Haaga-Helia
University of Applied Sciences

Automation Testing:
Implementation Methods and Scripting

Mari Pasanen

Bachelor's Thesis
Degree Programme in
Business Information
Technology
2017

Haaga-Helia
University of Applied Sciences

Abstract

21.11.2017

Author
Mari Pasanen

Degree programme
Business Information Technology

Report/thesis title

Automation Testing: Implementation Methods and Scripting

Number of pages and appendix pages
53 + 3

This thesis presents the basic concepts of automation testing and testing in general. The idea was to learn about automation testing for myself in case of possible future endeavours, but also to provide an easily accessible tutorial for those who might benefit from the knowledge as well but feel a bit overwhelmed about where to begin.

The thesis comprises of presenting automation testing concepts with SDLC and V-Model, different types of testing methods, introduction to both test management and test automation tools, basics of the testing process in general, possible risks, a case study to demonstrate an automated login case in action, and finally the final conclusions regarding the process of learning from the topic.

The case study was written in the form of a beginner tutorial, which gives step by step instructions on how to configure and build an environment to conduct a simplified case for a login process for a travel web site. The utilized programming language and environment was Java with Eclipse Oxygen, the tool selected was Selenium WebDriver and the assisting framework was selected as Cucumber. The overall final result was a working demo, which was documented for easy replication in case of need for learning.

Keywords
Test Automation, Beginner tutorial, Testing process, Cucumber, Selenium Webdriver, Java

Table of contents

1 Introduction

Testing comprises of multiple techniques and phases to be implemented in software and system testing in the lifecycle of development and maintenance. Manual testing through test runs with pre-defined test cases on necessary functions is the most normative way of understanding testing, but with huge systems and long development lifecycles comes the problem of repetition, the sheer amount of implementable test runs, and the required coverage through test cases.

Automation testing is a form of testing that enables repetitive method for running test cases and reducing manual work, and it also provides easier comparison between earlier test runs with new results. This becomes essential when considering regression tests, which are necessary to conduct after minor development, changes to the current system and version updates as an example. When considering large-scale systems comprising of hundreds of manual labour test cases, it would become almost an impossible task to test all related changes or entire systems whenever a change takes place. Automation testing can provide an easy tool to tackle the problem of manual repetition, by using the base result and calibrated test cases to check the outcome for a passed or failed test, which would only require action had there been any errors due to any correlating modifications.

Corresponding to the today's technological advancements and elaborately constructed systems, for most testing specialists it is almost a requirement to at least understand the basics of test automation, since it is a vital part to take into consideration during the test planning phase and when preparing to test future releases or builds. Unfortunately for many, the topic does seem very unapproachable and challenging due to the very technical aspect of the subject. As many testing specialists might only use planning and test execution in their everyday work lives, test automation does require understanding of a programming language and how to configure the environments, if one chooses to construct as accurate test suites as possible. Thus, the reason to choose the topic subject was not only to familiarize myself with automation testing concepts, but also to provide an easily accessible tutorial for those who might benefit from the knowledge as well but feel a bit overwhelmed about where to begin.

This thesis will tackle the subject by introducing the very basics of testing itself and how test automation can be integrated into the process. A few of the most popular test automation tools and frameworks will be introduced, for the reader to grasp the essence of their usage and the way they work from the perspective of the subject and in order to under-

stand the methods provided. Scripting is referred in the context of the thesis as the required programming scripts for the automated functionality and for the test cases which support the implementation. The thesis will also contain a basic but descriptive demonstration for the test automation implementation life cycle as a short case study, for the reader to see the actual process in action. Finally, a central keywords list can be found as an appendix at the very end of the thesis, to clarify some of the words and expression that may not be addressed within the thesis as profoundly.

2 Automation testing concepts

This chapter focuses on giving an insight into what automation testing is and why it would be beneficial to use. Some methodologies such as the Software Development Life Cycle, and its derivative, the V-Model will be introduced in order to give a basic idea of the software development process and how testing in general fits into it.

2.1 What is automation testing?

Automation testing is essentially tool-assisted testing, as the base function to test relies on the usage of a specified software that is used to automate, control and report the automation testing process of the system under test or SUT.
The concept of automation testing can be understood broadly and can consist of the process of automating manual tests, automating tests that can't be done manually or only partially automating the test case.
When it has been decided upon to implement automation testing to a developed or existing system, the implementation itself should be considered as a separate test implementation project as the costliness of test automation is the most prevalent during the designing, early implementation phase and during the maintenance of scripts. The actual running of the automated tests does not require manual and repetitive labour nor human interactions and can be done i.e. during nightly runs or while the supervising tester is doing something completely else.
Hence, the difference between manual and automated tests comes down to differences in implementation, costs, and human input. Therefore, there also lies a similar issue as with manual tests that if the automated tests are not implemented correctly then the result will be false. This ultimately may not even be shown on the report as false which then leads to obvious further problems if the configuration issues are not detected properly. Similarly, a manual tester may have a false result if the tester is not knowledgeable of the correct end-result or the process to be tested. (Sahla 18 September 2017.)

2.2 Why implement automation testing?

Automation testing itself can be an enormous benefit when it comes to repeating tests, testing manually unreachable background operations, and validating that integrated parts work together within the system. For instance, regression testing is one of the most popular phases of test automation implementation, since during this phase the modifications and changes of a system are tested to verify that everything works as before.

Performance tests can also benefit from the usage of test automation, since testing the capabilities of an application to handle large amounts of simultaneous traffic requires several test users to overload the system. Manually it would be time consuming if even impossible to create and manoeuvre i.e. 10000 users and the behaviour during test execution, but with the use of automation testing, simulating several users at once would decrease the workload and enable testers even to perform such tests.
If automation testing is implemented correctly, the benefit will definitely be the reduced testing time and costs, as the need to use several testers every single time to validate the same results can be almost completely eliminated for already run manual tests through automation. This does not necessarily mean that automation testing can replace manual testers entirely, as the need for tester's skills and understanding of the process is required at least before automating all tests, automation testing and manual testing are more likely to be complementary towards each other. (Dustin, Garrett & Gauf 2009, 23-24.)

2.3 SDLC (Software Development Life Cycle)

This chapter focuses on describing the life cycle of software development, where both manual and automated testing is an important factor. Software or Systems Development Life Cycle enables certain structure and understanding through models and methodology on how to proceed and structure the phases within the development of a software, application or system and is a vital part of understanding what a development project may entail for everyone involved.
What is usually included in the methodology framework is usually the following (Figure 1):

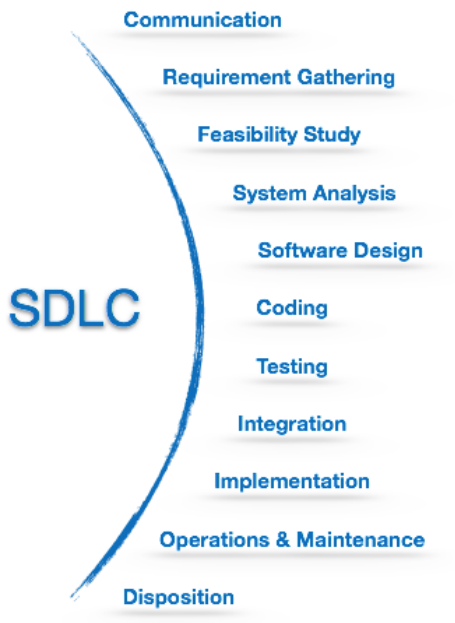

Figure 1 Common SDLC processes (Tutorialspoint.com 2017a).

Starting from the communication, which refers to the negotiation phase with the potential client on the terms of the project. During requirement gathering, the client and stakeholders are interviewed in order to map out the needs on the system performance and what is expected from the system in general. Feasibility study then combines the information gathered from the client and provides a plan for the software process, and a rough estimate for the costs and the technological challenges are mapped out. During the system analysis phase, the developers go through the limitations and problems the system may phase, and also the project scope is established. In the software design stage, the requirements and the system analysis findings are combined into a tangible document on how the system actually will be developed, and all required data dictionaries and pseudocode logic is meticulously provided. After the design phases have been completed, then the actual development and coding takes place during which errors are minimized as well as possible. The testing phase takes into account what has been developed and what should be tested the way that the product works the way the client intended, during this stage it is quite common for a long process of error management and fixing until the product is ready for phases to come. The integration stage is when all the libraries, data and possible interfaces are integrated to the system or program seamlessly. Implementation phase is when the product is actually taken into production for the client and installed on required workspaces, and thus ready for final use. Operation and maintenance ensures that the system works as intended and any downtime or issue is addressed as soon as problems occur, this stage also may contain the need for minor development where small components will be added incrementally as needed while the system is in production. Disposition finally means the stage where over time the software or parts of its data or functionality may become obsolete for future use, hence the stage may include data archiving or even the possibility of closing down the system completely.

The SDLC framework provides several models to be used in software engineering and development projects. Models such as the Waterfall Model, Iterative Model, Spiral Model, V-Model, Big Bang Model and the Agile Development Model among others from which only a few will be introduced (Tutorialspoint.com 2017b). Probably the most well-known model in SDLC is the “Waterfall Model”, where the idea is to move through each step as a cascading waterfall and performing each step individually. What this means from an agile perspective, is that if a step is skipped or a fatal error is found in earlier steps, then the old-school approach would have been to address the issue during the corresponding step. Once the entire cycle of steps has been completed, then the steps will be reiterated as long as development continues, or faults will be found in between the steps. Newer approach has been to get rid of the meticulous step-by-step following of the model, and instead agile sprints have been added in between the steps where the team can go back to

the start until the project phase can move forward. An example of a Waterfall Model, with an added agile development side drawn by using the example from Tutorialspoint.com web site, can be seen below (Figure 2):

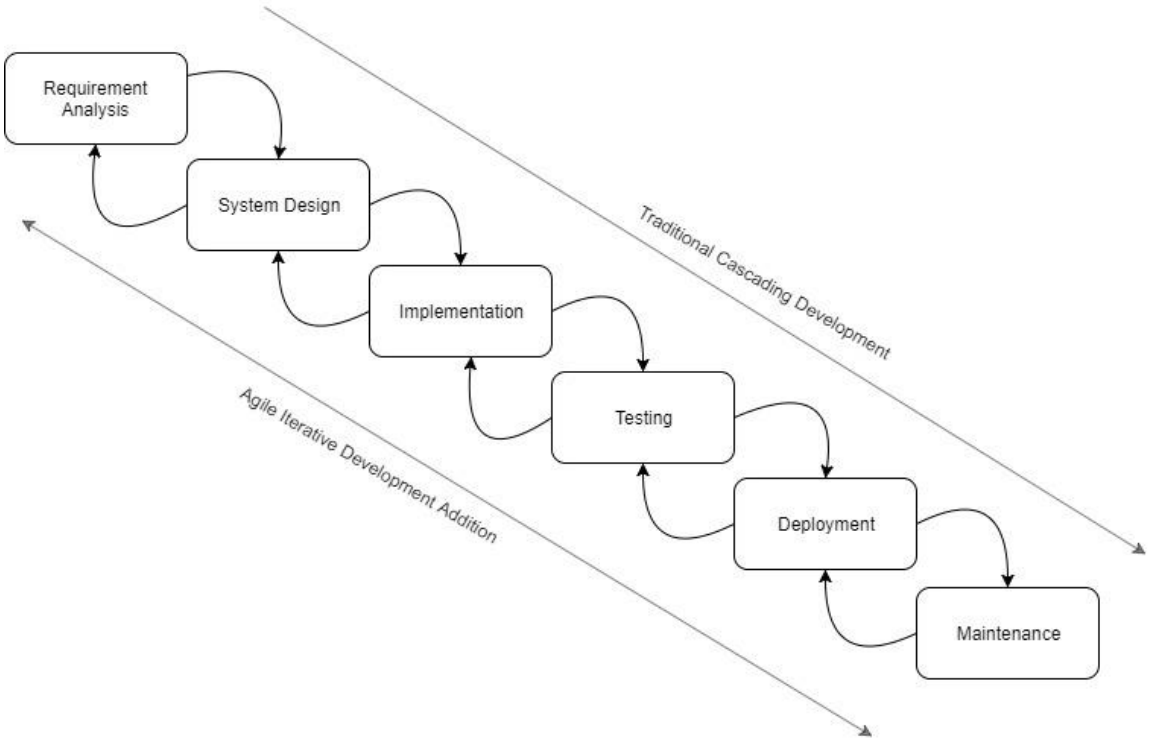

Figure 2 Waterfall Model Example (Tutorialspoint.com 2017c.)

Another well-known model from the SDLC framework called the V-Model, will be presented in the next subchapter in detail since is contains the correlating phases for testing and test design, thus relating more to the topic of the thesis. (Tutorialspoint.com 2017b.)

2.3.1 V-Model

The V-Model is an extension for the waterfall model from the SDLC conceptualisation. The model is also known as the *Verification and Validation model* and has a top-down association structure in the shape of the letter “V”. What is meant by this is that with each top-down development phase, there is an ascending and corresponding test phase that is to be realized simultaneously.

Even if the exact visual versions may differ from each other, especially depending on different schools of thought, the basic contents usually remain the same.

The below picture (Figure 3) has been drawn by using the example from the book “TMap NEXT®” by the Sogeti Netherlands of the CapGemini Group in order to visualize one example of a V-Model.

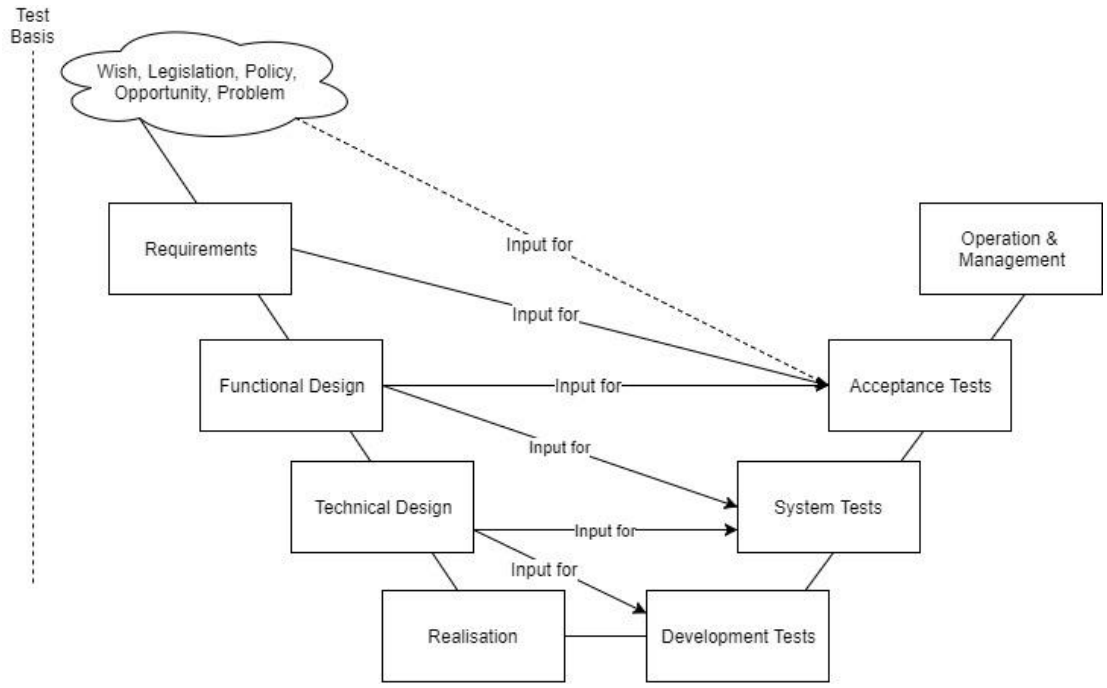

Figure 3 One interpretation of the V-Model (Koomen, T., van der Aalst, L., Broekman, B. & Vroon, M. 2014. 49).

The figure 3 illustrates how the left side represents the technical implementation side starting from the top. The development begins with the preparations and customer requests of the features, and formalizing the required preparations. The second phase would be the official requirements created together with the customer of the needed functionalities of the software or system. After this, comes the functional design phase where the requirements are designed to be implemented for the system in terms of the functionality in general. Then comes the technical design where the actual design and any technical documentation required is formalized. The realisation phase refers to the actual implementation of all the previously formalized designs from the software development aspect and any designs before the realisation are used as the test basis for the test design.

Contrary to the left side, the right side represents the phases for the test design and how the phases correspond with the development side and presents which test cases are to be finalized in which development phases. The acceptance testing is usually done at the end and after all major or critical defects have been dealt with, which would be during the User Acceptance Test (UAT) phase and for which the input is taken from customer input and the designed functionality. This stage is usually only meant for the customer to verify that the system or software performs as requested before being accepted into production. System testing takes input from both the functional and technical designs and revolves around testing the actual system and how it performs under different scenarios. The development test cases are usually done as the last test cases after system and acceptance test cases, but are also the first to be run during the testing phase. The development test

cases usually refer to integration and unit tests. The integration testing phase, which is done before system testing and after unit testing, checks that the interfaces work between the units that are being integrated in order to ensure the flow of data and undisruptive functionality between the separate units (Software Testing Fundamentals 2017a). The unit testing is testing where the written code is tested after certain rows of code have been created, and retested until the functionality can be deemed as fully working for the system. (Tutorialspoint.com 2017d.)

2.4 Test Driven Development (TDD)

Test Driven Development or TDD is an agile software development process that follows short development cycles through repetition. TDD is based mainly on unit testing, and since the cycles are very short, the person usually in charge of the software code ends up creating the unit tests in short code increments i.e. as short automated unit tests. (Johnson, E. 22 July 2015.)
A visualization of the Test Driven Development cycle can be seen from the below picture, drawn by using example from the Agile Alliance glossary on TDD (Figure 4):

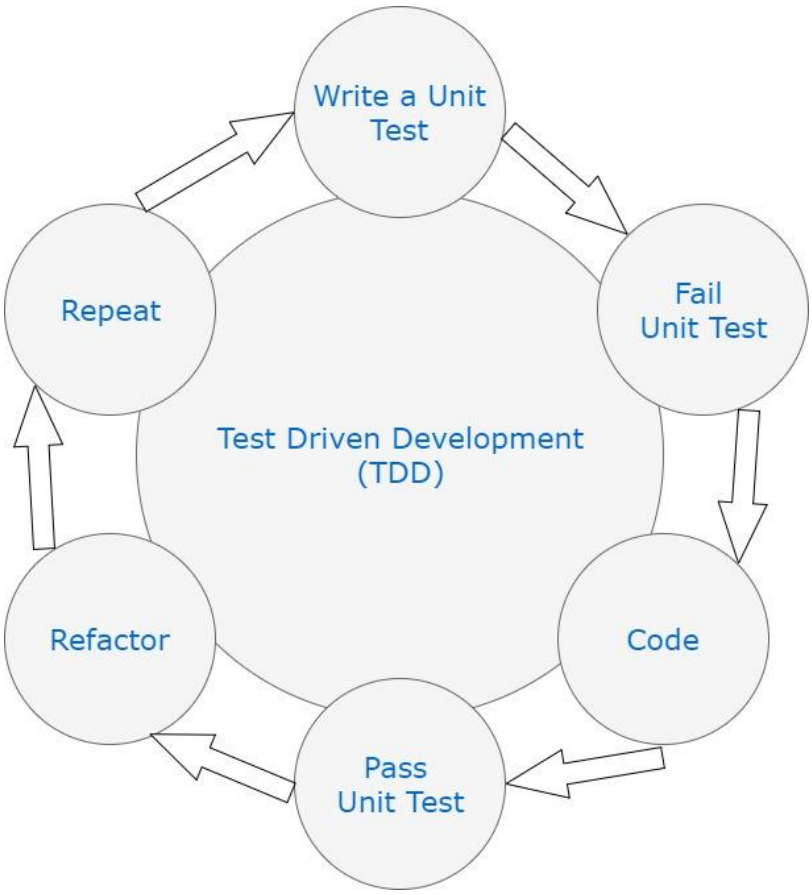

Figure 4 Visualization of Test Driven Development (Agile Alliance 2017a).

As seen from the above diagram (Figure 4), the TDD cycle begins by writing a simple unit test that should fail in the beginning. This can be done either by the programmer or the assigned tester, though usually as mentioned before unit tests fall under the programmer's territory for swifter execution. After the initial unit test has failed, the programmer writes a simple enough piece of code that is enough for the test to finally be passed successfully. (Agile Alliance 2017a).
Refactoring stage in this case by the definition from the Agile Alliance web site glossary is defined as follows:
"Refactoring consists of improving the internal structure of an existing program's source code, while preserving its external behavior." (Agile Alliance 2017b).

What this means is that in the refactoring stage, the code is improved until it matches a set of criteria for a working code and the development can move further in the process. (Agile Alliance 2017b.)
The process is repeated until the functionality of the software is established through the development and tested after new lines of code are added in increments for new unit tests to be completed, it is also preferable for this approach to have only one failing unit test at once. TDD only tests the program stubs and single pieces of a unified system, and therefore does not test how the software works or if the functionality is as it was designed.
In order to test the actual system behaviour, an approach by the name of BDD will be introduced next. (Agile Alliance 2017a.)

2.5 Behaviour Driven Development (BDD)

Behaviour Driven Development or BDD is a derivative and an extension from TDD approach, the difference being that TDD is meant mainly for unit testing where the functionality of a single component is verified through a repetitive process, BDD tests the desired behaviour of the components in general that may be based on certain requirements set for the functionality. The BDD process should enable the whole development team to discuss about how to test, in a way that the communication between the developers, testers and the business analysts is guided in a way that it is clear for all to know what is being tested and i.e for manual testers to create the implementable automation test cases through BDD use (Wilcox, R. 2017). As BDD is a derivative from TDD, one can't introduce the BDD approach to a team that is not already aware of the TDD, especially if the communication is to be guided throughout team members. (Agile Alliance 2017c.)
The BDD process is based on the notion of using user stories in the form of features and/or scenarios through "Given-When-Then"-syntax for testable scripts to illustrate the behaviour of the system in generalized "test cases" that can later be used for the basis of the automated test scripts, opposed to traditional way of conducting automation testing

test cases through regular code. For example, the features can be written by analysts or testers in charge of the test design, whereas the automation can then be implemented by the developers based on the already configured scenarios. Even though the exact syntax has not been always been formally formalized for the approach, there are a few BDD frameworks that support the specified structure i.e. Cucumber and JBehave. Also, a common syntax simplifies the ease of adopting the approach if the user story canvas has already been designed as such. As an example the below figure 5, which was drawn from the example visualization from the Inviqa UK web site, elaborates the structure of the "Given-When-Then"-syntax:

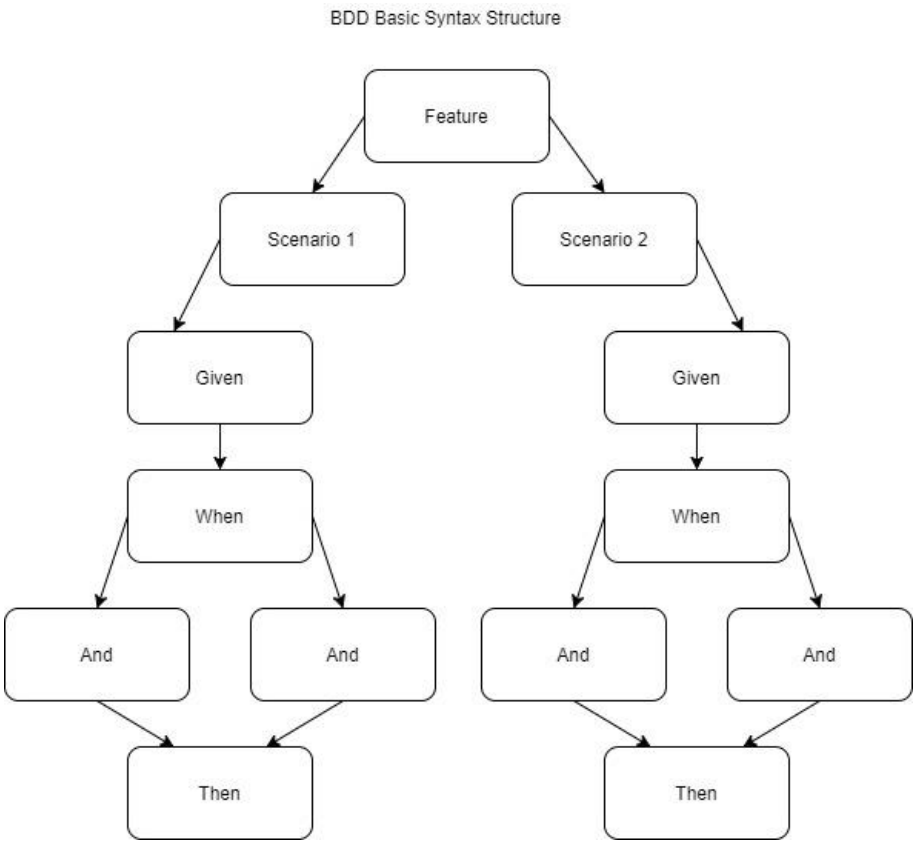

Figure 5 An Example of BDD Feature/Scenario structure (Kudryashov, K. 7 October 2015.).

The feature refers to the situation to be tested, which is divided into scenarios of the situation that will be tested separately. The "Given" refers to the pre-condition, while the "When" refers to the situation when this takes place, for instance when certain value is entered. The "Then" is the end result for the scenario i.e. a certain button that needs to be pushed at the end. All of these steps may contain an additional or optional action in the form of the "And"-condition. An example on how to write these features or scenarios will be addressed again in the chapter about "Gherkin" and the case study.

As mentioned before, BDD relates to the Cucumber framework in terms of using the "Given-When-Then"-syntax through a "Business Readable, Domain Specific" language called Gherkin, which is part of the framework functionality. Cucumber has been chosen as the framework for the upcoming case study, where the functionality of BDD will be demonstrated in action through the use of Gherkin on a later chapter. (Kudryashov, K. 7 October 2015.).

2.6 From manual tests to automation

Automation testing should be considered as software development and in most cases, requires as a separate project as the initial setup can be costly in terms of labour for the design and implementation. It is good to remember that not everything should be automated as the benefits may stop there and in reverse cause additional costs without the gained advantage.
The entire process requires almost always:

- Planning: Requirements, user stories and descriptions of features.
- Design: Designing and constructing environments, selecting frameworks and primary tools and possible assisting tools.
- Implementation: The actual coding of test scripts.
- Testing: It is also advisable to test initially that the setup and implemented tests work, this can be easily forgotten as human input can contain errors if not verified before use.
- Deployment: Deploying the tests to the executable system and the verification of results.

Due to this, it is virtually impossible to convert manual test cases directly into automation tests. Manual tests are also much more comprehensive as usually they contain the entire process flow or a user story, whereas an automated test should only be a small separate function, or several functions tested in small pieces. Mainly due to the reason that if one function fails in an automated test, then the testing process stops there, and the rest of the features will not be processed through.
Manual testing also may not be as costly initially, as the manual testing does not require coding nor as much environment setup. Although automation testing itself, especially if only used to verify results after builds, does not require as much labour as manually testing everything similarly every single time. Usually it is impossible to acquire test coverage of 100% of the testable tests with only using automated testing. It is vital to keep in mind that manual and automation testing requires completely different approaches, and one does not exclude the other. (Sahla 18 September 2017.)

3 Types of testing methods

There are several different testing methods and not all directly relating to automation testing directly. For the purpose of understanding the test methods from the viewpoint of testing in general, some brief examples are provided in this chapter.

3.1 Regression testing

Regression testing takes place usually when for instance, a new update, a bug fix or a minor development solution has been integrated into the system or a previously tested portion has been changed. What needs to be tested is that all the parts that are being affected work as they did before, and any new updates or changes to the system under test did not break the previously working parts. In most cases, everything needs to be tested again to ensure that everything is working completely.

Since usually the previous test cases need to be repeated and tested, which probably already have passed in previous versions, it could prove to be a tedious task to test everything manually again. In these situations, automation testing is the solution as previous tests may be automated and run without as much resources and time as the manual regression testing may take.

Let's say that all test cases had been run in a previous cycle of a developed system, now when a completely new part of the system is being under development and new functionalities added that do not directly affect the already tested parts. Now these previously tested parts can be run quickly as automated versions of functionally the same tests, for which the end results are checked and verified that the results will not produce an error and function the same as in the first cycle for all the previously tested parts, while the other new parts will now be tested manually as before. This would then be categorized a basic example of regression testing in action, through verification. (Guru99 2017a.)

3.2 Performance testing

Performance testing is crucial especially for systems that are under heavy traffic often, and needs to be tested for the systems to handle the capacity of simultaneous users as well as possible. This would be practically impossible to test manually as it would be impossible to have that number of testers per project. Hence automation testing can solve

the issue as the automated scripts can be programmed for instance to generate users and to simulate traffic and by using i.e. Jmeter which is a load testing tool used to analyze and measure performance.

Performance testing has different testing techniques, such as “Load Testing” for understanding how a system performs under a certain amount of load. “Stress Testing” for limit capacity testing and determining what might happen when the system reaches its maximum capacity. “Soak Testing” or endurance testing for testing how the memory or other parameters perform under constant system load, and also “Spike Testing” to determine how a system handles sudden changes in its user capacity. (Tutorialspoint.com 2017e.)

3.3 Unit testing

Unit testing is part of Test Driven Development, which was described in an earlier chapter, and follows an agile and iterative method to verify the program code correctness for isolated sections through a repetitive process to ensure that as few errors as possible are being left out for following phases and are found as early as possible. Unit testing takes place during the development phase of the software, meaning when the application or software is being coded and is usually considered a part of the actual development process and thus performed by the programmers themselves (Software Testing Fundamentals 2017b). In unit testing the idea is to test separate parts of the program code and verify that the individual parts work. Since the required functionalities may be missing to perform accurate unit tests, the functionality may be instead simulated with the help of stubs or drivers. A unit test may be anything needed for the situation, these may be a certain line of the code, a whole method or a class. Unit testing is also part of the methodology called “Extreme programming”, which advocates the use of short development cycles through quick and easy integrations, and thus is fitting for the idea of testing small units at a time (Guru 99 2017b). (ISTQB Exam Certification 2017a.)

Another similar testing method to unit testing is called “Component Testing”, which may follow the actual unit testing. The difference between the two is that where unit testing is based on methodological testing and repetition of the testing cycle and done by the developers, component testing is done by the testers instead. The functional components are afterwards tested to work with other internal or external components during the integration testing phase that follows. (ISTQB Exam Certification 2017b.)

3.4 Integration testing

Integration testing refers to the phase which follows unit testing, where the integrated software components are tested to verify that they work together as a fully functioning group. Since the unit testing tests that individual units in the software work individually, the integration testing then tests that the units then can work together. The need for this is that even if the units themselves may not produce any errors, the compatibility between these components may not function initially or the interfaces used for the communication between the components may not have the right configurations or be compatible at all.

There are two types of ways to test integrations, the component integration testing tests that the interfaces and integrated components of the main system work together whereas the system integration testing tests that the interfaces to possible external interfaces and packages are working as well i.e. the connection to the internet or possible external data from another system. Figure 6 demonstrates two components where the testing consists of only the integrated parts.

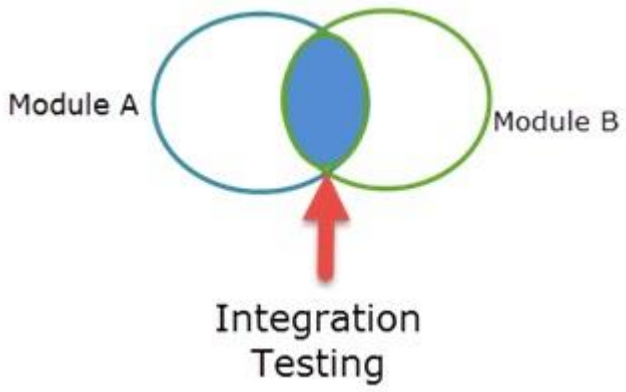

Figure 6 A visualization of the system integration testing where external components are combined (Guru99 2017c).

Integration testing has also multiple different approaches to the testing itself:

- Big Bang: Testing all components simultaneously as whole at once.
- Top Down: A top down cascading testing model where the top-level units or high-level components are tested first, and all following levels are tested in turns.
- Bottom Up: Similarly to the top down approach, but starting from the bottom up order. This may bring challenges as the top down levels may contain necessary configurations, hence some level of test drivers or stubs may be required to mimic the missing functionality of the top levels.
- Hybrid: The combination of both the top down and bottom up approaches.

These approaches contain both their benefits and faults. For instance, the “Big Bang” approach may limit the time for the integration for smaller systems, but makes it difficult to

pinpoint exact errors and thus be a high-risk method when considering multiple integrations. The incremental approaches “Top Down” and “Bottom Up” methods make it easier to detect errors more accurately and enable swift testing for individual levels, but make it impossible to map out the integrations as whole and since the functionality may be enacted through drivers the critical functionality may not be as accurate as hoped. (Software Testing Fundamentals 2017a.)

3.5 System testing

System testing phase basically is the phase when the entire system is tested as a whole, with all working integrated parts in accordance to the requirements set by the client. In actuality, the phase consists of multiple different testing methods used so that the system works in different types of usage scenarios. This definitely contributes to the fact that prior to the system testing, the majority of the test planning should have been done and in a level which covers all the aspects of the system that the functional specifications require according to the initial client requirements. The testing in this phase resembles somewhat of “Black Box Testing”, since there is no actual requirement to check the program code or any internal configurations of the software as the testing mainly focuses on the functionality itself. System testing is the most time-consuming part of the testing process, as the need to cover as many required functionalities may take time to implement different types of testing methods and to go through a substantial amount of runnable test cases depending on the scale of the system. It is also common for this stage to mainly consist of manual test runs for mapping out the processes required, and receiving feedback from actual use. For instance, the “Usability Testing” is also part of the system testing phase, where the ease and intuitiveness of use of the system in question can be measured through the user’s experience. Also, during system testing phase parts of the performance testing (i.e. Load Testing and Stress Testing), security testing, and even in between builds the regression testing, these testing methods may be executed mainly during the system testing for the need to cover the different parts of the system. Especially since the system should be mostly fully realized and functioning in all major areas of use. In large scale projects, the testing team is usually in charge of running the manual test cases and reporting results, but for small scale projects it may not be uncommon for the system testing and user acceptance testing to be combined, for instance in cases with limited testing resources, though this is not a desired scenario. (Guru99 2017d.)

3.6 Exploratory testing

Exploratory testing refers to the type of testing where the system or software is being used as one normally would and without any specific steps. One might also be navigating

through the software and stumble upon accidentally found defects, these types of tests may usually be part of the final user acceptance testing or just done by the business users of the company. The defect reporting in these cases might be more difficult than usual, as the steps beforehand might have gone unnoticed during software usage and reproducing the actual error will be more difficult again.
There are some tools to help with this process, for instance the “HP Sprinter” records every single click as a separate step from which test cases can be produced automatically, and the software also takes a screenshot and video from the last steps taken (Angerer, M. 2015). Downside to this is of course that the software requires a costly licence for “HP ALM” test tool to work at all.
Exploratory testing may not need specific test cases as the testing is done merely by exploring the environment and thus the test planning could be challenging, sole level of preparation would be recommended to know the processes to be tested. For instance, separate short one-step test cases can be created for each exploratory test run, if a test management tool is used in the project and the results are required for the reporting portion of the project. (Guru99 2017e.)

3.7 Smoke testing

Smoke testing or “Build Verification Testing” is a testing method to ensure that all major functionalities work accordingly. It is described as “non-exhaustive” testing with limited test cases for only the most necessary and critical functions that determine if it is possible to proceed with more detailed testing. Hence the name that originated from hardware testing, where a device was turned on for the first time and checked that the device would not catch fire or start to smoke. (Software Testing Fundamentals 2017c.)
This test phase can be automated for repetitive use, and to ensure a quick first glance after updates or builds. For example, if new builds are created frequently, an automated smoke testing suite may be configured to ensure that every build has not destroyed the functionality of the most critical parts of the system, and can be verified as soon as possible instead of manually mapping the situation. Especially since larger projects have more test cases, which may be run through the configured suite and be verified in minutes. (Functionize.com 2017.)

3.8 Security testing

Security testing falls under the category of non-functional testing, and the goal is to ensure that the system or software is free from any possible loopholes and weaknesses that may result into data loss or information breach to outsiders (ISTQB Exam Certification 2017c). These weaknesses and loopholes may either be intentional meaning a direct attack to the

system, or unintentional which means a system loophole where the sensitive information may not be as protected as it should. Through security testing, these actions may be detected early enough or the system can be verified to handle the possible attacks that may occur. There are six basic principles that should be always verified through security testing: confidentiality, integrity, authentication, authorization, availability and non-repudiation (Tutorialspoint.com 2017f).

There are several types of security testing, which contain the techniques to perform them. These can be for example:

- Vulnerability Scan: Using automated testing tools to go through the system in order to discover most known vulnerabilities quickly.
- Penetration testing: Simulating attacks from malicious hackers, idea is to exploit every vulnerability known from security weaknesses to accessing the features and data of the system in order to map out how the system handles these attacks. This can be either done manually or with an automated software. One example would be to perform an SQL Injection, where an access to the database can be achieved in order to perform administrative operations and even modifying and deleting data (Tutorialspoint.com 2017g). These attacks can be done either by white box testing where the tester is provided with system information, or by black box testing where no other information is known to the tester apart from the company name.
- Ethical Hacking: Attacking the system with the intent of exposing security flaws, instead of stealing actual information.
- Risk assessment: Defining all possible security risks either already known or that could occur and assigning them classifications for Low, Medium and High and addressing them according to the risk level.

These are a few examples of security testing types, as security testing is a very important and substantial part of software development. One way to tackle such broad scale of testing would be through test automation. For instance, there is a security testing framework called “BDD-Security”, which uses the Gherkin functionality to perform “Given-When-Then” syntax security tests i.e. for previously mentioned “SQL Injection” testing (Continuumsecurity.com 2017). Although both the non-functional and functional security tests may be implemented through test automation, to improve the coverage of tackling as many risks and as quickly as possible. (Guru99 2017f.)

3.9 User Acceptance Testing (UAT)

UAT or also known as Acceptance testing is the final phase in the software testing process. The purpose is for the client to verify that the finished system performs according to the set requirements formalized according to the client needs and validate that the system complies with the required business processes. This testing is performed by the actual end-users of the system, or client provided external testers who are familiar with how the system should perform. Preferably all major defects have been dealt with before even pro-

ceeding to this stage and the system should be ready for production stage, but if any defects are detected at this stage, a proper means of defect management and fixes should be discussed in order to attain client satisfaction to a degree.

UAT can be performed for instance as black box testing with users that are using the system as they would regularly and as type of usability testing where the user is using the system for the first time and tries to navigate and use the system without any knowledge of it. UAT phase can also be automated to a degree, especially in cases where the client lacks the resources to perform adequate acceptance testing or if the same simple tests only want to be verified to be correct quickly. It is important to still perform UAT mostly manually as user experience can't be replaced with automation in the long run. (Sharma, L. 29 January 2017.)

4 Tools and Frameworks

Tools are a vital part for both automated and manual testing. For automation testing there are several options for tools and frameworks that can be integrated to development environments and initiate the tests. For manual testing there are several tools to help manage test cases, defects and reporting, and some of which can be integrated with certain automation tools albeit only normally for commercially licensed large-scale software.

4.1 Test management tools

It can be tedious work to test without any documentation or tools to help with implementing the testing process. The key thing would be to have a place to store and create test cases, run the test cases, link requirements to the test cases, manage defects and provide reporting. There are options that can be utilized to help these functions as test management tools, and it can prove beneficial to be somewhat knowledgeable about how certain test tools can be utilized for manual testing in general.

4.1.1 Zephyr for Jira

Zephyr for Jira is a proprietary licensed test management tool plugin integrated with the software development tool Jira, which is mainly developed for more agile projects. There is a similar option for Jira to use another management tool plugin called "TestRail for Jira", which works through a cloud integration and connects with Jira for instance for defect trackability, depending on the preference (Atlassian Marketplace 2017b). Another tool, considered more of a plugin for Jira providing test management capabilities, would be Xray for Jira (Atlassian Marketplace 2017c.). All of these tools have the same underlying

goal, to provide test management opportunities for Jira, although all plugins contain different pricing, contents and usability.

Jira provides access for Kanban and Scrum boards which can be used for the project management aspect, and zephyr adds the full optimization for test management alongside it. It is possible for instance to create, manage and execute tests and plan the execution cycles related to them, along with linking possible defects and requirements as necessary. Compared to for instance another licensed test management tool HP ALM, Jira along with Zephyr can be classified into a more affordable category in terms of its pricing. Thus, Jira is very well known within the industry due to its variety of use. (Atlassian Marketplace 2017a.)

4.1.2 HP ALM

HP ALM refers to HP Application Lifecycle Management, which was formerly known as HP QC or Quality Center. The tool is a proprietary licensed software by HP and is usually utilized by larger software development companies, especially for old-fashioned waterfall application development projects since the various stakeholders are thought to be from developers all the way to product owners, providing a larger user base and integrated functionalities in one location. As the licensing fee is quite substantial, there would really be a dire need for a powerful test management software, however the tool does provide most required functions in one concise package (Kaul, N. 24 April 2017). It is possible to add and link requirements and to track the overall requirement coverage, create test cases and add the test cases into runnable test suites, manage the defects and link them to particular test runs, and provide comprehensive reporting methods to showcase the results, among other things. This is quite beneficial, as not all test management tools contain all possibilities and there may be need to use separate tools for requirements and defect reporting which then again makes it more difficult to track the coverage. (Guru99 2017g.)

4.1.3 Open-source management tools

There are some open-source testing tools if there is no reason nor funds to invest into a proprietary licensed software. For instance, there is an open-source agile test management tool called “Tarantula” which enables test design, testing and reporting options as required. Unfortunately, there will be no longer any future updates for “Tarantula” as of now as the development has been seized (Niittyvirta, A.). There is always somewhat of a risk when undertaking an open-source test management tool as a default method, since

the tools may not provide full utilization required for testing nor the option to integrate any other management tools. (Tarantula 2017.)

4.1.4 Excel-based Test Management

If for some reason open-source test management options are not desirable, then one option would be to utilize Excel sheets for test management and tracking. One can provide defect lists, test cases and requirements through Excel, but the most obvious downfall would be lack of coverage and tracking between the items. It would prove to be exhausting to track the correct defects to the test runs and requirements manually, but for a small scale project it could be a viable short-term option if there is no reason to use that much effort into comprehensive test management implementation. (Kaul, N. 8 March 2016.)

4.2 Examples of automation testing tools and frameworks

For test automation, the tool and selection of the tool is an important aspect of the development, since it is important to take into consideration the environment, the existing tools and capabilities of the developers, and any further preferences of use. Automation testing does require a tool for the entire functionality, so it is beneficial to know the differences of the usage for at least some of them.

What may be confusing is the difference between an automation testing framework and an automation testing tool as these are important to grasp if planning to utilize the concepts. The difference being that automation testing tools are used to implement the automation testing process in a tangible way, they are the platform for performing the automation testing itself and to create test cases and scripts. Frameworks are mostly meant as a set of guidelines and rules on how to create test cases and for the actual automation. The framework will also provide function libraries for plugins or applications, possible test data sources and any necessary modules and object details. A framework is therefore an addition to an automation testing tool, once which does not function properly without the underlying tool and vice versa. This chapter presents a few examples of regular testing tools, automation testing tools and automation testing frameworks which could be viable options for utilization depending on the availability, scale of the project and need. (Aebersold, K. 2017.)

4.2.1 Selenium WebDriver

Selenium is an open-source Web Automation Tool, used mainly for testing the user-interfaces of web sites. The tool provides a domain specific language called “Selenese” but it

is also compatible with most popular programming languages such as Java, C# and Python (Stewart, S. 2010). The Selenium WebDriver accepts programmed commands through a client API i.e. Java, which are then sent to the browser via a driver for a specific browser and returns the results of the communication between the HTML elements of a web page after the execution of the test. The WebDriver starts a controlled browser instance, which executes the test without a need for a separate server. Since Selenium WebDriver is mainly designed to use for browser based testing, it would be advisable to use other methods when needing to test any local or legacy applications. (SeleniumHQ 2017a.)
Additionally, there is another version of Selenium, known as Selenium IDE. It is implemented as a Firefox add-on which then records, edits and debugs tests directly on the browser. Through this, the scripts can be automatically recorded and edited, but are recorded in Selenese instead of another programming language for any browser actions. Selenium WebDriver will be used in the implementation of the case study in an upcoming chapter, where it's utilization will become more familiar. (SeleniumHQ 2017b.)

4.2.2 Cucumber

Cucumber is essentially a collaboration tool instead of a testing framework, which is a popular misconception among automation developers. Cucumber does facilitate the BDD process into automated testing through Gherkin language and because of that could be referred as more of a BDD framework instead. (Nicieja, K. 2016.)

Gherkin will be presented more thoroughly in an upcoming chapter, but as mentioned previously in the chapter about BDD, Cucumber provides the usage and integration of the Gherkin language and the "Given-When-Then" syntax. This can be implemented through an automation tool such as Selenium within an IDE environment, and the functional test case from the BDD syntax can then be linked to the programmed functionality. Thus, providing readability through development teams. Cucumber can be implemented with variety of tools and programming languages, but was initially developed with Ruby. Since then the implementations have included Java and C++ among others (Cucumber.io).

It could be possible to first write the Gherkin test cases by the business analysts, and afterwards developed into tangible automated tests and this is also where the usefulness of Cucumber becomes more prominent and also the danger if there is lack of understanding on how to formulate the Gherkin syntax into a form which makes sense for the developer to automate altogether. Especially if the usefulness of Cucumber is overshadowed by the lack of interest in use from the non-technical stakeholders (Cuadra, J. 2012).

For the purpose of demonstrating the use of BDD syntax, Cucumber has been selected as the BDD framework for the case study, which will be shown later on. There the utilization of Gherkin and the configuration of Cucumber will be elaborated more.

4.2.3 JBehave

Similarly to Cucumber, JBehave is another BDD framework using the Given-When-Then syntax. Both JBehave and Cucumber BDD frameworks are mainly utilized for acceptance test driven design, although usage can obviously be catered to the suitability of the situation. The main difference between the two is that JBehave was initially developed with Java, where Cucumber is developed with Ruby. Both also support the out of the box functionality for JUnit testing. Other than that, it would seem as Cucumber has gained more popularity in use especially since JBehave only supports stories which in Cucumber correspond to scenarios and are all gathered under Features. (JBehave 2015.)

4.2.4 Universal Functional Tester by HP

Universal Functional Tester, which was formerly known as QTP, is an automation testing tool created by HP and used mainly for system and local application testing in situations such as functional, regression or service testing.. The tool can be integrated with the HP Application Lifecycle Management (ALM for short) and formerly known as HP Quality Center (QC). While UFT provides a powerful tool for local and legacy system testing, the functionality does not translate well for browser application testing since UFT requires a local version on the used workstation. UFT is also heavily licensed and very expensive for companies to acquire, hence the tool is normally used by large-scale companies that usually have licenses for both UFT and ALM due to the cross-functionality of these testing tools. (Jain, A. 2017.)

4.2.5 Robot Framework

Robot framework is a generic automation testing framework for acceptance tests. It utilizes a tabular test data syntax which is based on the use of certain keywords. The framework provides Python or Java implemented libraries, where the users may create new keywords from the already existing ones. The framework was initially developed at Nokia Networks with Python programming language, and has since then become a popular automation framework sponsored by the “Robot Framework Foundation”. The software itself can be considered open source along with its most libraries and tools, even if it is widely used in test automation even by large-scale companies.
Robot framework does support Cucumber style BDD development along with the tabular functionality, although there are many ways to implement Robot Framework syntax. One

example being as provided from the Robot framework tutorials on how the actual syntax looks (Figure 7):

```
*** Settings ***
Library         OperatingSystem

*** Variables ***
${MESSAGE}      Hello, world!

*** Test Cases ***
My Test
    [Documentation]    Example test
    Log    ${MESSAGE}
    My Keyword    /tmp

Another Test
    Should Be Equal    ${MESSAGE}    Hello, world!

*** Keywords ***
My Keyword
    [Arguments]    ${path}
    Directory Should Exist    ${path}
```

Figure 7 An example of a tabular syntax for Robot Framework (Robot Framework User Guide Version 3.0.2. 2016.).

This syntax will be provided into formats such as plain text, HTML, tab-separated values (TSV), and reStructuredText (rest), which can be then edited in spreadsheet programs like Excel and also in text editors (Robot Framework User Guide Version 3.0.2. 2016.). All the data is provided in one large table on the file format, where the test data is recognized by the use of asterisks which contains the normal table name. The keywords within the syntax then can utilize all of the bundled libraries within the framework but also possible external libraries as well. Robot framework has been known to be paired with Selenium WebDriver as the automation testing tool for browser automation options. (Robot Framework.)

4.2.6 JMeter

JMeter is an Apache project used for load testing, which is an open source software written in Java. It analyses and measures the performance and endurance of web applications through automated testing, but nowadays it can also be utilized for other functions as well. JMeter tests the performance of both static and dynamic resources by simulating an excessive load on a single server, group of servers, networks and objects. It provides analytics on overall performance depending on the level of the load simulated, and tests the strength on how the application can handle sudden spikes of traffic. (The Apache Software Foundation 2017.)

4.2.7 JUnit

JUnit is an open source framework for unit testing to write repeatable tests. JUnit is heavily related to Test Driven Development, and has also been known as xUnit and SUnit. JUnit is imported as JAR-files, same way as Selenium and Cucumber, which are then added as external libraries to a corresponding project. JUnit utilizes “@”-annotations when using test methods, in order to invoke the methods in question into use. JUnit also requires test runner classes to run the tests and in order to see the results on the console afterwards. (junit-team/junit4 2017.)
Since JUnit is an out of the box feature for Cucumber, the annotations and test runner class will be demonstrated more comprehensively in the Case Study chapter. (JUnit Version 4.12. 2017.)

4.2.8 Jenkins

Jenkins is an open source, self-contained automation server developed in Java programming language and is focused on the continuous delivery approach (Jenkins). Jenkins is used to automate all kinds of functions that can relate to building, testing and the deployment of a software, and it can be also used to automate non-human functionality during the software development process. Jenkins is not actually a test automation tool nor a framework, but can instead be considered as a supporting tool for running automated tests. Jenkins can be set up to check any code changes taking place within environments like Git, do automatic builds with tools such as Maven, initiate tests, and provide automatic actions like production roll-backs or roll-forwards. Due to the described functionalities, Jenkins is an excellent candidate for launching automated nightly builds and tests with the configured Jenkins server. From the point-of-view of continuous delivery, nightly builds are a vital part of keeping an application clean and concise. It would be easier to launch heavy loading automated tests during night time, when there is no actual need for resources and any applications normally under use during daytime would not be affected (Berg, C. 2009). (Cloudbees.com 2017.)

5 Testing process

This chapter focuses on breaking down the basics of the testing process in general and how automation testing can be implemented in accordance to the test plan. The goal is to describe test planning itself and an overview on regular test tools and their use, mainly from the test design viewpoint as is.

5.1 Test Planning

Test planning forms the entire base for the actual testing process within the application or software development lifecycle, by skipping this part one would not be able to know the accurate amount of testable coverage, the risks involved and how to proceed with eventual regression testing. Due to this, most large-scale projects implement test planning right from the beginning of the project especially in agile iterative development, instead of focusing on testing only at the end of development. One reason also to start integrating the test automation planning from the very beginning, preferably as a project of its own, is to optimize its utilization as effortlessly as possible and to advance the start of the test automation use to as early as possible as the initialization will take its own time.

In the next sub-chapters, the very traditional and most common test planning process will be introduced. However, automation testing can have multiple ways of implementation and even very unconventional practises may be adopted for the planning itself and for constructing test cases. (Homès, B. & Homes, B. 2011. 228-229.)

5.1.1 Master Test Plan (MTP)

Master test plan is a comprehensive documentation describing testing consisting of the entire application and the corresponding project. MTP is usually formalized before the actual test planning takes place and should be used as reference for everything else being tested. (Montvelisky, J. 2008.)

5.1.2 Requirements

A requirement is a single functional and physical aspect of the system that is being documented from the needs of how the system or application should perform or have as a function. Most commonly these are specified together with the client, from which the functional design can be analysed and documented. The system or application will then be developed based on the functional design, but the testing itself will be planned and tested against the original requirements in order to be sure that the implementation has covered all the necessary client needs.

Requirements can be both functional and non-functional. Functional requirements usually focus on the behaviour of the system and what the stakeholders require from the system functionality. Non-functional requirements should cover everything else that the functional requirements won't, which usually relate to i.e. security, performance and the operation of the system. When planning automation testing usually the non-functional requirements are important, since they can be used to also define the need and criteria for automated performance and security testing.

The syntax of a requirement may vary, but constructing a single requirement into a use case form is a desired method, such as:

Functional requirement:
"As a user, I want to log into the system."

Non-Functional Requirement:
"System login must require both a username and a password."

This would enable easier analysis from which the functional and technical design may be documented as the focus is in one client need or i.e. a security requirement, and one requirement can easily be translated into one testable test case. (Eriksson, U. 5 April 2012.)

The risk of course lies with the discrepancies between the original requirement and the execution and whether testing is even able to be implemented based on what has been designed. Also in many cases a system may not even have any requirements to begin with, and in these cases the requirements are constructed just for testing purposes or the testing is planned based on client consultation and available documentation on the system or application. (Software Testing Studio 2017.)

5.1.3 Test Cases

Test cases are executable scripts to test a certain function of an application. They usually may be formalized to test one pre-determined requirement by creating a test case based on the requirement in question. Requirements are usually paired with a testable test case in order to track the coverage rate for testing, in the sense that if the test case passes, then the requirement in question should be fulfilled in terms of the application. If the test case fails however, then depending on the determined process, a defect about an error or other discrepancy may be created. Once the defect has been handled, the tester re-tests the test case and thus repeating the process.

In cases where there are several test cases, these could be then bundled into test sets containing all test cases of a certain process or function. Each test case may then contain several test executions that may be part of different test cycles and be run separately. This helps to track the progression of the development, or if something was broken that worked in a previous test cycle in a different test execution. The bundling and running test cases in separate cycles will ultimately assist the reporting needs and traceability of the project progression and even enable the creation of best practices and checklists for known issues.

The nature of the test cases may depend on the situation, but most manual testing for example is done by using the test case with basis for all actions. Due to this, a good test case consists of several steps for specific testable actions, with preferred expected results resulting from those actions.

For instance, when having a requirement *"As a user, I want to log into the system."*, an example of a test case being created from this could be as follows:

Step ID	Description	Expected result
Step 1	Launch <application name>.	The right application should open.
Step 2	Click the link to login page.	The login page should open.
Step 3	User enters <User name>.	User name is entered.
Step 4	User enters <Password>.	Password is entered.
Step 5	Click Login-button.	The user can login to <application name> successfully.

Here the brackets represent the input values determined for each test case. There is a possibility to create generic test cases, or some test tool applications allow parameters to be used as placeholders for the input values that can be changed by modifying fields separately within instances rather than the test case itself, which may be used by multiple projects at once.
There is no specific way to formulate a test case, whatever works for the situation is the method to be used. Most test tool applications may be expensive, and for smaller development projects Excel could be the only solution as mentioned in earlier chapters. Thus, enabling the tester to be creative during the test case creation period. Although, an Excel sheet may be difficult to use when tracking requirement coverage through test cases.
Once the test case has been created, the test execution itself takes place. (Bartlett, J. 2 December 2015.)

5.1.4 Test Execution

Once the planning has been sufficient, the actual execution of testing will begin. Test execution mainly entails the tester to perform beforehand planned test cases from the system under development. The tests may entail the tester to try user log in or other specified tasks that the client requirements consist of. If these requirements are met through the performed test cases, then the result of the test case will be passed. If the test does not meet the conditions, then however the result of the test will be deemed as failed.

Passed test cases will usually not require any further procedures except during the regression testing phase, when being run again, but failed tests need to be reported as quickly as possible to ensure swift fixing of these errors.

Once the errors have been fixed, the tester should re-test the same test case to ensure that the conditions are met, and the test case can be deemed as passed. If this does not happen, the process will be repeated until the fixes have been implemented correctly on between reporting and status updates. (Tutorialspoint.com 2017h.)

5.1.5 Defect Management

Defect management refers to the reporting of found errors during testing process, whether it be through test runs or free form exploratory testing. The main goal is to deliver a high-quality software and discover any anomalies that may disrupt this. The reporting itself may differ from the situation or comply with the desired method, but the best practise would be to be as precise as possible and preferable document the steps for how to reproduce the error and hopefully also a descriptive screen shot or video of the situation.

In most test tools the defect management feature has been integrated for ease of management, but as long as the error handling takes place and is being corrected the tester may even use an excel sheets for tracking purposes depending on the scale of the project or phase in testing. For projects with the need to handle multitude of defects, a definite recommendation would be to use an actual issue handling tool that can i.e. link the failed test case to the defect, enable the tester to give a comprehensive description on the error, maybe even inform the developer via email notification on the found defect and in turn notify the tester when the defect can be re-tested again. Few examples for this could be HP ALM/QC and Jira with a test management plugin.

Once the error has been appropriately handled, the defect will be reassigned back to the tester, who will verify how well the defect has been fixed. If however the defect still has not

been fixed, the process will be repeated until the required function is deemed fixed and functional. (Homès, B. & Homes, B. 2011. 257-259)

5.2 The importance of the planning and preparation of test tools

There are many varieties of testing tools to be used depending on the situation and project, as presented in the previous chapter about testing tools, which range from open source free-to-use applications to licensed corporate software, all the way to simplified excel sheets when considering test management in general. For automation testing, the varieties differ similarly, except some licensed products also integrate between test management and automation testing tools.

From the tool implementation point of view, that part of the decision to select the required management tools for manual and automation testing is part of the test design process. Different tools cater to different needs, if one is to understand the diversity and power different tools can provide, then the benefit will be much more prominent when fully utilized. For this particular reason, there may be even need for projects to map out the functionality and usage of proprietary tools versus open source, or even a requirement for separate tool specialists especially when planning for a large-scale project where the need to handle thousands of requirements and test cases is required from the test management tool. (Koomen, T., van der Aalst, L., Broekman, B. & Vroon, M. 2014. 385-387)

5.3 Implementation of languages from automation testing perspective

In order to implement automation testing in a functional way, a rigorous planning of programmed scripts is in order. After all automation testing project is considered as a development project simultaneously. In order to invoke the functionality of the executable test cases on a software, the tools used themselves are not enough as there is a need to understand programming logic and programming in general to achieve automation at all. For instance, an IDE such as Eclipse for Java can be used as the development environment for the code, which then has the tool libraries added within the project.
Eclipse IDE can be just one example of an environment used to provide the code with Java for the automated script to work. The same could be done i.e. with Microsoft Visual Studio and the C# programming language. It is always important to remember the compatibility of the programming language and the tools being used, since not all configurations necessarily go together. But along with just the programming languages, there is also supporting language for BDD framework, from which Gherkin will be used as an example.

5.3.1 Gherkin

Gherkin is not directly a programmable language, since the idea of Gherkin is to be a "Business Readable, Domain Specific Language" or BRDS for short that is part of the BDD approach as mentioned during the earlier chapters. This means that the Gherkin test case is written by detailing the software behaviour in certain keywords and sentences that the automated software code can understand, and provide the functionality from java code snippets.
The purpose of Gherkin is both the documentation and the automated tests themselves, and the Gherkin grammar consists of several different spoken languages, not limited to just English language.

The Gherkin syntax consists of Feature, Scenario, and Given-When-And-Then keyword usage. The Gherkin source file always needs the extension ".feature" to its filename and the test case begins with the word "Feature" and an optional description. Afterwards the situations will be described as scenarios, with each different scenario having its own Given-When-Then conditions.
An easy example of the syntax might look as follows:

Feature: Description of the situation to be tested or additional information needed.

Scenario: Business situation 1
Given Precondition
And Additional precondition
When Action of the user
And Other action
Then Final outcome

Scenario: Business situation 2
Given Precondition
When Action of the user
Then Final outcome

This would be a basic example of the Gherkin syntax, but there is also a possibility to add parameters through scenario outlines (cucumber/cucumber 2017). This would be accomplished by adding an "Examples"- table below the entire "Feature" with highlighted keywords. An example of a Gherkin feature file with the examples-table can be seen from below as follows (Figure 8):

```
Feature: Web site Login
  As a user, I want to login to the web site.

  Scenario Outline: Login
    Given User is on the home page
    And User navigates to the login page
    When User adds "<UserName>" and "<Password>" is entered
    Then User clicks on the submit button

    Examples: Animals
      | UserName | Password      |
      | tester1  | test1234      |
      | tester2  | test5678      |
```

Figure 8 An example of a Gherkin syntax with the usage of examples (Knight, A. 27 January 2017).

Usage of Gherkin parameters, brings reusability for testing with Cucumber and may even offer multiple testing scenarios at once through scenario outlines. The parameters added to the Gherkin syntax can stay the same, and by adding or modifying the examples-table the test data should be implementable through different situations. It is also possible to have multiple examples-tables, just as long the referenced parameters can be found from the Gherkin itself.

Once the Gherkin feature file has been finalized, the highlighted keywords will then be referenced with “@”-annotations in the executable code for the corresponding keyword. Gherkin then recognizes which steps have been run and the result of the test itself. The power of the use of these types of test cases lies in the possibility for the whole development team, including even the non-technical personnel, to understand what is being tested and what went wrong in the results. There also lies the risk of not implementing the test cases as fully intended, especially if formulated by a non-technical person who does not understand coding functionalities. This could lead into miscommunication on results, but the potential does outweigh the possibility of confusion. Gherkin is used as an example of the BDD syntax, since it has been selected along with Cucumber for the implementation of the Case Study. (Singh, V. 2014.)

5.4 Deliverables and understanding the results

Testing is essentially the tracking and validation of the quality of the software to the project stakeholders. The results need to be conveyed as concisely and comprehensively as possible, and this is where the importance of reporting comes along as the results of the testing process are the deliverables that can be attained. The test status reporting can happen in between and at the end of the development, the idea of the reporting is always only to advise the project management and the leading stakeholders on the situation

based on the test results, from which the management may make decisions on where to move forward. Testers should not make these kinds of important decisions themselves, as this could lead to miscommunication of expectations. Testing can therefore be considered as more of an advisory method, conveyed through the reported results. (ISTQB Exam Certification 2017d.)

What constitutes as reporting from the testing perspective? The reporting can vary depending on the scale and the scope. For instance, the reporting could consist of the test coverage or more precisely the percentage and ratio of passed, failed and not covered test cases, the percentage of open and closed defects based on risk level, and the reporting of any obstacles from the testing point-of-view. Many test tools provide the necessary metrics and report exports that may come in forms of different charts, visualized pictures, colour coding, and traceability reports. Or if such measures would not be attainable, then these charts could be created manually with i.e. Excel sheets. For example, one example of reporting could be as follows when reporting the status of test cases using Excel (Figure 9):

Summary	
Test Case total	18
Run	15
Passed	11
Failed	4
Not Run	3

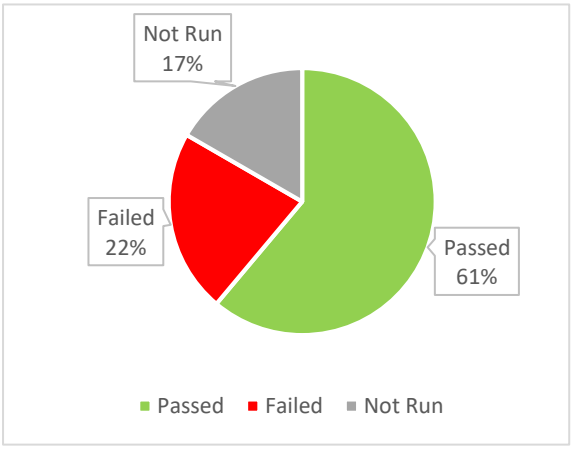

Figure 9 One example of test reporting using Excel.

Or when using a reporting tool, the ready provided reporting options might look as seen below in the example of the TestRail Management tool provided by their official web site (Figure 10):

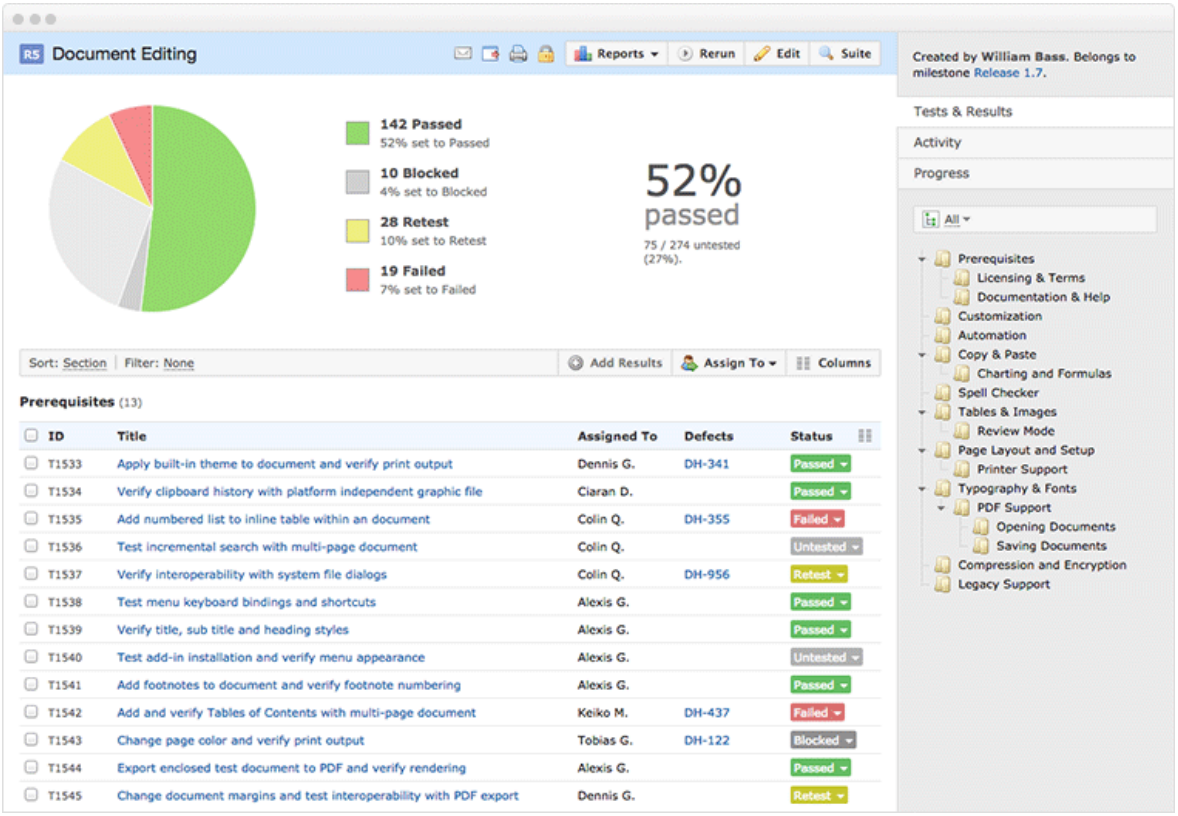

Figure 10 TestRail management tool test run overview (TestRail 2017).

As seen on the figure 10, the chart tracks the overall status of the test cases, which can then be generated into more comprehensive project reports. Test management tools usually also provide reporting templates to be used when presenting the findings to the stakeholders, who can then assess the situation and decide more accurately on the changes to the project timing, scope and the entire production readiness of the developed software. (Software Testing Help 2017.)

6 Risks of automation testing

This chapter revolves around the possible risks that automation testing may contain. As the method itself ultimately introduces a way to test multiple test cases without as many resources, one should remember that incomplete preparations, lack of maintenance, incompatible solutions and platforms may not provide as reliable results as one would hope.

The biggest factor when it comes to risks, is to manage the expectations of the managers and other stakeholders. Too often there are misconceptions on understanding the basis of test automation, just because there are now automated tests, does not necessarily mean that the tests can find bugs more easily or that everything can be tested with 100% coverage. More often automation testing is used just to verify that the basic software is functional, and that the previously tested results stay the same, instead of finding any new issues or bugs. Also, as not everything can be automated, there always should be something left out for the testers to verify the correct functionality, usability or to see any issues more profoundly. As manual and automated test cases are not exactly the same, the manual test cases can't directly be copied as automated ones, or the outcome of the results and the implementation won't be accurate.

Another misconception is to disregard the fact that the implementation of automation testing requires an actual software development project with the design, testing, deployment and maintenance in order to upkeep the correctness of the results and the whole process should not be rushed as lack of time might cause issues in the long run. In these situations, it would be obvious that if the automation testing implementation would not be designed nor maintained after the deployment, any new builds or versions would change the programmed logic of the automated test cases/scripts and even change the results. This could happen maybe even unbeknownst to others, especially if the results would be returned as falsely passed or falsely failed, thus, slowing down the detection rate of any possible new issues. It takes a lot of effort and upkeep to update everything to ensure that the results would not be false due to simple updating discrepancy. (Sahla 18 September 2017.)

Another risk would be not having skilful or the right people to implement the programmed code of the designed implementation, having team members lacking the knowledge of automation testing and the tools entirely, or completely disregarding any training opportunities for potential team members. This could lead to incompatibility issues with the environments, programming languages, used frameworks and any automation software tools, that may not be right for the tested software as the background knowledge would not be

there to identify the potential risks of using incompatible tools. For instance, let's say that the BDD framework scripts were pre-designed by the business analysts with only the knowledge on the business processes but not on the functional side or on what has been implemented, and because of this the design of the BDD test cases could end up with illogical conditions that the developers would not be able to implement as automated test scripts. Another scenario based on the risk of lack of knowledge might be that the implementation design could state that the backend systems would be readied to be tested with a browser automation tool such as Selenium, or that the HTML pages would be tested with a tool used for local applications or systems i.e. HP UFT. Hence, these scenarios resulting in failures to automate properly if the stakeholders would not be knowledgeable of the right use of these tools. (Software Testing Genius.)

7 Case study

The purpose of the case study is to show case on how to implement test automation in a possible real-life testing scenario. For the testing itself, the tools and frameworks selected for the implementation are selected as follows:

- Eclipse for Java developers with a Cucumber plugin.
- Selenium Web Driver automation tool.
- Cucumber as a BDD framework with Gherkin

The case study will be conducted on a freely distributed and simulated travel agency page, that does not have any real-life functionality for precautionary measures.
The idea is to demonstrate a one example on how to automate a login and logout test process, on a website with a ready-made registered test user. The home page that was used for the demonstration of the case study is visible below in figure 11:

Figure 11 Mercury Tours store front (Mercury Interactive 2005).

The process was implemented with the instructions and help from the “Tools QA” Website under “Cucumber Tutorial” by Lakshay Sharma (Sharma, L. 2014a). The downloaded

files, framework jars and some lines of code were kept almost the same to ensure the functionality and quality of the demo, but the entire process was implemented on a different website and different browser to demonstrate the functionality and the implementation, more so than to teach how to program new functionalities.

7.1 Installing tools and frameworks

What is needed in the beginning is to download all required tools, frameworks and libraries to ensure that the automation test build can function once being set up.
What is required altogether are in terms of tools and frameworks are:
Eclipse for the storing the project and programming scripts with Java programming language, Selenium Webdriver tool and its libraries that contain the knowledge to start and control the browser instance during test run, and Cucumber testing framework that implements the use of Gherkin language to write the automated test cases which are written in business readable language and will prompt the corresponding functionalities of the software to be tested. More about separate plugin downloads related to the frameworks and how this will be implemented in practice will be presented in the following chapters.

7.1.1 Installing Java, Eclipse and the Cucumber plugin

To start, one should download the latest version of Java SE, meaning the newest Java development kit (JDK), to the system from the ORACLE website by following the downloading instructions. This is so that the system can understand the Java programming language and run Java applications. The JDK is obviously needed for coding the functionality of the test scripts, so that the programmed code guides the automated tests to completion. The JDK version which was used for the purpose of the case study was Java SE 9.0.1. (Sharma, L 2015b.)

Next what is needed is to download the latest version of Eclipse IDE for Java Developers (version used was Eclipse Oxygen 64-bit), an environment for the development of the automation test projects and containing the project itself. To do this, the user needs to download the correct version corresponding to the operating system at hand from the official Eclipse website, save the .zip file to computer and once extracted, start the downloading from the eclipse.exe installation file. (Sharma, L 2015c.)

Now that Eclipse is ready to be used, a handy “Cucumber Eclipse Plugin” can be installed which helps the Eclipse environment to understand the Gherkin syntax better. The plugin is not a mandatory part of the project, but can help immensely to understand which parts

of the syntax are vital as the plugin also acts as a highlighter for the keywords and improves readability.
The plugin can be downloaded by navigating to the “Help” menu from the upper toolbar and selecting the option to “Install New Software”. From the prompted window the option “Add” should be selected, which prompts another window for inputting a web address. The website called “Tools QA”, which I used as the basis to implement my case study, supplied the following address for the plugin location:
“*http://cucumber.github.com/cucumber-eclipse/update-site*”
By adding this address after clicking “OK”, the “Cucumber Eclipse Plugin” option will be visible. Check the box next to it before clicking next, after agreeing to the usage terms the downloading will be prompted on screen and once downloaded will be used when writing the Gherkin scripts. (Sharma, L. 2014d.)

7.1.2 Downloading Cucumber and Selenium WebDriver

Cucumber is downloaded in several separate modules which are called jars. Each jar is a functional library that supports the functions of Cucumber and contains the program itself. According to the official Cucumber website, there is no separate setup installer to install the framework as only the required jar files are downloaded and implemented to the project manually. The required jar files were downloaded from the public maven repository for which the link was found from the official Cucumber Website and from the online maven repository website “https://search.maven.org” by searching the repository by the accurate jar names and with Java-language support and downloading them to an easily accessible location on the local drive so that the downloaded files can later be added to the project on Eclipse.
According to the “Tools QA” example, the required cucumber jar files that enable the functionality to work and which should be downloaded for the project were as follows:

Jar files relating to Cucumber itself:

- cucumber-core
- cucumber-java
- cucumber-junit
- cucumber-jvm-deps
- cucumber-reporting
- gherkin

Additional jars not exclusively affiliated to Cucumber:

- junit
- mockito-all
- cobertura

The jar files relating to Cucumber itself can be found from “http://repo1.maven.org/maven2/info/cukes/”.

The jar files for cucumber enable the framework to function as required with the addition of using the gherkin language from the gherkin jar file. The other jar files cover additional requirements for the testing. JUnit enables a visual representation of passed and failed tests after test runs by showing the failed tests in red and passed in green. Mockito is a mocking framework to create mock or test double objects to be tested in BDD testing (Wikipedia 2017). Cobertura on the other hand “Calculates the percentage of code accessed by tests” according to the official website (Cobertura 2.1.1.). (Sharma, L. 2014e.)

The Selenium WebDriver can be installed as a zip-file from the Selenium’s Download site. The version to be downloaded should support Java, and once extracted to the local disk the extracted folder should contain a general jar-file for the client, and a library folder for several other jar files. Similar to cucumber jar files, these will also be added manually to the Eclipse project as needed. The version used in the project was the latest at the time, Selenium WebDriver Java version 3.5.3. (Sharma, L. 2015f.)

7.2 Configuration of selected tools and frameworks

Before any scripts can be written and tests can be run, all of the downloaded tools and frameworks need to be configured for the project itself.

At first the Eclipse Oxygen client needs to be started. Every time the client is launched, a separate inquiry about the workspace is prompted on the screen unless the selection is selected as default. It is easy to select a location for the projects that can be easily accessible, although this is not as important since the location is set as the Eclipse program folder.

From the “Welcome” page, create a new project. This can be done by selecting:
File > New > Java Project
This action prompts a new window on the screen, and once the Project name has been added by clicking on “Finish” button the new project is ready to be configured.

What is needed next is to add the jars to the project that were downloaded in the previous chapter in order to add the associations to the project for the tool and framework. Starting off by adding the Selenium jars, by right-clicking the project name root folder and selecting “Properties” a window will be prompted on screen. From the window select “Java Build Path” on the right side of the window and then navigate to the “Libraries” tab. From here

click on the “Add External JARs…” button on the right side and navigate to the extracted folder for Selenium jars. First add the “Executable Jar File” for the client jar file found in the root folder of the extracted Selenium folder, and afterwards add all other jar files in the “lib” file and click the “Apply and Close” button. This ensures that all required Selenium jar files containing the functionalities are associated on the project. Below in figure 12 is a picture of the “Properties” window.

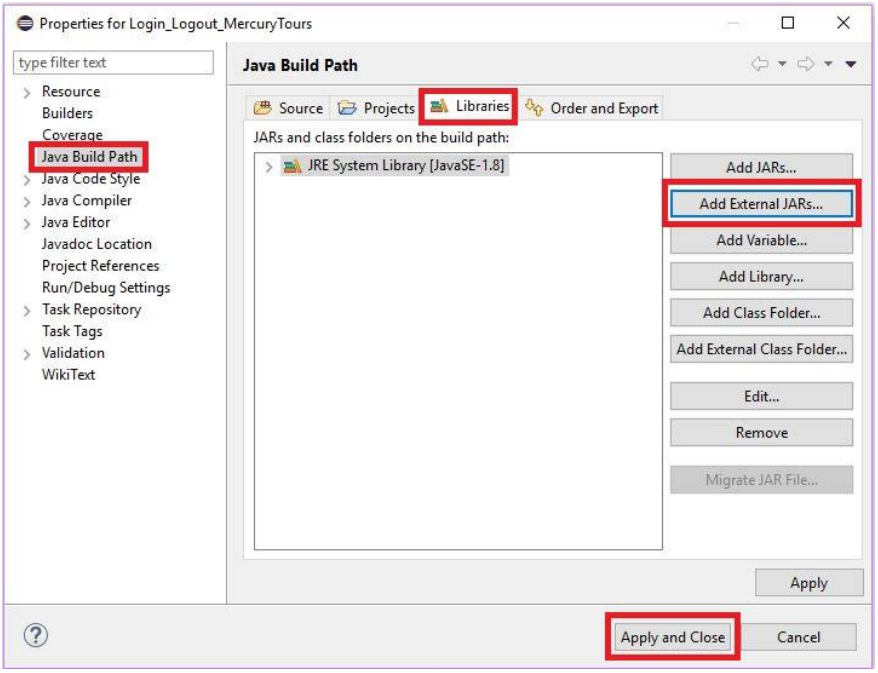

Figure 12 Adding Jar files for both Selenium WebDriver and Cucumber, all relevant selections have been highlighted in red.

After the Selenium jar files have been added, repeat the process for all jar files that were downloaded for Cucumber. At this stage the project should be adequately configured and ready to start with the process of writing the tests. The project should look as follows on the Eclipse “Package Explorer” (Figure 13):

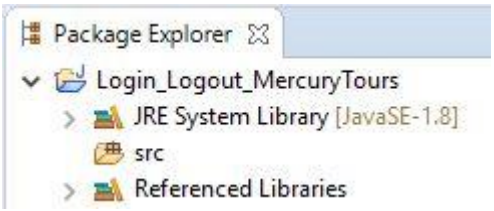

Figure 13 Project folder after configuration

The “Referenced Libraries” menu should now contain all previously added jar files and the next phase may begin. (Sharma, L. 2014g.)

7.3 Creating and running the automated test on the Mercury Tours web site

Before any writing of the test cases can start, it is beneficial to create a folder structure for the project. Starting by right clicking the default “src” file and adding a package file from selections “New” > “Package” in the Package Explorer. This prompts a “New Java Package” window, where a name for the package can be added and it is advised to keep the source folder the same. Give the package file a suitable name, in this instance i.e. “testCucumber”, and click “Finish”. After this, add another package similarly to the same “src” folder and name it i.e. “stepDefinition”.

Add a folder under the project structure by right clicking the project name and selecting “New” > “Folder”. Give the folder a name, i.e. in this case the name is “Feature”, and click “Finish”. (Sharma, L. 2014h.)

The project should look something like this in the package explorer (Figure 14):

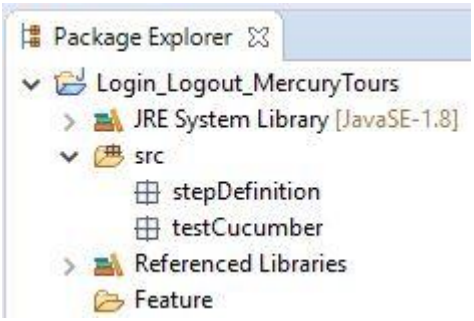

Figure 14 How the project should look after adding the file structure.

7.3.1 Creating a Selenium Java Test

To start the process, create a Java test script in Selenium for the Login and Logout process to see the functionality in action. After the code has been set to work, the same process will be converted into Cucumber scripts, where the use of Gherkin enables to understand how the test can function without trying to decipher the code too vigorously.

First add a class file to the recently created “testCucumber” package and give it a name i.e. “TestSelenium”. This can be done by right clicking the package and selecting “New” > “Class”. Remember to check the radio button for the option “public static void main” before clicking on the “Finish” button. The selection screen can be seen below (Figure 15):

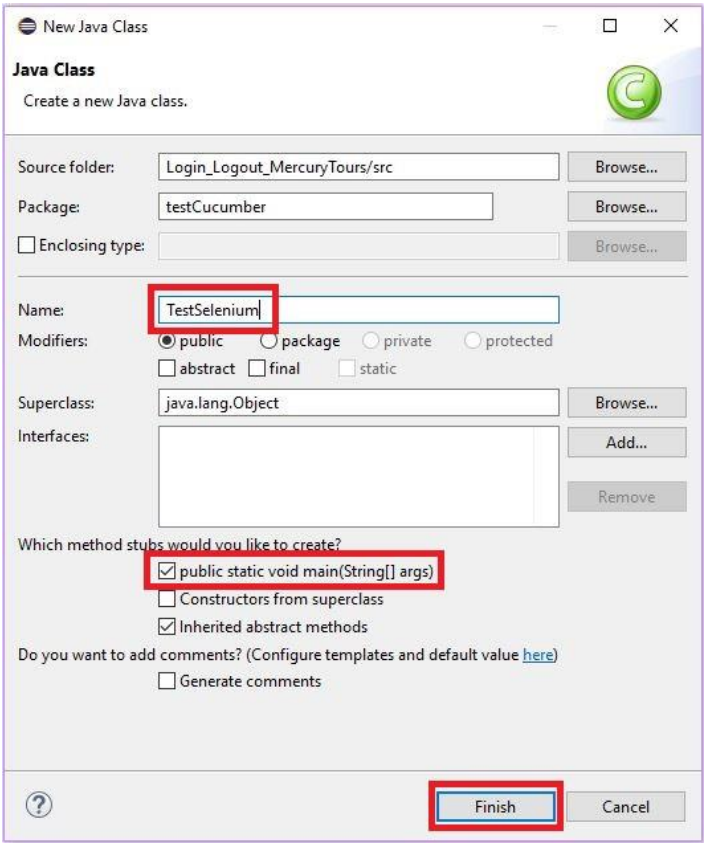

Figure 15 The selections when adding the Java Class.

The project should look like this at this point (Figure 16):

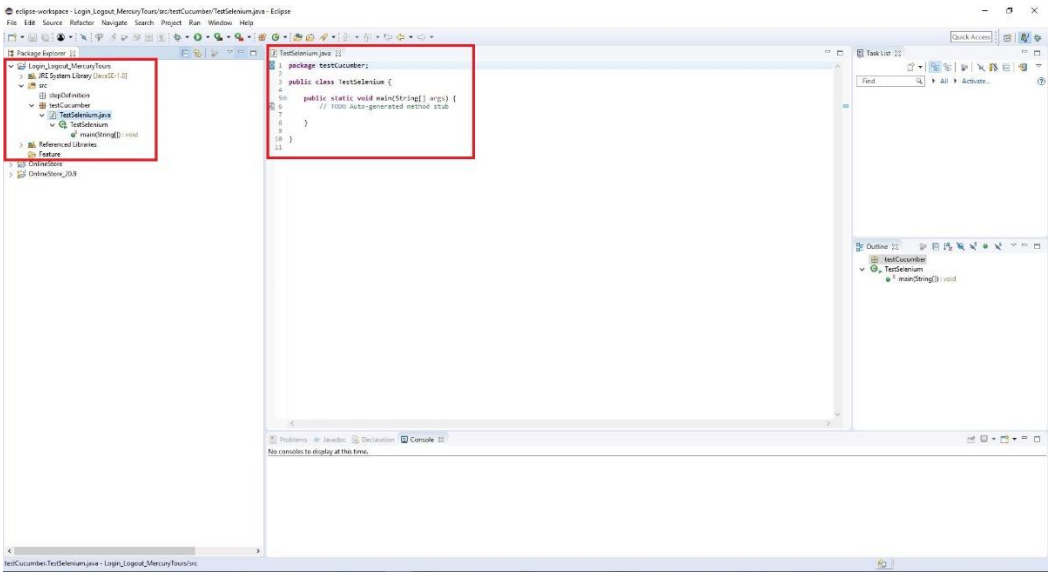

Figure 16 The project structure and outlook after adding a class, which is be highlighted in red.

Add the following code snippet to the recently created class in its entirety, although the green text is meant for commenting the functions of the code. The base for the code was taken from the tutorial of the “ToolsQA” website and adapted to work on the current project (Figure 17):

```
package testCucumber;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class TestSelenium {
   public static void main(String[] args) {
        // Create a new instance for the browser driver, it is important to add full path for
        the Chromedriver from the local folder downloaded on computer.

        WebDriver driver;
        System.setProperty("webdriver.chrome.driver", "C:\\Full driver folder
        path\\chromedriver.exe");
        driver = new ChromeDriver();

        //An Implicit wait, which is the amount of time that the search for the elements on the
        page can take before throwing an exception.

        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

        //Launch the Mercury Tours Web site

        driver.get("http://newtours.demoaut.com/");

        // Find the element "SIGN-ON" and click the sign-on page

        driver.findElement(By.linkText("SIGN-ON")).click();

        // Enter the user name on the element found

        driver.findElement(By.name("userName")).sendKeys("tester_1");

        // Enter the password on the found element

        driver.findElement(By.name("password")).sendKeys("test1234");

        // Presses the "Submit" button. The WebDriver will redirect the user to the home screen.

        driver.findElement(By.name("login")).click();

        // Print a Log In message to the console screen

        System.out.println("Login Successful");

        // Click the corresponding element for the log off button and logs off the user

        driver.findElement(By.linkText("SIGN-OFF")).click();

        // Print a Log In message to the console screen

        System.out.println("Logout Successful");

        // Closing the driver

        driver.quit();
    }
}
```

Figure 17 The initial code for the functionality of the automated test (Sharma, L. 2014h).

The code above should work in the way that the browser is launched to the home page, the Login link is clicked automatically and the username and password are inserted as values. The Submit button is clicked, and a message about successful login is printed out

on the Eclipse console screen. Afterwards the user should be logged out automatically, and the browser should close itself.

In order for the browser to be prompted on screen automatically, the appropriate driver needs to be downloaded separately and referenced on the code as a web driver. In this case the browser used was Chrome, which requires a ChromeDriver to work. The driver can be downloaded as an open source google API by searching with the name. The version used in this case study was ChromeDriver 2.33. (Google Sites.)
Similarly, if Mozilla Firefox was required for testing, the ChromeDriver would be replaced with GeckoDriver and downloaded and referenced in a similar way. (mozilla/geckodriver 2017.)

At this point the test code is run as a Java application by selecting:

“Run” > “Run As” > “Java Application”

Or by right clicking the “TestSelenium” Java file from the “testCucumber” package and selecting:

“Run As” > “Java Application”

If the code works as it is supposed to and the automated process was prompted on screen, then in the next subchapter the same process is converted to a Cucumber feature file with Gherkin language. (Sharma, L. 2014h.)

7.3.2 Creating a Cucumber Feature file

Previously on chapter 3.3.2 a short introduction to Gherkin language was given. In this chapter the knowledge of the keywords will be utilized and used to explain functionally how the language connects with the Java code and works in action.

At first, a feature file should be created within the previously created “Feature” folder in the package explorer. When creating a feature file, it is important to remember that Cucumber will not be able to detect the added features on the file unless the file has the “.feature” file extension added to the name and usually a single feature file has a single feature included for functional understandability. For instance, the name that will be added in this case study is “Login_MercuryTours.feature”.

Begin by right clicking the “Feature” folder in the package explorer and select “New” > “File”. Make sure that the selected folder is still correct and add the appropriate file name containing the required file extension and click “Finish”.
Add a simple Gherkin BDD script to the newly created file that defines what is to be tested in the process, following this simple example (Figure 18):

```
Feature: Login/Logout

Scenario: Successful Login as an existing user
            Given User is on Mercury Tours home page
            When User clicks the SIGN-ON button
            And User enters username and password
            Then Message is displayed Login Successful

Scenario: Successful Logout after Login
            When User Logs out from the application
            Then Message is displayed Logout Successful
```

Figure 18 The Gherkin BDD script used in the case study (Sharma, L. 2015i).

The Gherkin test case gives an insight on the functionality of the tested process and every single Given/When/And/Then line will have a corresponding annotation on the code file that will be created a bit later. When the code snippet starts with “@Given” format, the code recognizes the phase from the Gherkin scenario and shows it as passed or failed in the final reporting screen. If there is no linkage between the annotated piece of code and the Gherkin keyword, then the test run phase will return an error as the coverage would not be sufficient enough. As the Login information was added to the code directly, this time there is no need to implement parameters through examples-table as explained in the Gherkin chapter.

From the point of view of the base tutorial from “ToolsQA”, it would be advisable to download the “Natural Eclipse Editor” for Gherkin, which can be downloaded from the Eclipse Marketplace. What this addition does is to add a set of plugins that help the editing and maintaining of the Gherkin and other BDD files.

In order to download the suggested plugin is to select:

“Help” from the upper toolbar > “Eclipse Marketplace”.

Search with the keyword “Natural” and click “Install” to install the plugin, confirm “Cucumber Editor” as the BDD framework of choice and click “Confirm”. After this, accept the user

terms and click “Finish”. The version of the “Natural” used for the case study was “Natural 0.7.6”.

At this point, there is a need to create a new class for running the tests. JUnit framework was previously downloaded alongside with the Cucumber jars, and as Cucumber uses JUnit to run the tests and showcase the results, there is a requirement for a new separate “Test Runner” class. The class contains the glue code between the “.feature” file, the JUnit framework, and an upcoming code file for the corresponding Gherkin keywords, which was mentioned before as the annotated code snippets. (Sharma, L. 2015i.)

Create the class by right clicking the “testCucumber” package and name the class as “TestRunner” as per previous instructions, only this time leave the “public static void main” radio button unchecked.

The TestRunner class was taken directly from the “ToolsQA” website since the code snippet has an important role in the process, but the class itself is not long with lines of code. The example utilized in the case study should look something like this (Figure 19):

```
1  package testCucumber;
2
3  import org.junit.runner.RunWith;
4  import cucumber.api.CucumberOptions;
5  import cucumber.api.junit.Cucumber;
6
7  @RunWith(Cucumber.class)
8  @CucumberOptions(
9              features = "Feature"
10             ,glue={"stepDefinition"}
11                         )
12
13 public class TestRunner {
14
15 }
```

Figure 19 The class for the glue code used by JUnit (Sharma, L. 2015j).

According to the “ToolsQA” web site, the import statement on line 3 imports the “@RunWith” annotation from the actual JUnit class, while the annotation itself communicates the JUnit to run tests with the Cucumber class that is included in the “Cucumber.api.junit” package. Also, the import statement between lines 3 and 4 called “CucumberOptions” imports the same named annotation “@CucumberOptions”, which communicates with Cucumber to look for the feature files created. In addition to this the annotation communicates also the necessary reporting functions, usually on how to visualize the pass/fail scenario once the test is completed.

Now the Cucumber test can be run for the first time, even though the Java code and the Gherkin has not been associated or created. This step allows Cucumber to make suggestions for the syntax and creates the template where the code can be added easily to the suggested methods.
To do this, click either the “Run” button on Eclipse or right click the test runner class and navigate to “Run As” > “Junit Test”.

The window after the execution should look similar to this (Figure 20):

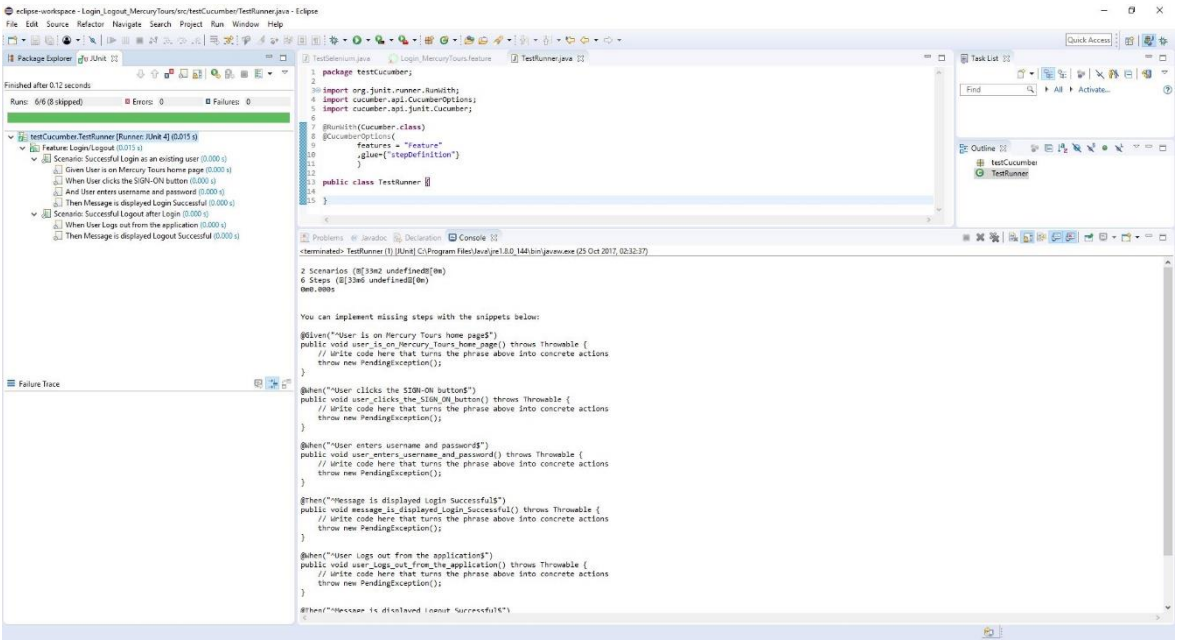

Figure 20 After Junit is run the first time and the initial Gherkin script has been added to the ".feature" file.

What is visible on the picture is that on the right side, the Junit has passed the test since the existing keywords have been found from the “.feature” file. However, since there is no code present yet that has been added to this section, Cucumber automatically suggests on the Console tab a syntax for code that corresponds to the existing Gherkin keywords that can be copied to the class that will be created to the “stepsDefinition” package. In case of possible errors, it is possible that the ”cucumber-java” version and the current Java version may not be compatible together after new updates have taken place. (Sharma, L. 2015j).

The project should look like this at this stage in the Package Explorer (Figure 21):

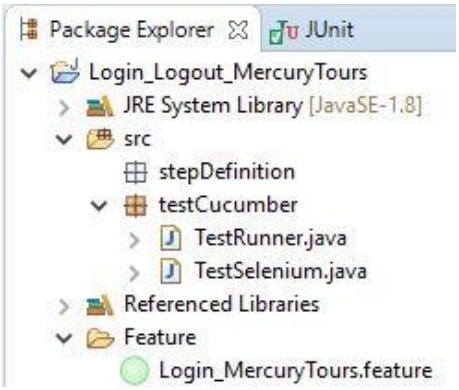

Figure 21 Existing files and packages by this point

Next, we finally add a class to the “stepDefinition” package, that contains the earlier created Java code for each corresponding Gherking keyword.
Add a new class with the name “TestSteps” to the “stepsDefinition”, it is important not the check the option for “public static void main” and finally click on the “Finish” button.

After running the “TestRunner” class for the first time, in the console window there was a suggested code syntax. This can be now copied below the text
“You can implement missing steps with the snippets below:”
as it is and paste to the “TestSteps” class.
The directly copied syntax should look like this on the class itself (Figure 22):

```
1  package stepDefinition;
2
3  public class TestSteps {
4
5      @Given("^User is on Mercury Tours home page$")
6      public void user_is_on_Mercury_Tours_home_page() throws Throwable {
7          // Write code here that turns the phrase above into concrete actions
8          throw new PendingException();
9      }
10
11     @When("^User clicks the SIGN-ON button$")
12     public void user_clicks_the_SIGN_ON_button() throws Throwable {
13         // Write code here that turns the phrase above into concrete actions
14         throw new PendingException();
15     }
16
17     @When("^User enters username and password$")
18     public void user_enters_username_and_password() throws Throwable {
19         // Write code here that turns the phrase above into concrete actions
20         throw new PendingException();
21     }
22
23     @Then("^Message is displayed Login Successful$")
24     public void message_is_displayed_Login_Successful() throws Throwable {
25         // Write code here that turns the phrase above into concrete actions
26         throw new PendingException();
27     }
28
29     @When("^User Logs out from the application$")
30     public void user_Logs_out_from_the_application() throws Throwable {
31         // Write code here that turns the phrase above into concrete actions
32         throw new PendingException();
33     }
34
35     @Then("^Message is displayed Logout Successful$")
36     public void message_is_displayed_Logout_Successful() throws Throwable {
37         // Write code here that turns the phrase above into concrete actions
38         throw new PendingException();
39     }
40 }
```

Figure 22 The copied code template after JUnit is run the first time after the .feature file is created.

There are noticeable errors as the keywords have not yet been linked, nor the code added to the commented lines. There is also missing a clear definition for the WebDriver and for the import packages. In order to modify the code more useful, the next step is to add the code and the linkages.

At this point, hover over the annotations and click on the “Import “annotation” (cucumber.api.java.en), which should remove the error underlying on the annotations at this point and add an import linkage to the keywords (Figure 23):

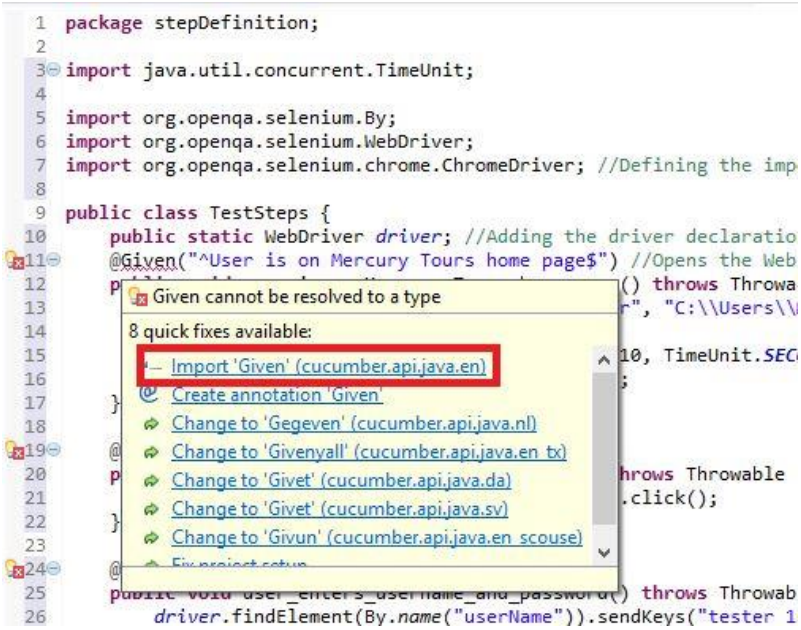

Figure 23 Click the highlighted quick fix to resolve the annotation issue.

After the linkages and code has been added, the class should look similar to the code as shown below (Figure 24):

```
1  package stepDefinition;
2
3  import java.util.concurrent.TimeUnit;
4
5  import org.openqa.selenium.By;
6  import org.openqa.selenium.WebDriver;
7  import org.openqa.selenium.chrome.ChromeDriver; //Defining the import files as before
8
9  import cucumber.api.java.en.Given;
10 import cucumber.api.java.en.Then;
11 import cucumber.api.java.en.When;
12
13 public class TestSteps {
14                public static WebDriver driver; //Adding the driver declaration
15                @Given("^User is on Mercury Tours home page$") //Opens the Web site
16                public void user_is_on_Mercury_Tours_home_page() throws Throwable {
17                                System.setProperty("webdriver.chrome.driver", "C:\\Us-
18 ers\\Mari\\Documents\\Selenium\\chromedriver_win32\\chromedriver.exe");
19         driver = new ChromeDriver();
20         driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
21         driver.get("http://newtours.demoaut.com/");
22                }
23
```

```
24                  @When("^User clicks the SIGN-ON button$")
25                  public void user_clicks_the_SIGN_ON_button() throws Throwable {
26                                  driver.findElement(By.linkText("SIGN-ON")).click();
27                  }
28
29                  @When("^User enters username and password$")
30                  public void user_enters_username_and_password() throws Throwable {
31                                  driver.findElement(By.name("userName")).sendKeys("tester_1");
32                                  driver.findElement(By.name("password")).sendKeys("test1234");
33                                  driver.findElement(By.name("login")).click();
34                  }
35
36                  @Then("^Message is displayed Login Successful$")
37                  public void message_is_displayed_Login_Successful() throws Throwable {
38                                  System.out.println("Login Successful");
39                  }
40
41                  @When("^User Logs out from the application$")
42                  public void user_Logs_out_from_the_application() throws Throwable {
43                                  driver.findElement(By.linkText("SIGN-OFF")).click();
44                  }
45
46                  @Then("^Message is displayed Logout Successful$")
47                  public void message_is_displayed_Logout_Successful() throws Throwable {
48                                  System.out.println("Logout Successful");
49                                  driver.quit();
50                  }
51 }
```

Figure 24 The code implementation for the generated template using the earlier functional Java code (Sharma, L. 2014k).

Run the Cucumber test by right clicking on the “TestRunner” class and then clicking: “Run As” > “Junit Test”

What should happen, is the exact same as when the Java program code was created for the first time. The browser is prompted automatically on screen, the login page is opened, the user name and password are inserted, the user is logged in and then immediately logged out, and finally the browser closes. Before this, there was no indication on the passed steps or for the final report on which scenarios had passed. This time the Eclipse should look like this after the test has completed the automated process (Figure 25):

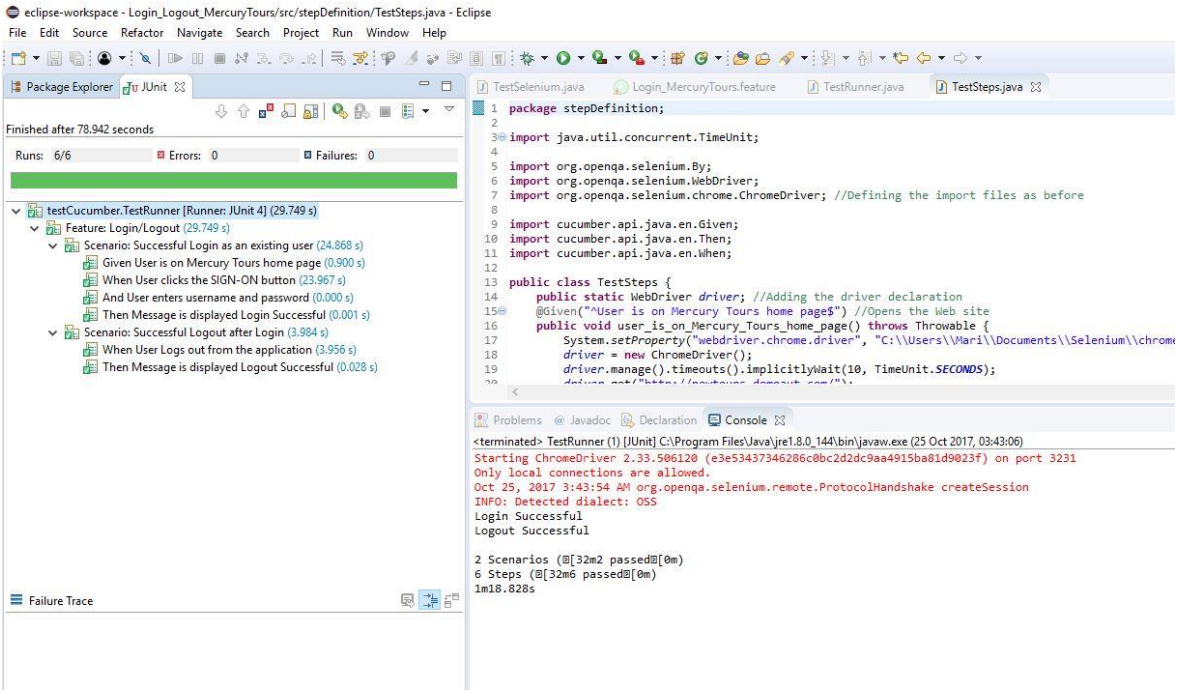

Figure 25 The final reporting screen after the automated test has been completed.

From this screen all the steps can be seen as passed, and how long each step took time in seconds along with the final summed up time of how long the process took altogether. (Sharma, L. 2014k).

This concludes the very basic tutorial on how to automate a simple Login process for a web site as an example using testing tools like Selenium and Cucumber with Gherkin syntax. The same process could be implemented on many different scenarios, depending of course on suitability and by tweaking the element names and code to fit the purpose.

8 Conclusion

My initial target for my thesis was to provide the reader as compact and as comprehensive guide for understanding automation testing and testing in general as it was possible within my own skillsets. For me the topic of automation testing seemed very unapproachable and difficult to grasp, and proved to be a great task to learn and to understand some basics regarding the concepts. Especially, since I have quite limited background on programming, which proved to be a great challenge when implementing the case study. I was able to find a great and explanatory tutorial for helping me to get started and to immerse myself with new tools and environments. The only programming experience I've had so far was from learning the very basics of C# programming language for Microsoft Visual Studio environment. Thus, using Java programming language and Eclipse as a new environment was a learning curve, which I was fortunately able to tackle to an extent as C# does share a few similarities with the syntax. Another challenge that was faced was the initial misconceptions and understanding of the tools that were selected from the beginning for the case study. As many internet sources cited, Cucumber is a testing framework. Due to this, it took me by surprise after investing time in my selections to find out that this was in fact not the case and it was instead utilized as more of a collaboration tool for the BDD framework usage. Although, the discovery helped me to see the benefits and potential of the combination of Selenium and Cucumber usage from a new viewpoint. I decided to keep Cucumber as the framework, despite its divisive status among automation developers.

Regarding the thesis itself, I wanted to start out by introducing the automation testing concepts and about how testing itself fits into the stages within the software development lifecycle. I did not go deeper into the topic within the scope of the thesis as deciphering the models and methodology further could warrant too much side-tracking from the actual topic at hand. I did however feel the need to introduce the V-Model, since it resonated with the topic of testing more accurately, as did TDD and by introducing BDD for its relatedness in terms of the case study. It is important to understand some of the testing methods available and how automation testing could be utilized, hence I felt the need to introduce some core testing methods that could be beneficial for the reader to know, especially concerning methods that can be automated.
As test automation is tool-assisted testing, I decided to give some examples of testing tools for both the test management and manual testing side and also for the tools and frameworks for automation testing. Including the tools I decided to utilize for the case study. Another important aspect to introduce in my thesis was the basics of the general

test process, from test planning to the test management tool planning and also regarding reporting. I also introduced possible risks related to automation testing, since it is important to understand situations that could cause miscommunication and issues, especially if the team implementing automation testing would be inexperienced or unfamiliar with possible risks. Finally, I used all that I had learned through the process of learning about the topic and utilized a prominent tutorial I had found to illustrate for the reader on how to possibly configure, build and implement a basic scenario for automated login testing. The reason for this was that I wanted to learn a basic way to implement automated testing myself, but I also felt that it would be easier for the reader to grasp the topic through a tangible example.

The grounds for selecting Selenium WebDriver as my automation tool was that it was a commonly used browser automation test tool, which would also mean a feasible realization for almost any web site. I selected Cucumber, because it enabled the use of the BDD syntax. As I myself am not a particularly technical person, it seemed compelling to learn the syntax for possible future need as the utilization was geared toward easier communication between the technical and the non-technical team members in terms of the understandability of the test scripts. I had to select Java as the implementable programming language, since Cucumber did not really support the use of C#. Although for me this seemed like an amazing opportunity to familiarize myself with Java programming language and Eclipse as the development environment.

What really was beneficial for me after the entire ordeal of leaning new concepts and building a working demo, was the entire learning experience in itself. I feel I was able to convey the concepts for a person with limited initial knowledge such as myself, and to gather as much information to a compact package as possible. Or at least from the perspective of what I have learned or would have liked to learn initially.
As a tester currently myself, I was able to utilize my prior knowledge on software testing concepts during the theory and research, which was beneficial in terms of shortened learning curve on some of the topics. By learning more of the topic of technical testing has given me the grounds to develop as a professional even further and to understand the topic more in possible future projects and more importantly to understand the possible risks that may be faced.

References

Aebersold, K. 2017. Test Automation Frameworks. URL: https://smartbear.com/learn/automated-testing/test-automation-frameworks/. Accessed: 15 September 2017.

Agile Alliance 2017a. Glossary – TDD. URL: https://www.agilealliance.org/glossary/tdd/#q=~(filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'tdd))~searchTerm~'~sort~false~sortDirection~'asc~page~1). Accessed: 5 October 2017.

Agile Alliance 2017b. Glossary – Refactoring. URL: https://www.agilealliance.org/glossary/refactoring/#q=~(filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'refactoring))~searchTerm~'~sort~false~sortDirection~'asc~page~1). Accessed: 5 October 2017.

Agile Alliance 2017c. Glossary – BDD. URL: https://www.agilealliance.org/glossary/bdd/#q=~(filters~(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'bdd))~searchTerm~'~sort~false~sortDirection~'asc~page~1). Accessed: 1 November 2017.

Angerer, M. 12 March 2015. HP Sprinter is the Smart Alternative. URL: https://resultspositive.com/hp-sprinter-is-the-smart-alternative/. Accessed: 14 September 2017.

Atlassian Marketplace 2017b. TestRail for Jira - Test Management. URL: https://marketplace.atlassian.com/plugins/com.testrail.jira.testrail-plugin/cloud/overview. Accessed: 20 October 2017.

Atlassian Marketplace 2017a. Zephyr for Jira - Test Management. URL: https://marketplace.atlassian.com/plugins/com.thed.zephyr.je/cloud/overview. Accessed: 20 October 2017.

Atlassian Marketplace 2017c. Xray - Test Management for Jira. URL:

https://marketplace.atlassian.com/plugins/com.xpandit.plugins.xray/server/overview. Accessed: 20 October 2017.

Bartlett, J. 2 December 2015. How to Write Test Cases for Software (with a sample). URL: https://blog.testlodge.com/how-to-write-test-cases-for-software-with-sample/. Accessed: 20 April 2017.

Berg, C. 29 July 2009. Continuous Integration vs. Nightly Build. URL: https://www.slideshare.net/poyklr/continuous-integration-vs-nightly-build. Accessed: 28 October 2017.

Cloudbees.com 2017. About Jenkins. URL: https://www.cloudbees.com/jenkins/about. Accessed: 28 October 2017.

Cobertura 2.1.1. URL: http://cobertura.github.io/cobertura/. Accessed: 25 September 2017.

Continuumsecurity.com 2017. BDD-SECURITY - SECURITY TESTING FRAMEWORK. URL: https://www.continuumsecurity.net/bdd-security/. Accessed: 25 October 2017.

Cuadra, J. 31 May 2012. Please don’t use Cucumber. URL: https://www.jimmycuadra.com/posts/please-don-t-use-cucumber/. Accessed: 30 October 2017.

Cucumber.io. Getting started with Cucumber. URL: https://cucumber.io/docs. Accessed: 16 September 2017.

cucumber/cucumber 2017. Gherkin. URL: https://github.com/cucumber/cucumber/wiki/Gherkin. Accessed: 14 October 2017.

Dustin, E., Garret, T. & Gauf, B. 2009. Implementing Automated Software Testing. Pearson Education. Massachusetts.

Eriksson, U. 5 April 2012. Functional Requirements VS Non-Functional Requirements. URL: http://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/. Accessed: 17 November 2017.

Functionize.com 12 June 2017. Smoke testing Suite: What it is, Why You Need it, and How to Automate. URL: https://www.functionize.com/blog/smoke-testing-suite. Accessed: 15 October 2017.

Google Sites. ChromeDriver - WebDriver for Chrome. URL: https://sites.google.com/a/chromium.org/chromedriver/. Accessed: 27 September 2017.

Guru99 2017a. What is Regression Testing? Test Cases, Tools & Examples. URL: https://www.guru99.com/regression-testing.html. Accessed: 15 May 2017.

Guru99 2017b. UNIT Testing Tutorial - Learn in 10 Minutes. URL: https://www.guru99.com/unit-testing-guide.html. Accessed: 20 October 2017.

Guru99 2017c. System INTEGRATION Testing (SIT): Complete Tutorial. URL: https://www.guru99.com/system-integration-testing.html. Accessed: 2 November 2017.

Guru99 2017d. What is System Testing? Types & Definition with Example. URL: https://www.guru99.com/system-testing.html. Accessed: 25 September 2017.

Guru99 2017e. Exploratory Testing Tutorial: Process, Techniques & Examples. URL: https://www.guru99.com/exploratory-testing.html. Accessed: 14 September 2017.

Guru99 2017f. What is Security Testing: Complete Tutorial. URL: https://www.guru99.com/what-is-security-testing.html. Accessed: 25 October 2017.

Guru99 2017g. Introduction to HP ALM(Quality Center). URL: https://www.guru99.com/hp-alm-introduction.html. Accessed: 20 October 2017.

Hellesøy, A. 3 March 2014. The world's most misunderstood collaboration tool. URL: https://cucumber.io/blog/2014/03/03/the-worlds-most-misunderstood-collaboration-tool. Accessed: 30 October 2017.

Homès, B. & Homes, B. 2011. Fundamentals of Software Testing. John Wiley & Sons, Incorporated.

ISTQB Exam Certification 2017a. What is Unit Testing?. URL: http://istqbexamcertification.com/what-is-unit-testing/. Accessed: 4 October 2017.

ISTQB Exam Certification 2017b. What is Component Testing?. URL: http://istqbexamcertification.com/what-is-component-testing/. Accessed: 4 October 2017.

ISTQB Exam Certification 2017c. What is Security testing in software testing?. URL: http://istqbexamcertification.com/what-is-security-testing-in-software/. Accessed: 25 October 2017.

ISTQB Exam Certification 2017d. What is test status report? and How to report test status?. URL: http://istqbexamcertification.com/what-is-test-status-report-and-how-to-report-test-status/. Accessed: 5 November 2017,

Jain, A. 1 September 2017. What is UFT (QTP)?. URL: https://www.learnqtp.com/what-is-qtp/. Accessed: 25 October 2017.

JBehave 2015. Feature of JBehave. URL: http://jbehave.org/reference/stable/features.html. Accessed: 16 September 2017.

Jenkins. Jenkins User Documentation. URL: https://jenkins.io/doc/. Accessed: 28 October 2017.

Johnson, E. 22 July 2015. Test Driven Development (TDD) in a Nutshell. Intland software. URL: https://intland.com/blog/agile/test-management/test-driven-development-tdd-nutshell-overview/. Accessed: 5 October 2017.

JUnit Version 4.12. 10 September 2017. About. URL: http://junit.org/junit4/. Accessed: 26 October 2017.

junit-team/junit4 17 September 2017. Test runners. URL: https://github.com/junit-team/junit4/wiki/Test-runners. Accessed: 26 October 2017.

Koomen, T., van der Aalst, L., Broekman, B. & Vroon, M. 2014. TMap NEXT® for result-driven testing. Sogeti Nederland B.V. Vianen.

Kudryashov, K. 7 October 2015. The beginner’s guide to BDD. URL: https://inviqa.com/blog/bdd-guide. Accessed: 1 November 2017

Kaul, N. 24 April 2017. How JIRA Led to the Demise of HP ALM. URL:

https://blog.smartbear.com/test-management/how-jira-led-to-the-demise-of-hp-alm/. Accessed: 20 October 2017.

Kaul, N. 8 March 2016. The Pros and Cons of Using Excel for Test Management. URL: https://blog.smartbear.com/automated-testing/pros-and-cons-using-excel-test-management/. Accessed: 20 October 2017.

Knight, A. 27 January 2017. BDD 101: GHERKIN BY EXAMPLE. URL: https://automationpanda.com/2017/01/27/bdd-101-gherkin-by-example/. Accessed: 14 October 2017.

Mercury Interactive 2005. Mercury Tours. URL: http://newtours.demoaut.com/. Accessed: 25 September 2017.

Montvelisky, J. 4 March 2008. Master Test Plan – the strategic side of testing. URL: http://qablog.practitest.com/master-test-plan-testing-strategic-side/. Accessed: 17 November 2017.

mozilla/geckodriver 2017. Geckodriver. URL: https://github.com/mozilla/geckodriver. Accessed: 27 September 2017.

Nicieja, K. 2 March 2016. 3 myths about Cucumber and Gherkin. URL: https://pilot.co/blog/cucumber-and-gherkin-myths/. Accessed: 30 October 2017.

Niittyvirta, A. Prove Expertise Ltd. URL: http://www.testiatarantula.com/. Accessed: 20 October 2017.

Robot Framework User Guide Version 3.0.2. 2016. 2.1.2 Supported file formats. URL: http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#supported-file-formats. Accessed: 18 November 2017.

Robot Framework. Introduction. URL: http://robotframework.org/. Accessed: 3 November 2017.

Sahla, K. 18 September 2017. Senior Consultant. Sogeti Finland Oy. Company internal learning course. Espoo.

SeleniumHQ 19 November 2017a. Selenium WebDriver. URL:

http://www.seleniumhq.org/docs/03_webdriver.jsp. Accessed: 20 November 2017.

SeleniumHQ 19 November 2017b. Selenium-IDE. URL: http://www.seleniumhq.org/docs/02_selenium_ide.jsp. Accessed: 20 November 2017.

Sharma, L. 29 January 2017. User Acceptance Testing – UAT. URL: http://toolsqa.com/software-testing/user-acceptance-testing-uat/. Accessed: 29 October 2017.

Sharma, L. 31 December 2014a. Cucumber Tutorial. URL: http://toolsqa.com/cucumber/cucumber-tutorial/. Accessed: 25 September 2017.

Sharma, L. 28 August 2015b. Download and Install Java. URL: http://toolsqa.com/selenium-webdriver/download-and-install-java/. Accessed: 25 September 2017.

Sharma, L. 28 August 2015c. Download and Start Eclipse. URL: http://toolsqa.com/selenium-webdriver/download-and-start-eclipse/. Accessed: 25 September 2017.

Sharma, L. 27 December 2014d. Install Cucumber Eclipse Plugin. URL: http://toolsqa.com/cucumber/install-cucumber-eclipse-plugin/. Accessed: 25 September 2017.

Sharma, L. 28 December 2014e. Download Cucumber JVM for Eclipse. URL: http://toolsqa.com/cucumber/download-cucumber-jvm-eclipse/. Accessed: 25 September 2017.

Sharma, L. 28 August 2015f. Download Selenium Webdriver Java client. URL: http://toolsqa.com/selenium-webdriver/download-selenium-webdriver-java-client/. Accessed: 25 September 2017.

Sharma, L. 28 December 2014g. Configure Eclipse with Cucumber. URL: http://toolsqa.com/cucumber/configure-eclipse-cucumber/. Accessed: 26 September 2017.

Sharma, L. 29 December 2014h. First Cucumber Selenium Java Test. URL: http://toolsqa.com/cucumber/first-cucumber-selenium-java-test/. Accessed: 27 September 2017.

Sharma, L. 9 December 2015i. Cucumber Feature File. URL: http://toolsqa.com/cucumber/cucumber-jvm-feature-file/. Accessed: 27 September 2017.

Sharma, L. 9 December 2015j. JUnit Test Runner Class. URL: http://toolsqa.com/cucumber/junit-test-runner-class/. Accessed: 27 September 2017.

Sharma, L. 31 December 2014k. Step Definition. URL: http://toolsqa.com/cucumber/step-definition/. Accessed: 27 September 2017.

Singh, V. 31 December 2014. Gherkin. URL: https://automationpanda.com/2017/01/27/bdd-101-gherkin-by-example/. Accessed: 14 October 2017.

Software Testing Fundamentals 2017a. Integration Testing. URL: http://softwaretestingfundamentals.com/integration-testing/. Accessed: 4 October 2017.

Software Testing Fundamentals 2017b. Unit Testing. URL: http://softwaretestingfundamentals.com/integration-testing/. Accessed: 4 October 2017.

Software Testing Fundamentals 2017c. SMOKE TESTING Fundamentals. URL: http://softwaretestingfundamentals.com/smoke-testing/. Accessed: 15 October 2017.

Software Testing Genius. Automated Functional Testing Techniques and Risks Associated with Automation. URL: http://www.softwaretestinggenius.com/automated-functional-testing-techniques-and-risks-associated-with-automation. Accessed: 16 November 2017.

Software Testing Help 2017. A Simple 12 Steps Guide to Write an Effective Test Summary Report. URL: http://www.softwaretestinghelp.com/test-summary-report-template-download-sample/. Accessed: 5 November 2017.

Software Testing Studio, 2 March 2017. Requirements Analysis Is Vital For Effective Software Testing. URL: https://hubtechinsider.wordpress.com/2011/07/28/how-do-you-write-software-requirements-what-are-software-requirements-what-is-a-software-requirement/. Accessed: 15 May 2017.

Stewart, S. 2010. Selenium WebDriver. URL:

http://www.aosabook.org/en/selenium.html. Accessed: 16 September 2017.

Tarantula 2017. Agile Test Management. URL:
http://www.tarantula.fi/old/. Accessed 20 October 2017.

TestRail 2017. Modern test management software tool. URL:
http://www.gurock.com/testrail/software-testing-reports.l.html. Accessed: 5 November 2017.

The Apache Software Foundation 2017. Apache JMeter™. URL:
http://jmeter.apache.org/. Accessed: 4 November 2017.

Tutorialspoint.com 2017a. Software Development Life Cycle. URL:
https://www.tutorialspoint.com/software_engineering/software_development_life_cycle.htm. Accessed: 28 October 2017.

Tutorialspoint.com 2017b. SDLC - Overview. URL:
https://www.tutorialspoint.com/sdlc/sdlc_overview.htm. Accessed: 28 October 2017.

Tutorialspoint.com 2017c. SDLC - Quick Guide. URL:
https://www.tutorialspoint.com/sdlc/sdlc_quick_guide.htm. Accessed: 28 October 2017.

Tutorialspoint.com 2017d. SDLC – V-Model. URL:
https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm. Accessed: 28 October 2017.

Tutorialspoint.com 2017e. Performance Testing. URL:
https://www.tutorialspoint.com/software_testing_dictionary/performance_testing.htm. Accessed: 5 June 2017.

Tutorialspoint.com 2017f. Security Testing. URL:
https://www.tutorialspoint.com/software_testing_dictionary/security_testing.htm: 25 October 2017.

Tutorialspoint.com 2017g. Security Testing - Injection. URL:
https://www.tutorialspoint.com/security_testing/testing_injection.htm. Accessed: 25 October 2017.

Tutorialspoint.com 2017h. Test Execution. URL:

https://www.tutorialspoint.com/software_testing_dictionary/test_execution.htm. Accessed: 15 October 2017.

Wilcox, R. 2017. Your Boss Won't Appreciate TDD: Try This Behavior-Driven Development Example. URL: https://www.toptal.com/freelance/your-boss-won-t-appreciate-tdd-try-bdd. Accessed: 1 November 2017.

Wikipedia 24 October 2017. Mockito. URL: https://en.wikipedia.org/wiki/Mockito. Accessed: 25 September 2017.

Appendices

Appendix 1. Central Keywords List

Black-box testing	*Testing where the internal structures of the application, i.e. the program code is not known to the tester.*
Cucumber	*Behavior Driven Development Framework, to implement a syntax in the form of "Given-When-Then", as programmable automation test cases.*
Defect	*A reported bug in the system or application found from the functionality, code, or design.*
Driver	*A computer program, used to control a certain device or function attached to the computer or application. Acts as sort of a translator between programs or devices.*
Framework	*In software development, a set of rules that provide certain functionality to tools and programs. These may be for example supporting programs, code libraries, compilers etc.*
Gherkin	*A Business Readable, Domain Specific Language that provides the functionality to write Behavior Driven Development syntax for i.e. automated test cases. Mimics everyday grammar for easier understandability.*
IDE	*Integrated Development Environment, a software application used by programmers to code and develop a program. Usually contains an editor for the source code, a compiler, and possibly a debugging function. Examples would be Eclipse, Microsoft Visual Studio and NetBeans.*

Jira	*A software for issue, bug and project advancement tracking, but does not contain native test management capabilities. A proprietary software developed by Atlassian.*
MTP	*Master Test Plan, an overall plan on how to progress with test planning and test management created as a first test before test planning.*
Non-repudiation	*To repudiate is to deny. Non-repudiation, the assurance that one can't deny something, i.e. if a contract has been signed.*
SDLC	*Software Development Life Cycle, a set of models and methodologies to guide the progress of the development of a software or a system.*
Stub	*A replacement of a certain functionality (piece of code) within a program code. For instance when testing a certain part of a program that is missing a critical functionality in order to mimic that.*
SUT	*System Under Test, a term in software testing when referring to a particular system that is being tested for correct operations.*
Test Management	*The actual activity of managing the testing in general in accordance to the test plan. Whether being for manual or automation testing.*
TSV	*Tab-separated values, a simple text format to store data in tabular structure. For instance data in Excel sheets and database tables. Used i.e. for Robot Framework.*
UAT	*User Acceptance Testing, the final phase in testing where the client verifies the behaviour of the*

	program and how well the requirements were met before moving to production stage.
White-box testing	*As opposed to black-box testing, the internal workings of the system and programming skills are used to formulate test cases and to test i.e. the backend side.*