard K. Aichernig. “Survey on test data generation tools: An evaluation of white- and gray-box testing tools for C#, C++, Eiffel, and Java”. In: *International Journal on Software Tools for Technology Transfer* (2014), pp. 727–751. ISSN: 1433-2779, 1433-2787.

[13] Mian Asbat Ahmad. “New Strategies for Automated Random Testing”. PhD thesis. York, England: The University of York, York, 2013.

[14] Ian Sommerville. *Software Engineering*. 7th ed. Addison Wesley, 2004. ISBN: 978-0-321-21026-5.

[15] Erich Gamma and Kent Beck. “Junit A Cook’s Tour”. In: 4 (1999).

[16] Andreas Leitner et al. “Reconciling Manual and Automated Testing: The AutoTest Experience”. In: *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. Waikoloa, HI, USA: IEEE, 2007.

[17] Cem Kaner. “Pitfalls and Strategies in Automated Testing”. In: (1997).

[18] John Watkins. *Testing IT: An Off-the-Shelf Software Testing Process*. 2001. ISBN: 0-521-79546-X.

[19] E. Miller. “Advanced methods in automated software test”. In: *Proceedings. Conference on Software Maintenance 1990*. San Diego, CA, USA: IEEE Comput. Soc. Press, 1990. ISBN: 978-0-8186-2091-1.

[20] Shivprasad Koirala and Sham Sheikh. *Software testing interview questions*. Computer science series. Hingham, Mass: Infinity Science Press, 2008. ISBN: 978-1-934015-24-7.

[21] James Bach. *Agile Test Automation*. 2003.

[22] Elfriede Dustin, Thom Garrett, and Bernie Gauf. *Implementing Automated Software Testing: How to lower costs while raising quality*. 1st. Pearson Education, Inc, 2009. ISBN: 978-0-321-58051-1.

[23] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: Introduction, management, and performance*. 1st. Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-43287-0.

[24] Stefan Berner, Roland Weber, and Rudolf K Keller. “Observations and Lessons Learned from Automated Testing”. In: 2005, pp. 571–579.

[25] Mark Fewster and Dorothy Graham. “Software Test Automation: Effective Use of Test Execution Tools. Published by Addison-Wesley, Harlow, Essex, U.K., 1999. ISBN: 0-201-33140-3, 574 pages”. In: *Software Testing, Verification and Reliability* (1999). ISSN: 0960-0833, 1099-1689.

[26] Randall W Rice. “Surviving the Top Ten Challenges of Software Test Automation”. In: 2003.

[27] David Hunt. *Software Test-on-Demand - A Smarter Way*. 2012.

[28] Selenium Project. *Selenium Documentation*. Copyright 2008–2012. URL: http://docs.seleniumhq.org/docs/ (visited on 10/16/2018).

[29] Raimund Hocke. *Sikuli*. Copyright 2017. URL: http://sikulix.com/ (visited on 10/16/2018).

[30] Auqtus AB. *Eye Automate*. Copyright 2018. URL: http://www.eyeautomate.com/ (visited on 10/16/2018).

[31] SmartBear Software. *Test Complete Features*. Copyright 2018. URL: https://smartbear.com/product/testcomplete/features/ (visited on 10/16/2018).

[32] *UISpec4J*. 2014. URL: http://github.com/UISpec4J/UISpec4J (visited on 10/16/2018).

[33] Christian Hargraves. *Jameleon - An automated testing tool*. Copyright 2003–2008. URL: http://jameleon.sourceforge.net/ (visited on 10/16/2018).

[34] *Worksoft Certify*. Copyright 2018. URL: http://www.worksoft.com/products/worksoft-certify (visited on 10/16/2018).

[35] *Qualitia*. Copyright 2018. URL: http://www.qualitiasoft.com/ (visited on 10/16/2018).

[36] Viktor Zigo. *Xpather*. Copyright 2005–2009. URL: http://xpath.alephzarro.com/ (visited on 10/16/2018).

[37] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. 1st. Addison Wesley Professional, 2009. ISBN: 0-321-57936-4.

[38] Hui Liu and Hee Beng Kuan Tan. "Covering code behavior on input validation in functional testing". In: *Information and Software Technology* 51 (2009), pp. 546–553. ISSN: 09505849.

[39] Pankaj Jalote Vipindeep V. "List of Common Bugs and Programming Practices to avoid them". In: (2005).

[40] Raghuram Bharathan. *Apache Maven Cookbook*. 2015. ISBN: 1-78528-612-9.

[41] JUnit. *JUnit*. Copyright 2002–2018. URL: http://www.junit.org/junit4/ (visited on 10/16/2018).

[42] Selenium. *Page Objects*. 2015. URL: http://github.com/SeleniumHQ/selenium/wiki/PageObjects (visited on 10/16/2018).

[43] Linda Hayes. "Automated Testing Handbook". In: (1999).

[44] IEEE Spectrum. *The 2018 Top Programming Languages*. Copyright 2018. URL: http://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages (visited on 10/16/2018).