jamk.fi

Determination and Implementation of Mobile Testing Automation Tool

Descom

Joona Mulari
Mikko Wilmi

Thesis
August 2015

Business Information Systems
School of Business

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author(s) Mulari, Joona Wilmi, Mikko	Type of publication Bachelor's thesis	Date 6.10.2015
		Language of publication: English
	Number of pages 78+2	Permission for web publication: x
Title of publication **Determination and Implementation of Mobile Testing Automation Tool**		
Degree programme Business Information Systems		
Tutor(s) Kiviaho, Niko		
Assigned by Descom Oy		

Abstract

The objective of the thesis was to examine which mobile testing automation tool would work best for mobile versions of online stores developed by Descom and how it could be integrated and used as a part of the continuous development system already in use at Descom.

In the study based on design research (applied action research), many mobile testing automation tools were analyzed. Those tools were compared to eleven evaluation criteria published by TestHuddle and to six selection criteria suggested by Descom. Based on these studies one tool was chosen. It was then installed, configured and attached to testing automation infrastructure, similar to Descom's.

The chosen tool turned out to be very suitable for all of Descom's needs and it fitted perfectly to the already established testing infrastructure. The testing methods used at Descom supported the ones used with the chosen tool and the deployment and usage was a fairly straightforward task to manage.

The tool adds a new, fairly inscrutable point of view to the testing strategy used at Descom, increasing the overall coverage of testing. It will help to reduce possible problems and bugs that might occur on mobile versions of Descom's online stores.

Keywords/tags (subjects)

Testing, Mobile test automation, Framework, Appium

Miscellaneous

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Kuvailulehti

Tekijä(t) Mulari, Joona Wilmi, Mikko	Julkaisun laji Opinnäytetyö	Päivämäärä 6.10.2015
	Sivumäärä 78+2	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Mobiilitestausautomaatiotyökalun valinta ja käyttöönotto

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Työn ohjaaja(t)
Niko Kiviaho

Toimeksiantaja(t)
Descom Oy

Tiivistelmä

Opinnäytetyön tavoitteena oli tutkia, mikä mobiilitestauksen automatisointiin sopiva työkalu kävisi parhaiten Descom Oy:n kehittämien verkkokauppojen mobiilisivujen testaukseen ja kuinka se voitaisiin liittää yrityksen jo valmiiksi rakennettuun jatkuvan julkaisun ratkaisumalliin.

Kehittämistutkimuksessa otettiin tarkastelun alle monia mobiilitestauksen automatisointiin tarkoitettuja työkaluja. Niitä verrattiin yhteentoista TestHuddle-sivustolla julkaistussa artikkelissa esitettyyn valintakriteeriin ja kuuteen Descomin puolelta esitettyyn valintakriteeriin. Näiden perusteella päädyttiin yhteen työkaluun, joka asennettiin, konfiguroitiin ja liitettiin osaksi testausautomaatioinfrastruktuuria, joka vastasi Descomilla käytössä olevaa järjestelmää.

Valittu työkalu sopi ongelmitta Descomilla käytettyyn infrastruktuuriin. Työkalu sopi hyvin myös Descomilla käytettyihin testausmenetelmiin eikä sen käyttöönotto tai käyttäminen vaatinut liian monimutkaisia toimenpiteitä tai ponnisteluja.

Työkalu lisää Descomin testausstrategiaan uuden, ennen tätä opinnäytetyötä vähän tutkitun näkökulman, jonka avulla testaus on entistä kattavampi kokonaisuus. Se tulee varmasti vähentämään ongelmia ja bugeja, joita mobiilinettikauppojen kehityksessä saattaa tulla vastaan.

Avainsanat (asiasanat)
Testaus, mobiili testausautomaatio, sovelluskehys, Appium

Muut tiedot

Contents

Figures

Tables

Acronyms and terminology

Apache 2.0 License	Software released under Apache 2.0 License can be modified and distributed without concern for royalties.
API	An application programming interface.
APK	Android application package
App	Application
AUT	Application under test
ATDD	Acceptance test-driven development
CI	Continuous integration
CSS	Cascading Style Sheet
GUI	Graphical user interface
HTML	HyperText Markup Language
JAR	Java Archive. Package file format for Java class files.
Jenkins	Continuous integration tool
JSON	JavaScript Object Notation
KIF	Keep It Functional
LTS	Long-term support
IDE	Integrated development environment
MSI	Windows Installer. Used for installing, maintaining and deletion of the software on Windows systems.
Node.js	Runtime environment written in JavaScript for network applications.
npm	Package manager for JavaScript.
pip	Package management system for installing and managing software written in Python.
POM	Project Object Model
Python	Cross-platform programming language that is suitable for building almost any type of program.
reST	reStructuredText
Robot Framework	Framework for generic test automation that is used for acceptance testing and ATDD.
SCM	Software configuration management

SDK	Software development kit
Selenium	A collection of tools that are used to help automate software testing.
TSV	Tab-separated values
UI	User interface
URL	Uniform resource locator
VM	Virtual machine
WAR	Web application archive
XML	Extensible Markup Language

1 Introduction

This thesis focuses on researching which mobile testing automation tool worked best for smoke and regressions tests for mobile versions of the online stores developed by Descom Oy and how it is integrated and used with their systems. This was achieved by examining a selection of tools with the aid of eleven-step selection that were introduced in an article published in TestHuddle. Those tools were also compared to a certain list of criteria introduced by the representatives of Descom. After the tool was chosen it was installed, configured and combined with the technologies already in use.

Online shopping has been a part of people’s lives for almost two decades. Major online retail shops like Amazon and eBay opened their websites in the mid-1990s paving the way for smaller, more independent businesses and giving the populace possibility to purchase items that might not even be possible to find anywhere near the customer online with only a click of a mouse. Nowadays as more and more people use mobile devices for their daily activities on the internet, web stores also have to be compatible with the ever-so-changing array of mobile apparatuses, which poses many potential issues for companies developing those web stores. Bugs and errors might run rampant and the companies might lose paying customers if those web stores are not tested with care. This testing is crucial to the success of the websites and it should be done well. (Online shopping, 2015.)

Often these tests have to perform repetitive and very labor intensive tasks which are not ideal to test manually by a human, which is the reason why it is important to take automation into account when developing those tests. While the testing automation is in a very advanced stage for desktop software and web content, the same technologies do not necessarily work in the mobile world. At Descom the test engineers are more familiar with the testing of the desktop sites than the mobile versions, and therefore it is important for this thesis to find and figure out a way to implement a simple and working solution for the mobile testing automation that is relatively easy to use and learn.

Thesis assigner

Descom is a company specialized in marketing and technology. It was founded in 1997 in Jyväskylä, Finland. Descom provides different types of customer experiences ranging from marketing and sales to customer service. They also design and implement web stores to customers. The company employs over 260 employees and operates in four countries, including Sweden and Poland. In 2014 Descom's turnover was over 35 million euros. (Descom's website)

Thesis structure

Chapter three of this thesis concentrates on the basic principles of the software testing and software testing automation.

Chapter four gives an outlook of the test automation tools that are relevant to the thesis and also general info about the mobile test automation tools. In chapter six these mobile test automation tools are compared against each other and also against criteria provided by Descom.

Chapter five explains the basic principle of the continuous integration and Jenkins tool. Jenkins is an integral part of the testing automation at Descom so it is important to get to know that tool.

Chapter six concentrates on the selection process of the mobile test automation tool and chapter seven focuses on the implementation process of said tool.

2 Research and implementation

2.1 Research questions

This thesis aimed to answer the following research questions:

- Which tool is the best suited for mobile web automation testing at Descom?
- How to integrate the chosen tool into existing testing infrastructure at Descom?

These research questions were the basis for the thesis writing process. The following research method (described below) was used to accomplish this goal.

2.2 Research method

The thesis was created using design research (applied action research) method to get familiar with the systems Descom was currently using for test automation, and sample test cases for the e-shops were created to get to know the system and tools. In meetings following this phase it was discussed with Descom representatives what kind of mobile web automation tools would be best suited for this case. Mainly online sources were searched (because the subject is fairly new in the field of test automation) for the mobile automation tools and compared to each other.

As stated by Kananen (2012, 19) design research (applied action research) starts with the need for the change for the better. Design research (applied action research) is always based on the theory basis and the output of the research relies heavily on that.

The subject of the development can be process, action, state of affairs or product, so in other words anything that can be affected. Affecting the subject is called an intervention. It is important to define what kind of actions are required to make the desired change. (Kananen, 2012, 21)

3 Software testing

3.1 Software testing in general

Software testing is an important part of the software development. Bad quality software can cause a variety of problems, such as loss of time or money, and in the worst case scenario it can cause death. Often faults are caused by human errors but it is also possible that some external environment conditions affect the software’s performance, for example radiation or magnetism. (Certified Tester Foundation Level Syllabus, 2011.)

To reduce risks it is of utmost importance to test the system properly. When defects are found and fixed, it improves the quality of the system as a whole. And, when tests do not find many faults it gives confidence about the quality of the system. (Ibid.)

Static testing

In static testing the software is not executed. Instead, it includes manual examinations of the code, for example. These are called reviews. Defects found during reviews are usually cheaper to fix than if they are fixed during the dynamic testing. In addition to code design specifications, test cases and user guides can also be reviewed, among others. Reviews can be good for finding oversights in requirement specifications, for example. These kind of oversights are not easily found during dynamic testing. (Certified Tester Foundation Level Syllabus, 2011.)

Dynamic testing

In contrast to the static testing, dynamic testing includes execution of the software. To put it simply, dynamic testing finds the actual failures but not the cause for them. (Ibid.)

White-box testing

In white-box testing the person doing the testing is familiar with the structure of the program. The downside in this kind of strategy is the fact that specifications are sometimes not considered. (Myers, Sandler, & Badgett 2012, Chapter 2)

Usually white-box testing is done in unit testing level, however, it can also be done in integration and system levels. Since most times unit testing is done by the programmers themselves they are already well-versed in the program’s source code. Some of the white-box testing techniques include (White-box testing, 2015):

- Data flow testing
- Branch testing
- Statement coverage

White-box tests can be easily automated. While white-box testing is very effective for finding bugs it does not consider the fact that some of the features may not yet have been implemented into the software. (Ibid.)

Advantages:

- The source code should be better optimized after white-box testing since this method shows defects.
- The knowledge of the source code is really helpful.

Disadvantages:

- It tests software as it is currently built and does not consider things that have not yet been implemented (ibid.)

Black-box testing

Black-box testing is a testing strategy where the tester does not know how the software is compiled. The term black-box testing refers to the fact that in this type testing tester cannot see inside the program, i.e. it is a black box. Black-box testing is also known as data-driven testing or functional testing. In black-box testing tester knows what the software under test is supposed to do when he or she does the inputs. Tester does not necessarily need any programming skills. Black-box testing reflects on how the user experiences the program. (Myers, Sandler, & Badgett, 2012, Chapter 2)

According to Myers, Sandler, and Badgett (2012, Chapter 2) it is nearly impossible to test all imaginable combinations of inputs. For this reason a number of different methodologies for black-box testing have been created.

Some of the more popular black-box methodologies (Myers, Sandler, & Badgett 2012, Chapter 4):

- Equivalence partitioning
- Boundary value analysis
- Error guessing

Equivalence partitioning is consisted of the following properties (ibid.):

1. *“It reduces, by more than a count of one, the number of other test cases that must be developed to achieve some predefined goal of "reasonable" testing.*
2. *It covers a large set of other possible test cases. That is, it tells us something about the presence or absence of errors over and above this specific set of input values.”*

The former means basically an approach where the biggest number of inputs are tested with the least amount of test cases. The latter entails that inputs should be divided into a number of equivalence classes. So when a test is performed in one class and it fails, it can be assumed that every other case involving the same class would detect the same error. (Ibid.)

Even though this technique is much better than randomly choosing the test cases to perform, it is not without its flaws. Boundary value analysis (explained below) helps with some of those flaws. (Ibid.)

Boundary value analysis

According to Myers, Sandler, and Badgett (2012, Chapter 4), the following two things are the biggest differentiators in boundary value analysis compared to equivalence partitioning:

1. *“Rather than selecting any element in an equivalence class as being representative, boundary value analysis requires that one or more elements be selected such that each edge of the equivalence class is the subject of a test.*

2. *Rather than just focusing attention on the input conditions (input space), test cases are also derived by considering the result space (output equivalence classes).”*

To put it simply boundary value analysis focuses, among other things, on the minimum and maximum values that can be put in to the input field. Therefore, if an input field should accept 1-255 letters or numbers, the test cases should be written for 0, 1, 255 and 256, for example. (Ibid.)

Grey-box testing

In grey-box testing white-box and black-box methods are combined. It aims to take the best parts of the both methods. Test engineer using grey-box method knows at least some parts of the application’s inner structure; however, the tests are done using black-box approach. (Software Testing Fundamentals, Gray Box Testing)

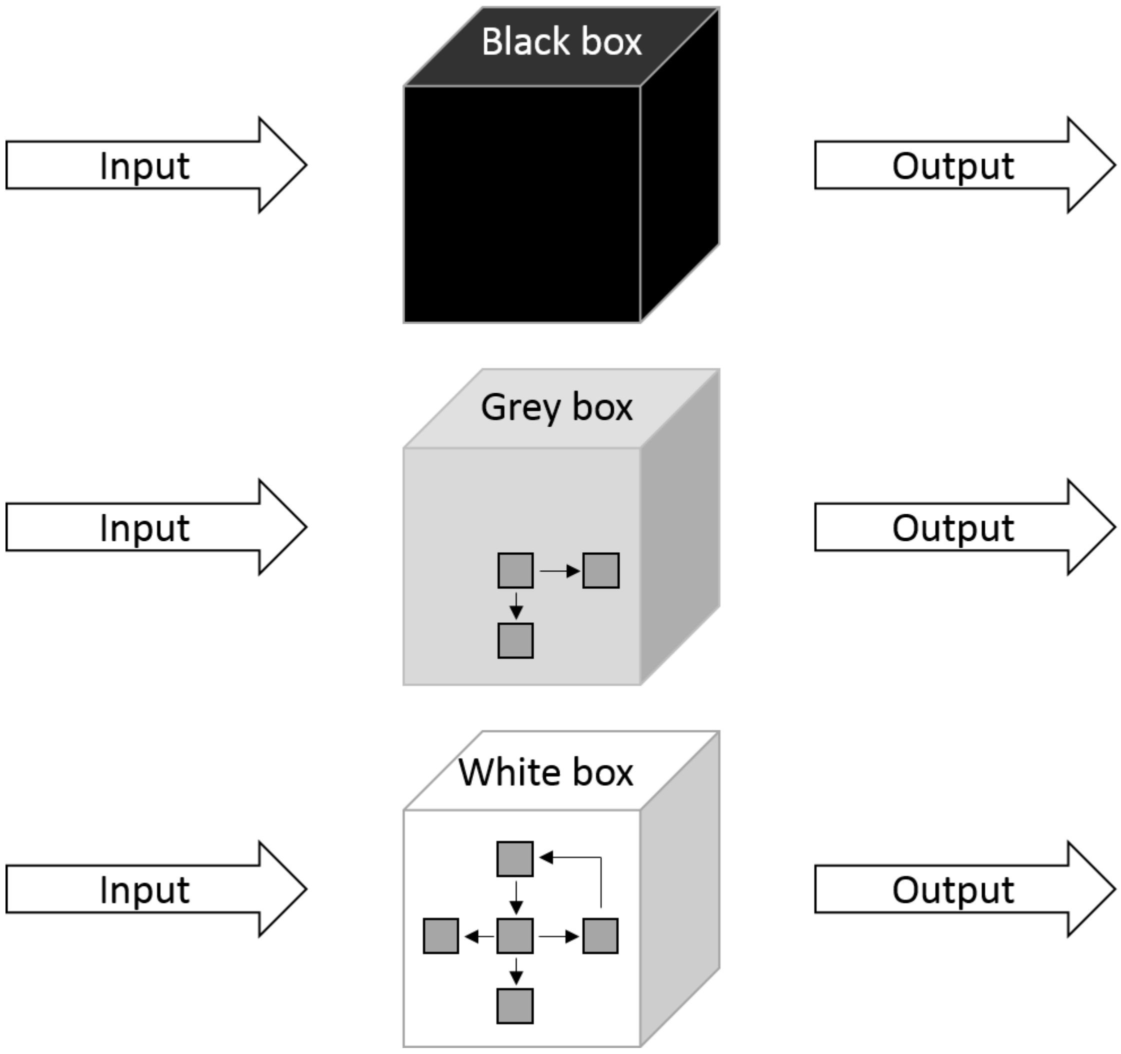


Figure 1. Software test methods (adapted from Comparison among Black-box & White-box Tests, N.d.)

Testing levels

Usually testing is divided into four separate levels: unit testing, integration testing, system testing and acceptance testing. Sometimes they are divided even more precisely; component integration testing, system integration testing, alpha testing and beta testing are added to the mix. (Certified Tester Foundation Level Syllabus, 2011.)

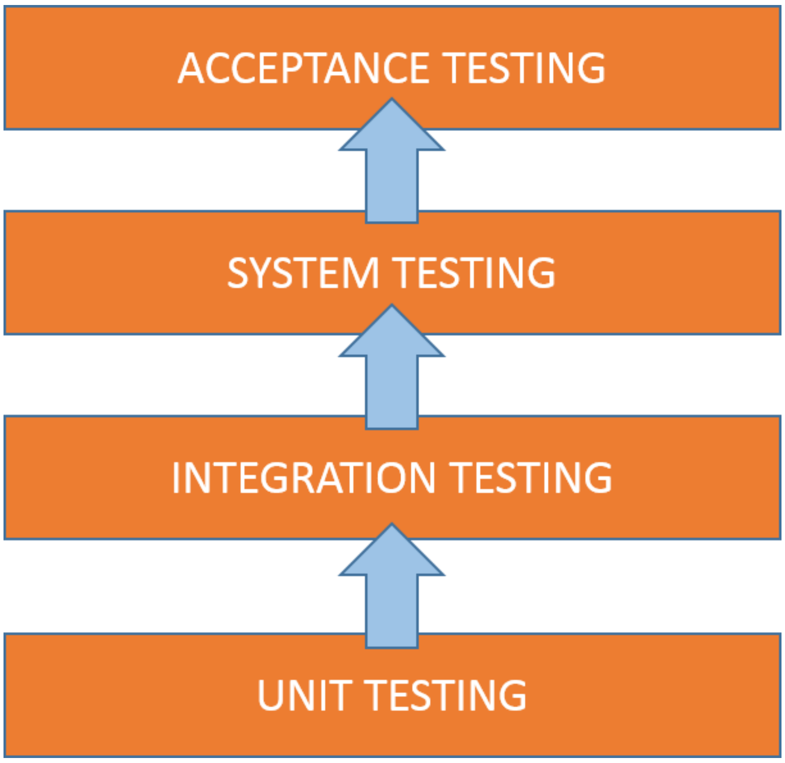


Figure 2. Software test levels (adapted from Software Testing Levels, 2011)

Unit testing

Unit testing is also known as component or module testing. In this phase the smallest parts of the program are tested, for example functions or classes. Unit testing can be done separately, outside of the system. The person doing the tests is also usually the programmer himself/herself. Because of the nature of this type testing source code access is normally required. (Certified Tester Foundation Level Syllabus, 2011.)

By performing unit testing adequately faults can be found very early in the development cycle. This can greatly reduce the costs of the software development. For example, if some bug is found during system testing, it is generally much more costly to fix it then than during unit testing. (ISTQB Exam Certification, What is Unit testing?)

Integration testing

In integration testing phase the aim is to find faults in the components when they are integrated to each other. Sometimes there is more than one level of integration. These are called component integration testing and system integration testing and they are performed after normal integration testing. Component integration testing tests how components work with other components. System integration testing is performed e.g. after system testing and it tests how hardware and software interact. (Certified Tester Foundation Level Syllabus, 2011.)

There are different kinds of strategies for doing the integration testing, such as bottom-up and top-down. In general the faults are easier to find and to isolate if integration testing is done incrementally and not in a so-called big bang. (Ibid.) Big bang basically means an approach where every module is integrated concurrently into the system. The major downside in this approach is the fact that defects are hard to find since the integration happens so late. On the plus side everything is finished before the integration testing begins. (ISTQB Exam Certification, What is Integration testing?)

System testing

System testing includes the testing of the whole system (program/application). It is important that test environment should be as similar as possible to the final target so there should not be any failures that are caused because of the environment. (Certified Tester Foundation Level Syllabus, 2011.)

In this phase it is seen if the requirement specifications are met. Other features that can be tested include business processes and use cases, for example. Also interaction with the operating system can be tested. (Ibid.)

According to the Certified Tester Foundation Level Syllabus (2011), an independent team is often used to perform the system testing.

Since system testing takes place after integration testing level all of the software that is tested in the system testing should have passed the integration phase. System testing includes, however, is not limited to the following:

- Usability testing
- Security testing
- Compatibility testing
- Software performance testing
- Sanity testing (System testing, 2015)

Acceptance testing

Acceptance testing is done to gain confidence that the system works as expected. Although faults can be found in this level of testing it is not the main point of the acceptance testing. Acceptance testing can show if the system is ready for deployment. (Certified Tester Foundation Level Syllabus, 2011.)

Smoke testing

Smoke testing (also known as Build Verification Testing) is a type of testing method that ensures that the software's most essential functions work. Before going on with the testing a smoke test is usually run. This can determine if the tests should be continued. If the smoke test fails all other tests should be suspended and wait for the new build. When a new build of the software is prepared it is good practice to run smoke tests. Smoke tests can be performed manually or by automating them. If there are new builds being released constantly it is probably wise to automate the tests. Smoke testing gives some assurance that changes made to the software have not broken anything. (Software Testing Fundamentals, Smoke Testing)

This type of testing is generally used in Integration, System and Acceptance Testing (ibid.).

Regression testing

Regression testing tries to find new bugs after there have been changes in the software. In regression testing tests that were previously completed are run again to see if the new changes in the software affect them. In other words bugs that were previously fixed should stay fixed even after changes in the software. Regression testing is very labor-intensive so it is a good practice to automate it. (Huston, N.d.)

Ad-hoc testing

Ad-hoc testing is a form of testing where there is not really any structure. Employers doing the testing try to break the system without utilizing any particular test cases. Ad-hoc testing relies on the intuitiveness of the testers so it is exceedingly important that they are experienced and have knowledge of the ins and outs of the system. To make most out of ad-hoc testing it is a good practice to perform the tests on areas of the software which are prone to breaking and/or have a lot of defects. (Nadig, 2015)

3.2 Software test automation

Usually testing has been manual labor in which the test engineer does the testing and sees if the results match the expected results. In general the software should be tested every time there is a change in the code. Doing this manually is very time consuming. Automated tests are run using different kind of automation tools. The benefit of the test automation is the fact that once tests are created they can be executed multiple times and it does not bring additional costs. By doing automated tests it is also possible to simulate multiple simultaneous users. (SmartBear, N.d.)

Benefits

There are many benefits in test automation. For example regression tests can be easily run on a new version of the software. Tests can also be run more often. Some types of testing, like stress tests, would be almost impossible without utilizing test automation. Tests can be easily repeated and they always stay the same since there is little possibility of human error. When tests can be automated testing should go faster, at least in theory. This means the product should be sooner ready to be released in to the market. (Laukkanen, 2006)

Drawbacks

Test automation cannot be used in testing how user-friendly or good looking the application is. This is where manual testing with human touch is still important.

3.3 Mobile testing

"Computer technology changes rapidly. In a blink of an eye the computer went from the desktop to the laptop and now to the handheld mobile device. This migration has changed the way we conduct our lives, businesses, and governments. It has also significantly affected the way software developers and testers do their jobs." (Myers, Sandler, & Badgett, 2012, Chapter 11.)

Testing mobile applications is very challenging compared to other types of software testing. The application in itself might not be the source of the problem. The platforms and environments that are the base where the application is run might just be some of the variables that can cause headaches when mobile testing is considered. Software testers must also take into account the appearance of the operating system, possible interruptions and other problems on the network and the physical differences and hardware configurations of the possible devices that might use the application. All these combined create plenty of different variables and circumstances that will complicate the testing process. These problems might add up and allow new problems to originate making testing an overwhelming process. When all these things are considered, planning the testing might prove to be a challenging feat to perform. (Ibid.)

Difficulties of mobile testing

Myers, Sandler and Badgett (2012, Chapter 11) divide the problems that complicate mobile software testing into four segments: device diversity, carrier network infrastructure, scripting and usability. All of these should be taken into consideration when planning test cases for mobile software testing.

Device diversity

The amount of mobile devices in the world is growing exponentially. A novice in the field of mobile software testing might not even know, how many different kinds of them there are in the world. Different resolutions, sizes of the screens, operating systems, browsers and user interfaces are some of the possible differences that mobile devices might have. Testers should be aware of these variations when designing mobile software testing so that the test cases are as inclusive as possible. (Ibid.)

The amount of mobile devices in the world and the constant growth of that number means that it is impossible to plan and execute testing so that every device on the planet is factored in. However, every device that is left out from the testing procedure might be incompatible with the software when the software is released. This might cause a large number of people to avoid the tested software altogether. (Ibid.)

Network infrastructure of mobile carriers

Mobile devices are usually connected to the internet via a wireless connection which is provided by a network carrier. These wireless connections are not always reliable and occasionally the device might lose its connection to the web. Mobile software testing should be planned with this in mind. Testers should understand how these networks work and what kind of problems they might cause. (Ibid.)

Scripting

Mobile software testing should not be done solely by hand with a real device because especially executing multiple similar test cases might lead to results that are faulty caused by human errors. Furthermore, it will take plenty of time. People tend to make mistakes but that is just the human nature. This problem can be addressed by making automated scripts, that can perform the test cases quickly and without errors. Just a few of years ago it was very problematic to operate the test scripts on real devices, however, luckily the operating systems have developed so that nowadays it is possible with an aid of certain software and without the need for rooting the device or changing it in any way. (Ibid.)

Usability

Testing the usability of the mobile software adds more to the challenge of mobile testing. The testing team has to manually inspect if the tested software works correctly on different platforms and if the appearance of the software is the same as planned. This takes much more time than testing on a desktop because the variety of devices. (Ibid.)

Testing methods

Mobile software testing is somewhat similar to testing internet based applications where it is crucial to take into account different browsers and the possible complications they might create. Mobile applications have similar variables and more. (Ibid.)

When testing the back-end of a mobile software, the testing methods are equivalent to testing methods that are used to test desktop internet-based software. Testers must take care of the data which travels from back-end to front-end and back stays intact and can move without obstruction etc. Also it would be ideal to monitor the stress levels that the software's back-end can handle with appropriate stress tests. (Ibid.)

When testing the part of the software that goes to the end-user, testers must remember what kind of people are going to use it. Also they need to know where and when the software is being used. In addition, testers should be aware of the special situations that might not occur with desktop-based software and how to react to them, such as how the software will function when battery is low, phone is connected to a charger, when phone's memory is limited, the connection to the internet cuts in and out and how the software will react when other features, such as calls or text messages occur. (Myers, Sandler, & Badgett, 2012, Chapter 11.)

Testing platforms

When testing mobile software, one critical decision must be made: whether to test with emulators or with real devices. The selection must be made based on the needs of the project in hand. Both alternatives have their pros and cons that should be taken into consideration. The table below illustrates this. (ibid.)

Table 1. Comparison of emulators and real devices

Real Devices		Emulators	
Advantages	**Disadvantages**	**Advantages**	**Disadvantages**
Ability to test responsiveness of the application	Unable to install metric or diagnostic development tools	Easy to manage; Multiple device support with single emulator	Underlying hardware may skew performance on a real device
Ability to inspect application visually	Possible network problems	Cost-efficient	Inability to indentify device-related bugs
Test carriers network responsiveness	Expensive to use		
Identify device specific bugs			

Testing with real devices

Testing with real devices can be a very time consuming process, especially when scripts and automation are not used. However hands-on-testing with a real device will give the tester the best feeling about the tested software and the experience that the end-user will have. Of course some of the testing can only be done on a real device, e.g. seeing how the wireless internet connection provided by phone-carrier works and how the software reacts to normal actions performed by a mobile phone such as receiving and making calls and text messages. Also when testing is done on a real device, certain bugs that are connected to the particular phone in hand are revealed. (Ibid.)

On the other hand, testing with real devices might be very expensive. The devices must be bought and phone carriers must be paid for the wireless connection to the internet. These payments will multiply when multiple devices are acquired to cover as many test cases as possible. (Ibid.)

Testing with emulators

Testing with emulators is, unlike testing with real devices, cheaper with the added bonus of the possibility to test on different platforms without any added costs of new devices. When starting to test applications, emulators are a great asset that should be utilized. Testing with emulator increases the possibility of finding the biggest bugs and faults from the tested application. (Ibid.)

When emulators are deployed on a high-performance computer, their performance can be enhanced by redirecting some of the computer's power to the emulator. This will accelerate the speed of test execution immensely. Emulators and their configurations can also be changed fast and as many times as it is needed without any additional costs. (Ibid.)

However, emulators do not give the same testing experience as real devices. They are only copies of the software and cannot truly emulate every feature that is found on a real device. Hence, some of the properties that might work as intended on an emulator, might not work correctly on a real mobile phone. (Ibid.)

3.4 Mobile web testing

Nowadays websites should look and feel great and also work well in a desktop environment and on mobile devices, with all different types of configurations of platforms and browsers. Achieving this might take a lot of effort and work hours from the developer. Also, if the website has so severe bugs that the user is prevented from using the site entirely on a certain device, the company that owns the website might lose a large amount of money. That money can be lost from ad revenue, or in a case of an internet store, straight from the sales. When this happens at a busy time of the year when the websites need to be operational, the impact can be devastating and monetary losses massive. Those are a couple of valid reasons why it is very important to test websites to ensure that they work with mobile devices without problems and that the quality meets the industry standards of today. (Warner, Lafontaine. 2010, Chapter 7.)

Approaches to mobile web testing

As with testing mobile applications, mobile web sites can be tested with emulators and real devices. Firing up the browser and opening the website that is to be tested with either of these two will give the tester idea of the current state of the website and expose major flaws that might make the website unusable. Other and arguably faster way to check the responsiveness and operation of the website quickly is to use development tools built in to different browsers. Google Chrome and Mozilla Firefox are examples of these browsers with great development tools. Chrome for example has the ability to resize the browser window to the size of certain mobile devices and mimic the wireless internet connection of the mobile devices by limiting the amount of data it will process slowing the action of the site. (Pettit, 2014)

Other options for mobile web testing is to use paid services found online which help with the testing process. For example one of these is BrowserStack which allows its users to gain access to all of the browsers both desktop and mobile. It also has emulators of multiple mobile devices, multiple desktop OS’s and other pre-installed developer tools. These services work great and they are used by many respectable companies but they often tend to cost very much, which is a sizeable obstacle for startups and for companies that are not willing to spend much money on the mobile web testing. (Ibid.)

3.5 Mobile testing automation

With the help of mobile test automation it is possible to greatly reduce the costs associated with testing and it also helps to improve the test efficiency. To make the testing process work adequately it is important to choose the right tool for the job. (Sathyan, Narayanan, Narayan, & Vallathai 2012, Chapter 9)

Since there are myriad phones in today’s market it is not uncommon for companies to acquire remote service providers who will do the testing. In this way the company does not have to buy many different kind of devices just for testing purposes. (ibid.)

Mobile test automation vs traditional test automation

Mobile test automation has its own set of challenges. There are many different types of mobile devices with differing screen sizes and resolutions. Also mobile phones can be connected to the internet in different ways, such as Wi-Fi and 3G. (Johanson, 2013)

It is important to understand what types of devices an application’s users are using. Test cases should be created in such a way that they have the biggest coverage with as few tests as possible. It is generally a good idea to use a test framework that does not modify the code of the application or root the device. Emulators (for Android) and simulators (for iOS devices) do not reflect the experience that the end-users will experience so it is better to use real devices. (Ibid.)

One issue to consider with mobile test automation is the extensibility of the current test infrastructure. It is better to use something that can be integrated with the current system. (Ibid.)

4 Testing tools

4.1 Software testing automation tools

In order to perform automated testing at its fullest potential it is necessary to implement different tools for it. Those tools allow test engineers to run a huge number of tests which would take a long time if performed manually. This is why manual testing should be kept to a minimum and everything that could be automated, should be automated. (Software Test Automation Tools, N.d.)

Selenium

Selenium is a collection of tools used to help to automate software testing. It is mostly used to test web applications with various testing frameworks, like Robot Framework and JUnit and it can be controlled with multiple different programming languages, including Java, Python, C#. It also supports many different platforms (Windows, OSX, Linux) and major browsers (Internet Explorer, Google Chrome, Mozilla Firefox, Safari, Opera) both for computers and mobile devices. The latter needs the aid of appropriate tools such as Selendroid or Appium to work correctly. (Selenium Documentation, Introduction.)

Of the different software tools that Selenium comes with each of them has a specific role. The users can decide which of those tools suit their needs best and which is the most useful in the project at hand. They all give a different perspective for approaching and solving the problems of software testing automation. (Selenium Documentation, Introduction.)

Selenium IDE

Selenium IDE is an easy-to-use Firefox plugin that makes writing and executing test cases effortless. It is perfect for users that are not experienced with any programming language but still want to become skillful in Selenium commands. (Selenium Documentation, Selenium-IDE.)

The plugin allows the user to record the test case by following the actions made by the user. It then transforms those user-made tasks into a runnable script. That script can be saved as an HTML- file or in another script language. User can then execute those tests whenever he or she pleases. Multiple tests cases can be saved as a test suite which makes executing and maintaining them easier. (Selenium Documentation, Selenium-IDE.)

The recorded tests are not without fault and can also be edited afterwards. User can change the script by altering the values, targets and commands as needed. (Selenium Documentation, Selenium-IDE.)

Selenium Grid

Selenium Grid allows you test multiple automated tests parallel in different machines with different browser combinations at the same time. It is very useful when there is a need to test the cases against many different types of browsers, operating systems and their combinations. (Selenium Documentation, Selenium Grid.)

Selenium Grid consists of a single hub, a master computer of sorts and a bunch of nodes that are connected to the hub. Nodes are either a physical computers or VM's (Virtual Machines). Hub is responsible of distributing the test cases that are assigned to the Selenium Grid, to a node that has the same desired capabilities that the test case has. So, if the test case has a capability that says that it needs to be run on a Windows-machine with Firefox-browser, hub finds a matching node with those capabilities and commands the node to run that test case. Nodes are registered to the hub when they are connected, so the hub knows the exact configuration of its browser-platform. (Selenium Documentation, Selenium Grid.)

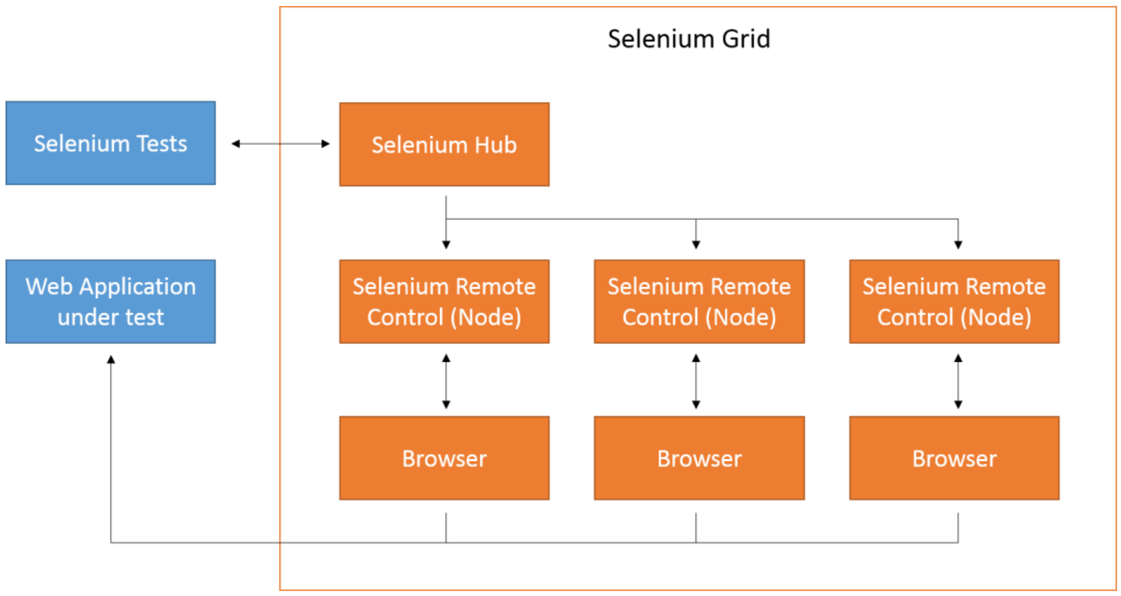

Selenium Grid
Selenium Tests
Selenium Hub
Web Application under test
Selenium Remote Control (Node)
Selenium Remote Control (Node)
Selenium Remote Control (Node)
Browser
Browser
Browser

Figure 3. Selenium Grid (adapted from How it Works, N.d.)

Selenium RC

Selenium RC was part of the first version of the Selenium project and it remained as the main project in developing Selenium until Selenium WebDriver was introduced with Selenium 2.0. Due to that advancement Selenium RC is not developed anymore but is in a supported state. It has features that Selenium 2.0 does not support yet like being able to understand multiple different scripting languages (Java, Python, C# etc.) and support for almost every browser under the sun. (Selenium Documentation, Selenium 1 (Selenium RC).)

Selenium RC consists of Selenium Server and client libraries. Server is the part that is between the test program and the application that is under testing. It receives all the commands sent in by the test program, runs them against the application that is being tested and reports the results back to the user. Client libraries provide the users own test program the ability to communicate with Selenium Server by passing commands and functions to be tested and receiving the results of those test, thus building a working software test automation architecture. (Ibid.)

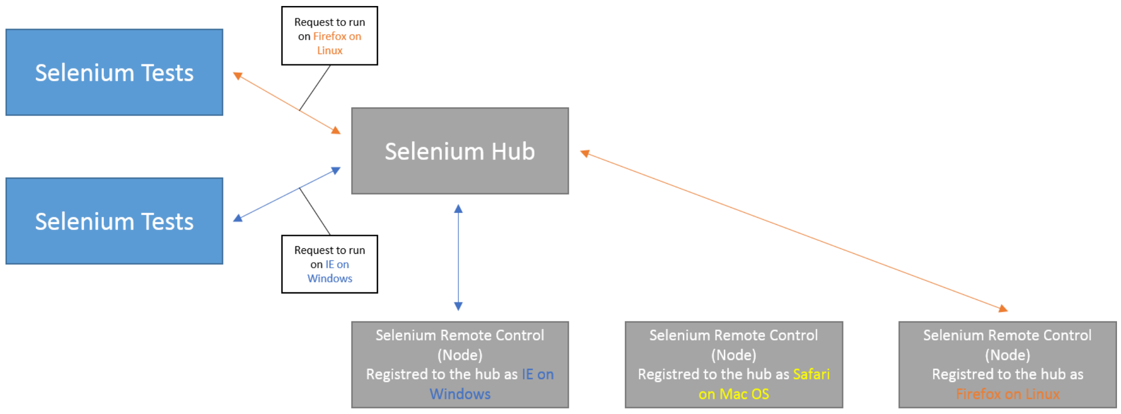

Request to run on Firefox on Linux
Selenium Tests
Selenium Hub
Selenium Tests
Request to run on IE on Windows
Selenium Remote Control (Node) Registred to the hub as IE on Windows
Selenium Remote Control (Node) Registred to the hub as Safari on Mac OS
Selenium Remote Control (Node) Registred to the hub as Firefox on Linux

Figure 4. Selenium Grid in action (adapted from How it Works, N.d.)

Selenium WebDriver

WebDriver was a new feature that was integrated to Selenium when the version 2.0 was released. It was developed as an answer to the limitations of Selenium-RC. It provides simpler programming interface and solutions for testing modern, dynamic web-apps. (Selenium Documentation, Selenium WebDriver)

Unlike Selenium RC, WebDriver does not need Selenium Server in order to work correctly. It uses the browser's native support for automation, unlike Selenium RC which delivered specific JavaScript functions from a certain library to the browser to be driven within the browser with more JavaScript. This makes the executing of the tests faster, however, with less accurate results. (Ibid.)

Robot Framework

Robot Framework is a framework for generic test automation that is used for acceptance testing and ATDD. It uses keywords for its testing approach and a tabular test data syntax that is very easy to learn and use. This syntax is used to create the test cases and the high-level keywords. It can be extended by using testing libraries which are implemented with either Java or Python. (Robot Framework User Guide Version 2.8.7.)

The framework itself is based on Python -programming language. It can also run Jython, which is based on Java and IronPython, which is based on .NET. It is released under Apache License 2.0 and its development is supported by Nokia Networks. (Ibid.)

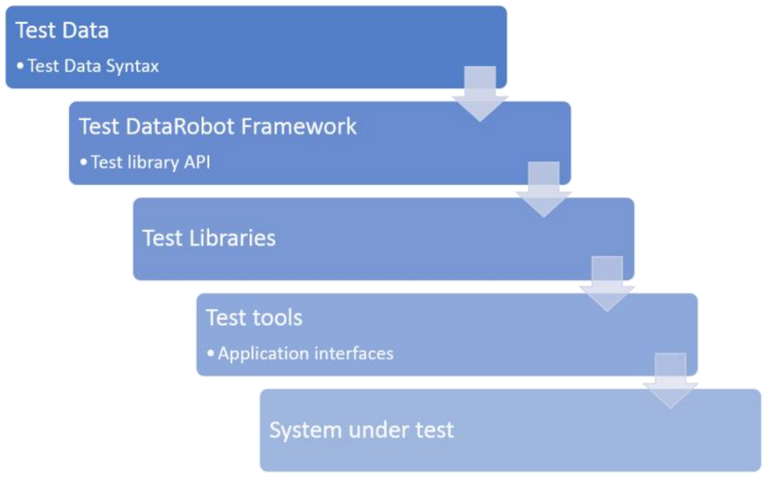


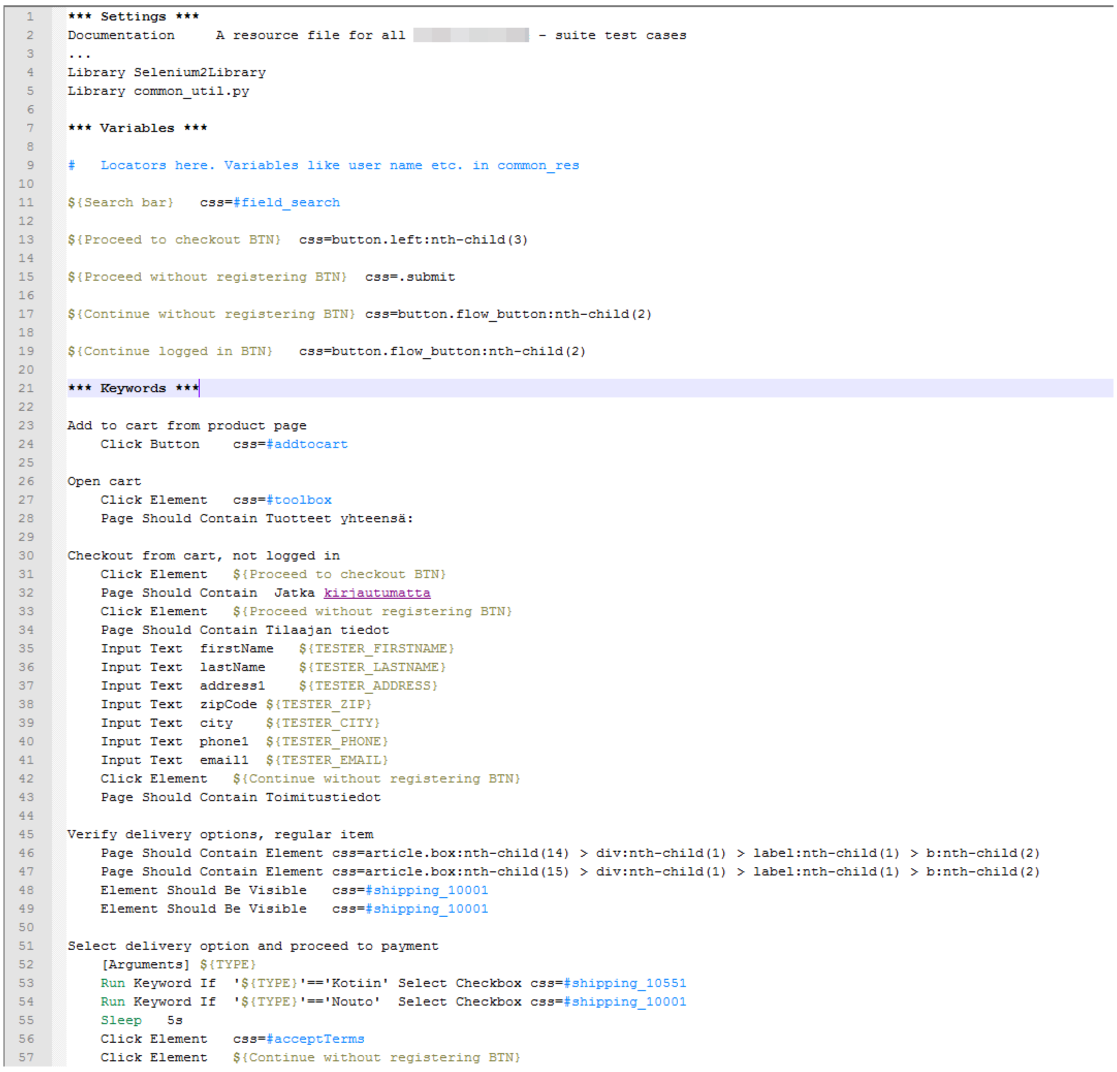
Figure 5. Robot Framework's infrastructure

Test Cases

Robot Frameworks tests are done in a tabular format. There are four different formats: HTML, TSV, reST and plain text. Every one of these has its own advantages and disadvantages, however, a plain text file is generally recommended. (Robot Framework User Guide Version 2.8.7.)

When a single text file contains multiple test cases it automatically becomes a test suite (ibid.).

When defining test data tables (e.g. Setting, Variables) at least one asterisk must be put before the name. Usually it is set like this: ***Variables***, however, *Variables works just as well. (Ibid.).



```
*** Settings ***
Documentation     A resource file for all [redacted] - suite test cases
...
Library Selenium2Library
Library common_util.py

*** Variables ***

#   Locators here. Variables like user name etc. in common_res

${Search bar}    css=#field_search

${Proceed to checkout BTN}  css=button.left:nth-child(3)

${Proceed without registering BTN}  css=.submit

${Continue without registering BTN} css=button.flow_button:nth-child(2)

${Continue logged in BTN}   css=button.flow_button:nth-child(2)

*** Keywords ***

Add to cart from product page
    Click Button    css=#addtocart

Open cart
    Click Element   css=#toolbox
    Page Should Contain Tuotteet yhteensä:

Checkout from cart, not logged in
    Click Element   ${Proceed to checkout BTN}
    Page Should Contain  Jatka kirjautumatta
    Click Element   ${Proceed without registering BTN}
    Page Should Contain Tilaajan tiedot
    Input Text  firstName   ${TESTER_FIRSTNAME}
    Input Text  lastName    ${TESTER_LASTNAME}
    Input Text  address1    ${TESTER_ADDRESS}
    Input Text  zipCode ${TESTER_ZIP}
    Input Text  city    ${TESTER_CITY}
    Input Text  phone1  ${TESTER_PHONE}
    Input Text  email1  ${TESTER_EMAIL}
    Click Element   ${Continue without registering BTN}
    Page Should Contain Toimitustiedot

Verify delivery options, regular item
    Page Should Contain Element css=article.box:nth-child(14) > div:nth-child(1) > label:nth-child(1) > b:nth-child(2)
    Page Should Contain Element css=article.box:nth-child(15) > div:nth-child(1) > label:nth-child(1) > b:nth-child(2)
    Element Should Be Visible   css=#shipping_10001
    Element Should Be Visible   css=#shipping_10001

Select delivery option and proceed to payment
    [Arguments] ${TYPE}
    Run Keyword If  '${TYPE}'=='Kotiin' Select Checkbox css=#shipping_10551
    Run Keyword If  '${TYPE}'=='Nouto'  Select Checkbox css=#shipping_10001
    Sleep   5s
    Click Element   css=#acceptTerms
    Click Element   ${Continue without registering BTN}
```

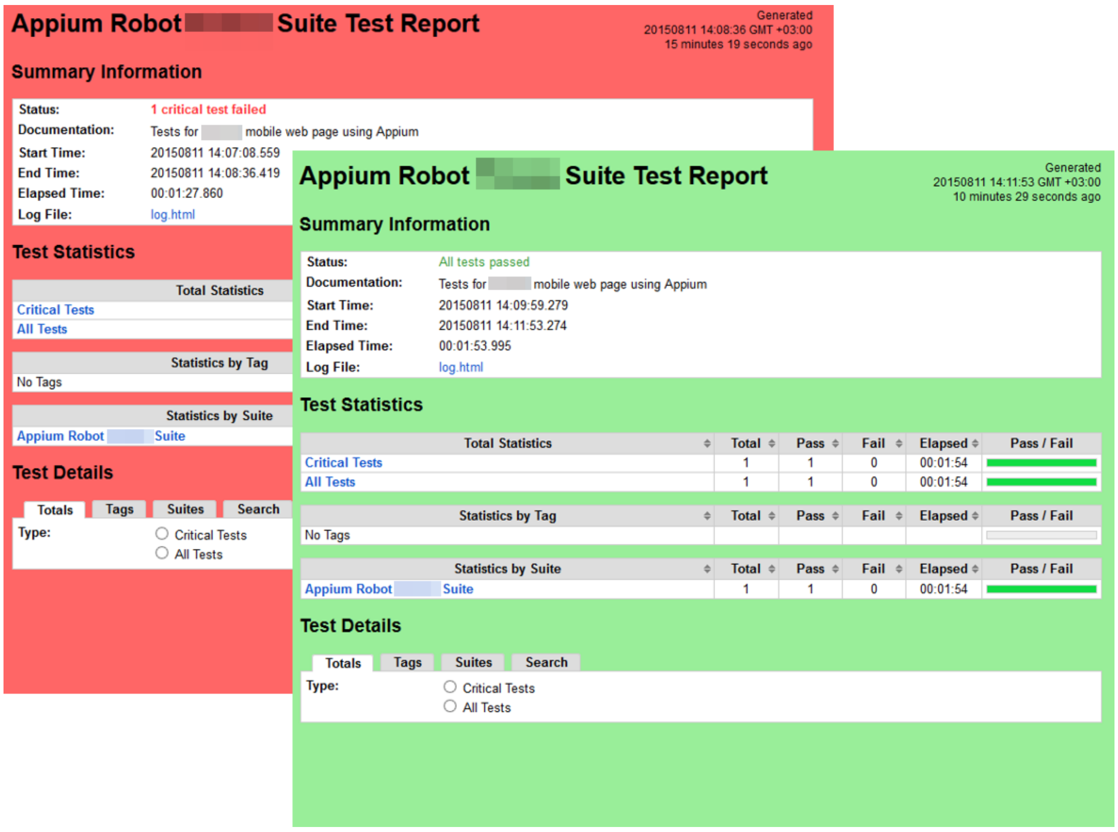
Figure 6. Robot Framework example resource.txt for test suite

Test execution

Robot Framework tests are normally run using pybot, jybot or ipybot script using Python, Jython and IronPython respectively. To run a suite called test.txt with Robot Framework type *pybot test.txt* to the Windows Command Prompt. This assumes you have navigated to the folder which contains the test suite. Alternatively *pybot pathtotest/test.txt* can be written. (Robot Framework User Guide Version 2.8.7.)

Test Output

After executing the tests Robot Framework creates three different result files: output.xml, log.html and report.html. Output.xml has the results in XML format. Log.html is probably the most important of the three when the test results need to be examined in detail since it contains detailed info about the executed tests. Report.html is a general overview of the executed test(s) and has color coding. As can be seen below (Figure 5.), if the background is green, the test has passed and if it is red it has failed. Report.html has convenient links to log.html if and when more detailed info is needed. (Ibid.)



Appium Robot [redacted] Suite Test Report

Generated
20150811 14:08:36 GMT +03:00
15 minutes 19 seconds ago

Summary Information

Status:	1 critical test failed
Documentation:	Tests for [redacted] mobile web page using Appium
Start Time:	20150811 14:07:08.559
End Time:	20150811 14:08:36.419
Elapsed Time:	00:01:27.860
Log File:	log.html

Test Statistics

Total Statistics
Critical Tests
All Tests

Statistics by Tag
No Tags

Statistics by Suite
Appium Robot [redacted] Suite

Test Details

Totals Tags Suites Search

Type: ○ Critical Tests ○ All Tests

Appium Robot [redacted] Suite Test Report

Generated
20150811 14:11:53 GMT +03:00
10 minutes 29 seconds ago

Summary Information

Status:	All tests passed
Documentation:	Tests for [redacted] mobile web page using Appium
Start Time:	20150811 14:09:59.279
End Time:	20150811 14:11:53.274
Elapsed Time:	00:01:53.995
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:01:54	🟩
All Tests	1	1	0	00:01:54	🟩

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Appium Robot [redacted] Suite	1	1	0	00:01:54	🟩

Test Details

Totals Tags Suites Search

Type: ○ Critical Tests ○ All Tests

Figure 7. Robot Framework test suite reports

Selenium2Library

Robot Framework has a web testing library called Selenium2Library which is based on the Selenium 2 and WebDriver. Most modern browsers are supported. Tests are run in a real browser. When Selenium2Library is to be used in a test case it must imported into the test suite. (Tomac, 2015)

AppiumLibrary

Robot Framework has its own Appium testing library called AppiumLibrary. It requires at least Python 2.0. When developers want to use AppiumLibrary it must imported into the Robot Framework test suite. (Chang, 2015)

4.2 Mobile testing automation tools

According to Top 10 Mobile Testing Tools (2015), the most popular, preferably multi-platform, test tools were chosen to be inspected here and later on in more detail in chapter six. Also, eggPlant was included since it seemed very popular.

Appium

Appium is a tool that can be used to automate tests for native, mobile web and hybrid applications on Android, iOS and FirefoxOS platforms. Appium can be used to test on simulators (iOS, FirefoxOS), emulators (Android) and on real devices (iOS, Android). It is released as an open-source project and is primarily supported by Sauce Labs, which has designed most of Appium software, graphics and is responsible of majority of Appium community management. (About Appium; Appium Sponsors.)

Native applications are applications that are written specifically for either Android or for iOS. Hybrid applications are equipped with a wrapper around so called "webview", which enables a native app to communicate with content on the web. These include projects which are made with for example PhoneGap. (About Appium.)

Reasons to use Appium

Appium gives developers freedom to use whatever development tools they desire and whatever scripting language they are familiar with. The only restriction is that it should be compatible with Selenium's WebDriver. These languages include Ruby, Python, Java, JavaScript, PHP, C# etc. Developers are also free to use any testing framework they please. (Ibid.)

Also, applications do not need to be recompiled or modified in any way when they are tested with Appium. This is achieved by using standardized automation APIs on all platforms. (Ibid.)

How Appium works

Appium uses different automation frameworks for all the mobile platforms. This removes the need to compile in any code or frameworks outside of Appium’s own to the tested app. This way the tested application stays the same and does not get changed in any way. The provided frameworks are:

- Android 2.3+: Instrumentation made by Google
- Android 4.2+: UiAutomator made by Google
- iOS: UiAutomation made by Apple

These frameworks are wrapped in Selenium WebDriver API. This API uses specific client-server protocol called JSON Wire Protocol. Using this protocol server can be paired with a client that is written in any language. This client can then communicate with server with HTTP requests. Appium and WebDriver client work together as automation libraries and not as conventional testing framework. This allows users to use any testing framework they want. (Ibid.)

Selendroid

Mobile application and mobile web testing is becoming more and more important in today’s society. Selendroid is a framework suited for this purpose. Tests are done using Selenium 2 client API so people who are already familiar with Selenium should have no problem using Selendroid. Selendroid utilizes JSON wire protocol. Unlike some other test frameworks Selendroid does not need to modify the application under test. Selendroid works with emulators and real devices. Selendroid works with Selenium Grid (more on this below). (Dary, & Palots N.d)

Selendroid requires at least Java SDK 1.6. Also, JAVA_HOME Environment variable needs to be set. In addition, Android-SDK is required and ANDROID_HOME Environment variable must be set. (ibid.)

Android device must be plugged in to the computer which has the selendroid-standalone running (ibid.).

Selendroid Architecture

Selendroid has four major components: Selendroid-Client, Selendroid-Server, AndroidDriver-App and Selendroid-Standalone. For the automation the most important component is the Selendroid-Server. (Selendroid, Selendroid's Architecture)

Robotium

Robotium is an open source testing automation framework, designed for Android native and hybrid applications. It was founded and developed by Redas Rana and it is hosted under Google code. It is used to write very rigid black-box UI tests easily and fast. Robotium handles multiple Android activities so the tester can write acceptance and system tests that span them. These tests can be run on emulators and real devices. (User scenario testing for Android.)

MonkeyTalk

MonkeyTalk is a testing tool that can be used to record very simple and manageable test scripts and play them back. It is a cross-platform tool that supports iOS and Android applications, hybrid applications and mobile web applications. It is designed to be simple to use, so even people with only a little experience with testing or writing test scripts can start to use it comfortably. There are two versions of MonkeyTalk available: Community Edition which has the basic scripting and editing functionality and Professional

Edition which extends the tool with extended reporting, automated end-to-end workflow and possibility to connect and integrate with CloudMonkey LabManager. (About MonkeyTalk Platform)

eggPlant

eggPlant is a GUI driven test tool developed by TestPlant. It offers capability to test any kind of programs and mobile applications without OS and device limitations. (TestPlant.)

Testing takes place inside the computers firewall and there is no need to install anything to the devices under test. Also there is no need to modify the tested software in any way in order to make eggPlant to work correctly. (ibid.)

eggPlant is able to see the view that is on the device under test. This view is scanned by an algorithm that is designed to recognize images. This algorithm can be taught to spot any differences that occur in the supposed view. It can identify colors, work in a dynamic environment habited by for example Flash or Silverlight based technologies. When eggPlant spots an error, it captures a screenshot from the situation and stores it with an error log to help developers find the bug that caused the initial fault. (Ibid.)

eggPlant is designed to be easy-to-use testing tool. It can be used by user that might not be very experienced with testing tools or testing in general. It produces script in "SenseTalk" language which is easy to interpret and writing it does not necessarily need any earlier scripting experience. (Ibid.)

Calabash

Calabash is a testing tool for UI acceptance testing, used to test Android and iOS applications. Calabash is free, open source tool which is developed and updated by Xamarin. Tests in Calabash are written using Cucumber testing framework. Calabash works as a bridge providing Cucumber tests to be run against the tested software. Cucumber provides easy-to-understand test script syntax that can be written and read by users that might not be very technically advanced. (Introduction to Calabash)

Calabash can be integrated with Test Cloud, a paid service offered by Xamarin that offers the possibility to run Calabash tests through hundreds of different real devices each configured differently. Tests run on Test Cloud can be added as a step in a continuous integration system which allows tests to be executed when the source code is expanded or altered giving the developer feedback instantly if errors or bugs are encountered. (Ibid.)

5 Continuous integration

5.1 What is it?

Continuous integration is a software development practice that promotes a way of producing better quality software by making the developers integrate a little bit of software continuously to the project. This means that software development teams also have to keep track of the quality of their output. Continuous integration exposes possible flaws early in the development cycle and in a small scale before the software is finished, making the fixing of the problems easier and faster. If the bugs survive to the production version of the software, rooting them out requires more effort and is likely to cost more money. (Berg 2012, 1.)

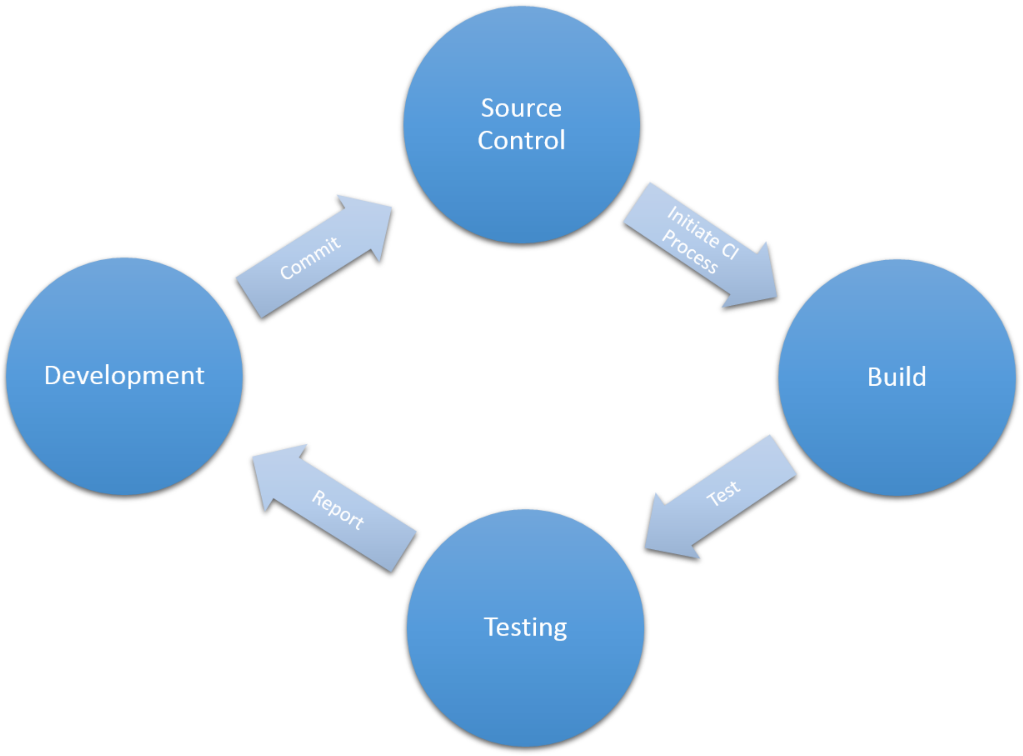


Figure 8. Basic continuous integration principle (adapted from Continuous Stories, N.d.)

5.2 Benefits

The most prominent benefit of using continuous integration in a software project is reducing the risks. Unlike in deferred integration, continuous integration helps to understand how much time it will take to do something and how much has been done already. Development teams are constantly aware of the bugs that are still present in the software and know the ins and outs of it. (Fowler, 2006.)

Although continuous integration itself does not remove any bugs, it helps tremendously in finding them. Since the software is only changed a small increment at a time, it is easy to track the cause of a new bug. This way there will be no overwhelming amounts of bugs that are difficult to root out, due to their interactions with each other. Developers are also mentally in a better state, meaning more confident and motivated, when there are only a few bugs present. (Ibid.)

When using continuous integration frequent development is possible. Frequent development allows developers to publish a new version of the software frequently, thus giving users the possibility to have new and improved software in their hands more often. This way they can comment and give more feedback on new features, increasing the quality of the software and allowing customers and developers to get on the same page on what is required from the software. (Ibid.)

5.3 Best practices

Certain guidelines are needed to get continuous integration to work effortlessly in a development environment. Fowler (2006) describes following list of practices that are useful and effective while implementing continuous integration:

- Using a Single Source Repository
- Automated Builds

- Self-Testing Builds
- Commit To the Mainline Every Day
- Broken Build Must Be Fixed Immediately
- Build Fast
- Do Testing in a Clone of the Production Environment
- Anyone Should Be Able to Get Latest Version Easily
- Keep Development Visible for Everyone
- Deployment Should Be Automated

5.4 Jenkins

Jenkins is a Java based continuous integration server tool that increases the speed of software development with the aid of automation. It has the ability to manage different kind of development procedures like builds, deployments, documentation and tests. Jenkins can be paired with version control system e.g. Git or Mercurial so it can for example keep track of any changes that are made to the code and act accordingly either by running tests against this new, changed version of the software or by doing something else it has been told to do. It can also run shell scripts and Windows batch commands. Jenkins supports community-made plugins that extends its repertoire of actions and lets it link with many of today’s widely used technologies. (Vogel 2015.)

Jenkins was created by Kohsuke Kawaguchi in 2004. It was originally forked from a project called Hudson which was owned by Oracle after a dispute between the parties involved in the project. (Ibid.)

Installation

The easiest way to install Jenkins is to download a native package from Jenkins homepage, www.jenkins-ci.org. There are many different versions of the native package for different operating systems like Windows, Mac OS X, Ubuntu, Red Hat. Also there are versions for lesser known platforms like openSUSE, FreeBSD, OpenBSD and Gentoo. (Vogel 2015.)

Another way to install Jenkins is to download a WAR file from the same page as the native packages and start it from command line with *java -jar jenkins*.war* (ibid.).

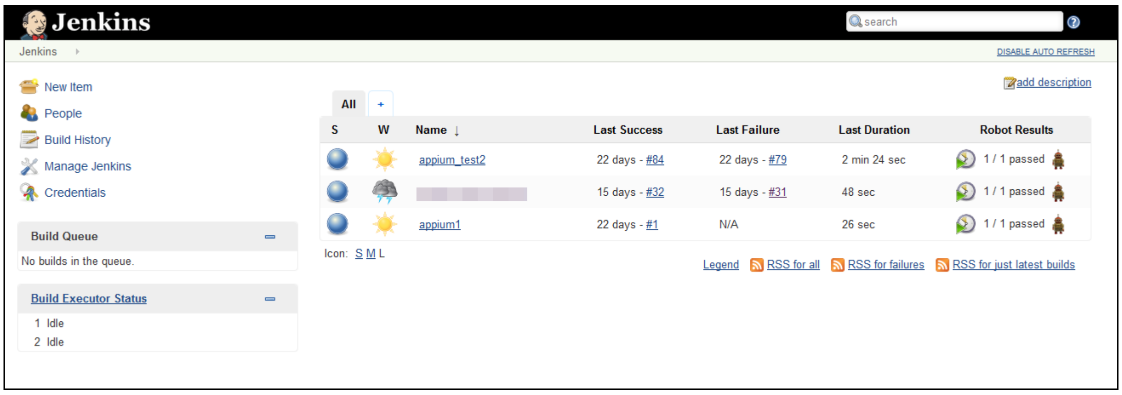
After the installation process Jenkins will start under http://localhost:8080/ if it was started locally (ibid.).

There is also a so called Jenkins LTS release which is a more stable version of Jenkins. LTS version gets released less often and with fewer changes than the normal version of Jenkins which gets weekly updates and bug fixes. Only important and major bugs are fixed on the LTS version. This version is great for people who want a more reliable version of Jenkins with as few bells and whistles as possible. Installation of the LTS version is the same as the non-LTS version. (Kawaguchi 2015.)

Configuration

Most of the configuration that needs to be done for Jenkins can be done from its web-based interface. After installation Jenkins runs with default configurations. It is very important to secure Jenkins by at least declaring some restriction for different user types. Users that are not registered to a Jenkins server are anonymous. It is recommended to change their access and rights to "read-only". This reduces the risk of misuse of the Jenkins server. (Vogel 2015.)

User with "administrator" credentials can add plugins to Jenkins. They enhance the functionality of Jenkins adding more features to it. (Ibid.)


Jenkins

search

Jenkins

DISABLE AUTO REFRESH

New Item
People
Build History
Manage Jenkins
Credentials

add description

Build Queue

No builds in the queue.

Build Executor Status

1 Idle
2 Idle

All +

S	W	Name ↓	Last Success	Last Failure	Last Duration	Robot Results
●	☀	appium_test2	22 days - #84	22 days - #79	2 min 24 sec	1 / 1 passed
●	⛈	[redacted]	15 days - #32	15 days - #31	48 sec	1 / 1 passed
●	☀	appium1	22 days - #1	N/A	26 sec	1 / 1 passed

Icon: S M L

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

Figure 9. Jenkins dashboard

Jenkins jobs

Software projects are added to Jenkins by making new jobs in the web interface. These jobs can execute different steps of the software development like running unit tests or generate documentations. Usually multiple jobs are used to do all the tasks needed in the whole software project. Jenkins supports multiple types of jobs all equipped with different kinds of unique properties. (Vogel 2015.)

The "free-style software project" jobs are general, all purpose jobs that can perform many different actions. These include doing different types of builds, running tests or executing repetitive batch tasks. These types of jobs are not limited to a certain SCM. (Ibid.)

Jenkins is also able to do a job dedicated to Maven. These Maven jobs are used with projects that use Apache Maven that is a project management tool. It is built around a POM. This XML file contains all the information needed that Maven uses to build the project. Using Maven job Jenkins can directly access the POM and take advantage of it. This reduces configuration needed to run the jobs massively. (Welcome to Apache Maven.)

Multi-configuration job is suitable for projects where for example builds will produce many similar build steps. It allows user to run builds with multiple different configurations. These might include testing on multiple different environments with different databases. It is even possible to build with different machines altogether. (Kawaguchi, 2015.)

It is also possible to hook Jenkins up with projects that run outside of Jenkins with external jobs. That project can even be on a remote machine. This way Jenkins can be used as a dashboard for existing automation systems. (Ibid.)

Plugins

Jenkins has a strong core that consists of different elements and components. Everything cannot be supported, so Jenkins supports a plethora of plugin all made to extend its usability and features. As Jenkins is an open source project, these plugins can be made by anyone. Plugins can easily be installed from the web interface. (Ibid.)

6 Choosing the mobile test automation tool

6.1 Background

Different kinds of tools made for mobile testing automation have sprung into the market in recent years, all made to compensate distinct aspects and deficiencies of existing mobile testing techniques. Choosing the right tool from this colorful cavalcade of testing tools is very important, so that the required testing can be executed correctly and without flaws. The tool should be compatible with the tested software and with the testing environment and practices already in use. However, every tool has its drawbacks and properties that does not quite fit to the project at hand. This should be kept in mind when choosing the right tool for the job.

The process of choosing the right testing automation tool should not be a hasty task. On the contrary, it should be a deliberate and well-thought process. This is important because in the future it is time consuming and costly to change the practices that are generated when the tools are introduced for the first time. Many companies are not ready for that kind of hassle, therefore, as always when introducing new methods and tools the choice should be made with future in mind.

In this research, the eleven viewpoints introduced in TestHuddle (2014) article and in TestLab4apps (2014) article were used in order to help to decide, which mobile automation tool is best suited for the needs of Descom. The testing environment and the starting point for the thesis, in the point of view of the tools and technologies used in Descom, is described in the Appendix 1. The features found out by studying those different mobile automation tools were compared with the aid of the eleven viewpoints to the criteria obtained from Descom. Below is more information about the eleven steps used in this research.

6.2 Initial selection process

Initially, many different mobile testing automation tools were taken in to consideration but after inspecting the tools on the market, six were taken in to closer examination. Reason for picking these six were their popularity and the amount of support that they are given, which is huge compared to other, smaller tools. There are still some tools floating around on the web that are not supported in any way and are thus highly unstable and prone to have a lot of unwanted features that might hinder the testing experience.

Also, some tools like KIF that might initially look very appealing turn out to only support iOS. This is a very counterproductive quality, as the thesis does not focus on testing only on iOS and the authors do not have the necessary equipment to perform it at the time of the writing.

6.3 Selection steps

The eleven steps used in defining the right mobile testing automation tool for this case are listed below as follows.

1 Supported mobile platforms

There are many different operating systems for mobile devices. The support for these different platforms varies a great deal concerning tools used in automated mobile testing. When choosing a right tool for certain company or for a certain project, it is important to find out which operating systems, like iOS, Android or Windows, the tool under inspection supports. It is also very important to figure out which versions of these operating systems are supported.

2 Supported application types

It is also mandatory to find out what kind of application types these mobile testing automation tools support. There are three types of applications: native, web-based and hybrid. Most of the tools support only some of those application types, not all of them. Because of this reason, it might be necessary to use multiple different tools in tandem in order to test many different types of applications. However there are a few exceptions on the market nowadays which do support all of the application types.

3 Source code requirements

Test engineers cannot always access the source code of the application under test which sets some restrictions for the tools used in mobile automation testing. For example, when testing an iOS application the test engineer might be able to get hold of a so called "app package" instead of the source code. This app package gives better testing coverage than just using the installable version of the software but pales in comparison with the specificity of testing with the source code. Thus it is important to find out, if the testing tool needs the source code of the application in order to work as intended.

4 Application refactoring requirements

When the source code requirements have been analyzed, the next step is to check if the testing tool under inspection requires some kind of modifications to the applications that are tested with it. Different tools need varying amounts of application refactoring in order to work correctly. In some cases an external library has to be implemented to the project and a new build has to be made just for testing purposes. Some tools do not need any refactoring. Most demanding tools necessitate that the source code of the application needs to be modified.

5 Test scripts generation

Testing should be made easier and faster when automation is added to it. It also should take the unnecessary work out of testing engineers’ hands. When this principle is followed, creating test scripts should not be an overly complicated or too much time consuming phase of testing. Using a simple automated test script generation by recording users’ actions is a great way to achieve a fine test coverage with a relatively small amount of work. These automatically created scripts should support parameterization, meaning that they could be altered and changed by hand in the future. This way the scripts scale well and are re-usable. On the other hand, these automatically created scripts are not the most accurate and some of the tools do not support that feature at all.

Another alternative is to write the test scripts by hand using the scripting languages supported by the testing automation tool. These hand-made scripts demand more work but they turn out to be more flexible and more modifiable. Choosing between these two methods of producing test scripts depends on the resources that are allocated to this task and what is possible in the scope of the chosen mobile testing automation tool.

6 Programming language specifications

The scripting language used with the mobile testing automation tools should be compatible with the working environment where it is being implemented. If the test engineers are familiar, for example with Python or Java, it is reasonable to gravitate towards a tool that supports those scripting languages. Learning a new scripting language is a tedious job. It might take a lot of time and eat away the resources that are poured to automating the testing. In some cases the architecture of the application might conduct the choice of the scripting language; if the application is written using an object-oriented language it is advisable to use similar language for the test scripts.

7 Runtime object recognition

One of the key features that must be clarified when considering a tool for mobile testing automation is the way how it recognizes objects during runtime and how it handles them. It is also important to examine how easy it is to maintain object recognition in object library. Unique object identification decreases the impact of changes and mitigates the difficulty of maintaining the test scripts. This is a very nice feature that makes test engineer’s job much easier.

8 Data driven inputs

Nowadays software is more and more interactive and they require users to give different types of inputs. Test engineer should be aware of how the software behaves when it is given many different kinds of inputs. In testing phase these different inputs should be imported from some external source, for example from database or CSV file. This is better practice than hard coding test scripts or manually giving the inputs. The testing infrastructure should be able to add this automatically generated data to the inputs, so that the combinations change. It also should be able to respond to these variations accordingly. This method expands the test coverage by huge amount and reduces the need for repetition.

These so called data-driven tests test the limits of the inputs and inputs which are incorrectly set. To ensure that these tests work correctly the data source needs to be properly created and kept up-to-date.

9 Result and error logging

The test result should be clearly displayed in the chosen mobile test automation tool. The most important thing is to see clearly, if the test was passed or failed. Also, every bit of additional info that can help with tracking down the errors and fixing them is important. For example, screenshots or videos can help with finding the errors and by utilizing those the results should be better, in general. The chosen tool should also log every possible error clearly and precisely. The person analyzing the report should be able to filter the report by time, priority, text or type. The possibility of changing the format of the report is also a good additional feature.

10 Continuous integration

Mobile test automation tool (like normal test automation tool) should support teamwork and continuous integration and its required components. These include but are not limited to IDEs, test frameworks, version control systems and issue tracking software. Making use of the continuous integration adds to the quality of products and leaving it out from test planning and test execution can cause a lot of problems when software is developed further. The tool should be able to run chosen tests automatically when there is new build and also to time the tests to run at a specific time. Some other important features include breaking down the test suites in to smaller chunks, running tests in parallel and when problems arise the developers should receive the reports automatically.

11 Pricing model

There are open-source and paid mobile test automation tools. When choosing an open source tool it is of utmost importance to validate how actively it is supported and developed. This way it can be ascertained that the chosen tool is long-living. In paid tools it is important to study the pricing models. There are different kinds of licensing models, for example pay per use, pay per node and period of validity. In addition, it must be checked if possibly necessary add-ons are paid. Usually there are trial versions available for these paid tools; it is good idea to download these and test if the chosen tool works for this specific case.

It is also important to review the automation tool by its ease-of-use. The tool's complexity should correspond to the know-how of the staff performing the testing. Training takes a lot of time and money from the company if the chosen mobile test automation tool is really intricate.

6.4 Tool choices and comparison

Six different mobile test automation tools were chosen to the comparison: Appium, Selendroid, Robotium, MonkeyTalk, eggPlant and Calabash. Below is a table that shows the differences of the mobile test automation tools when compared to the eleven, previously discussed steps.

Table 2. Comparison of the mobile test automation tools

	Appium	**Selendroid**	**Robotium**	**MonkeyTalk**	**eggPlant**	**Calabash**
Supported platforms	iOS, Android & FirefoxOS	Android	Android	iOS & Android	iOS, Android, BlackBerry & Windows Phone	iOS & Android
Supported application types	Native, hybrid & web	Native, hybrid & web	Native & hybrid	Native, hybrid & web	Native, hybrid & web	Native (hybrid also possible with libraries)
Access to the source code needed	No	No	No	No (except the first time AUT is built)	No	No
Application needs to be modified	No	No	No	No (except the first time AUT is built)	No	Yes
Test scripts generation	Manual, UI	Manual, Selendroid inspector	By hand or with Robotium Recorder	MonkeyIDE record feature or scripts in any IDE	eggPlant IDE for capturing actions, manual scripts	No keyword support
Programming language specificati	Python, Java, PHP, RF keywords etc.	Python, Java, C# etc.	Java	MonkeyTalk language, Java and JavaScript	SenseTalk®	Gherkin
Supports true object recognition	Yes	Yes	Yes	Yes	No	No
Supports data driven inputs	Yes	Yes	Excel files	CSV files	CSV, txt and XML files	CSV & XLS files
Result and error logging	Errors log to Appiums own console	Erros are presented in terminal	Goes to .xml file, with APIs to other formats	Screenshots and HTML files	Results are logged to .txt and to couple of .csv files	IDE shows the results
Continuous integration	Yes (for example Jenkins)	Yes (for example Jenkins)	Yes (for example Jenkins)	Yes (for example Jenkins)	Yes (for example Jenkins)	Yes (for example Jenkins)
Pricing model	Open-source	Free (Apache 2.0 License)	Free (Apache 2.0 License)	Open-source (professiona l edition costs)	License must be bought for fixed time	Open-source

Appium

Appium supports iOS, Android and FirefoxOS. It supports all types of applications (native apps, mobile web apps and hybrid apps). Appium does not require an access to the source code in order to run the tests. Also, the AUT does not need to be modified.

Test scripts can be written manually. It is also possible to use community created program called appium.exe (or appium.app on Mac OS) which can record actions and these can be exported to programming language of choice.

Supported languages include Ruby, Python, Java, JavaScript, Objective C, PHP, C# and Robot Framework’s own keywords. Appium has true object recognition. On Android objects can be recognized by element type, accessibility label, hierarchy and Android ID. IOS has the same ones with the exception of Android ID. Appium supports data driven inputs.

Appium error messages are displayed in the terminal. If Appium is running from the Appium.exe the error messages will be displayed there. Appium supports continuous integration, for example Jenkins. Appium is open-source software so it is completely free.

Selendroid

Selendroid is for Android platform only. It supports native Android apps, hybrid apps and mobile web apps. Selendroid does not require an access to the source code of the AUT and, also, no modification of the AUT is needed either in order to run the tests.

Tests are usually generated manually; however, it is also possible to use Selendroid Inspector to create test cases. It contains a feature which attempts to see which element was clicked. Selendroid supports the same languages as Selenium client so C#, Java and Python will work, for example. As Selendroid is based on Selenium it also supports the same object recognition methods. It depends on the programming language. Errors are displayed in the terminal if the tests are run from there.

Selendroid supports CI so it will work with Jenkins, for example. It is under Apache 2.0 License so it is free to use and modify as one sees fit.

Robotium

Robotium is Android test automation framework. Supports native Android apps and hybrid apps. Robotium does not need access to the source of the AUT. No modification of the AUT is required.

Test scripts can be generated by hand or with Robotium Recorder which is a paid software designed to ease the scripting process. Robotium tests are created using Java. Robotium has true object recognition. Robotium supports data driven inputting to a certain point, for example Excel file can be used to store different variables and parameters.

With little configuration, test results can be logged in to .xml file. Also, third party APIs can be used to generate results in to different formats, like HTML. Robotium supports Maven, Gradle and Ant and integrates well with Jenkins. Robotium is released under Apache 2.0 License.

MonkeyTalk

MonkeyTalk supports Android and iOS testing. Android version 2.2 (or greater) is required. IOS requires version 4.0. Native apps, hybrid apps and mobile web can be tested using MonkeyTalk. MonkeyTalk does not need source code access to run tests. However, the first time the AUT is built it requires source code access because the MonkeyTalk Agent must be installed (see below).

In order to run the tests MonkeyTalk Agent must be installed to the application during the build process. After this has been done the tests can be run without access to the source code. There should be two different copies of the app: the one that has the MonkeyTalk Agent installed and the one that is released.

MonkeyTalk tests are created using the MonkeyTalk IDE's record feature. Test scripts can also be written in any IDE and run in Ant runner or Java runner. Scripts are written in MonkeyTalk language. More complicated test cases can be done using JavaScript or Java. MonkeyTalk has true object recognition both for iOS and Android.

MonkeyTalk supports CSV files that can be created with any kind of spreadsheet tool. Test results are shown in summaries that provide helpful screenshots. With those it easy to find out why some particular test step has failed. Tests are formatted in easy-to-read HTML files. MonkeyTalk also supports xUnit standards.

MonkeyTalk supports most CI tools. Jenkins, for example, works great. MonkeyTalk is open-source so it is free. There is also a professional edition of the software that's not free.

eggPlant

eggPlant supports Android, BlackBerry, iOS and Windows Phone. It can be used to test native apps, hybrid apps and mobile web. No source code is needed to perform tests using eggPlant. Application does not need to be modified.

eggPlant IDE to capture actions or manual scripts to write test cases. Scripts for eggPlant are created using SenseTalk®. True object recognition is not supported; instead eggPlant uses image based system.

CSV, txt and XML file types are supported. eggPlant logs its results to a .txt file and into a couple .csv files. These files can be accessed through eggPlant IDE or manually.

eggPlant has a number of plug-ins and they are collected under the name eggIntegration. With them it is possible to integrate eggPlant with Jenkins, IBM UrbanCode Deploy and IBM Rational Quality Manager, for example. License for eggPlant is bought for specific amount of time. Its cost depends on whether one person or whole team uses it. With the license an access to training is also granted.

Calabash

Calabash supports Android and iOS. Calabash is for native applications but it is also possible to test hybrid apps with the help of libraries: JavaScript and Ruby. Source code access is not required in order to run the tests. In order to run Calabash tests the AUT requires some changes. It's good practice to create duplicate of your application just for test running purposes.

Keyword-driven testing is not possible in Calabash. Programming language called Gherkin is used to write Calabash tests. Gherkin is easier for less technically savvy people to understand because of its natural sounding syntax. Calabash does not support true object recognition. Web elements can be identified using JavaScript for queries.

Since Calabash can implement Ruby it can use CSV or XLS files to import data. IDE in use will show how if the test(s) passed or failed.

Xamarin (creator of Calabash) provides a continuous integration solution called Xamarin Test Cloud. It is also possible to integrate Calabash with Jenkins. Calabash is open source program so it is completely free.

6.5 Client criteria

This chapter introduces the criteria received from the representatives of Descom that were used, in conjunction with the steps above, to determine which mobile automation tool suits Descom's needs best. As the authors familiarized themselves with all of these tools and examined them through the 11 steps introduced earlier, all of their quirks and features became very familiar and with this information in mind, it was possible to compare the tools to the selection criteria provided by Descom.

According to Descom's representative, the tool should be able to:

- Test web applications
- Continuously integrate with Jenkins
- Work with Robot Framework
- Be initially free, preferably open-source
- Same tests that are made for desktop versions should work without too much configuration
- Have the ability to utilize Python bindings

Test web applications

A lot of mobile testing automation tools have support for native or hybrid applications. Those features are not important concerning the thesis. A strong support for web applications was needed because this thesis is written with mobile web store testing in mind as Descom needed to research the possible solutions for testing their mobile versions of their websites.

Continuously integrate with Jenkins

Every automated test made for web stores at Descom is run through Jenkins with timers dictating when to execute them. The new tests for the mobile versions of the stores would have to be connected to the same Jenkins. This would ensure that the tests are not scattered around different places and the test engineers would find them easier to manage in a familiar place.

Should be able to work with Robot Framework

Robot Framework has proven to be very effective generic testing automation framework which has won many test engineers to its side due to its simple-to-use test syntax and keyword driven testing approaches. This is also the case at Descom. The testing engineers are familiar with this framework after working with it in the past and also after building the current automated tests for web stores. This also takes away the need to learn new, time consuming technologies.

It should be initially free, preferably open-source

Many mobile automation tools are paid or offer solutions like cloud based services or additional features for a fee. Tools like that are arguably great in quality and they have an excellent customer service. These tools offer a great deal for companies ready to pay for them or have good experiences with the companies making them. However, as this study and mobile testing in general at Descom is only on a very basic level it is not possible to acquire funding for tools like that at the moment. Free or open-source tools offer almost similar results and are also expandable in the future; therefore, they served this study well.

Same tests that are made for desktop versions should work without too much configuration

There are plenty of automated tests for mobile web stores already made in Descom which is why it should be very convenient if the new mobile automation tool would accept and run the completed tests as is or with some configuration. This would reduce the workload that is needed to kick start mobile automation testing as the testing engineers do not need to write the tests for beginning all over again. Redundancy is not a feature to be promoted.

Have the ability to utilize Python bindings

As the testing automation for web stores at Descom is executed with Robot Framework which is primarily controlled by Python, it would be desirable that this habit would continue in the mobile testing realm as well. Doing this will ensure that people that are going to take care of the mobile automation testing are going to be familiar with the technologies needed to create, run and modify the actual tests without any problems. They would not have to learn new technologies which leaves more time for actual testing. Also Python is arguably very simple programming language to learn so if new people are going to take the role of mobile automation test engineers introducing themselves to the job is not so daunting task.

6.6 Comparing tools to the criteria

After deciding the criteria what the new mobile automation testing tool should fulfill with the representative of Descom we compared them to the six tools we had inspected more closely. The summary of the results can be found below.

Table 3. Comparing mobile test automation tools against the client criteria

	Appium	Selendroid	Robotium	MonkeyTalk	eggPlant	Calabash
Tests web applications	✓	✓	🚫	✓	✓	🚫
Utilizes python bindings	✓	✓	🚫	🚫	🚫	🚫
Integrates with Jenkins	✓	✓	✓	✓	✓	✓
Works with Robot Framework	✓	🚫	✓	✓	🚫	🚫
Free or open-source	✓	✓	✓	✓	🚫	✓
Desktop version tests work	✓	✓/🚫	🚫	🚫	🚫	🚫

As it can be seen above (Table 3), different tools have very different features. They all are made with divergent goals in mind. Some of them might have key features that are useful for example when testing native applications and others might have these properties cut out to make room for other qualities for example to have web application testing in better shape.

Web application testing and the ability to test mobile websites was the most important quality that was looked for in a mobile testing automation tool. Robotium and Calabash turned out to be the two tools that did not have this feature. This was the major point that practically made their possibility to be the chosen tool non-existent.

The utilization of Python bindings was the next criteria. Overall this was the second least supported feature. Only Appium and Selendroid supported Python directly and without any hassle. Other tools were limited to their own scripting languages like MonkeyTalk and eggPlant and some simply did not support Python. As testing engineers were accustomed to using Python as their scripting language support for it was desirable.

Other criteria that had to do with preserving the same methods in mobile testing automation as in the testing the traditional desktop versions of the mobile stores were the support for Jenkins and Robot Framework. Jenkins was generally supported by all of the tools under inspection. Integration with Robot Framework were more problematic. Selendroid, eggPlant and Calabash did not support Robot Framework. It could be possible with heavy configuration, however, for the time being it is not worth it.

Only eggPlant was not free to use or open-source tool. It had an annual license fee that had to be bought amount of which was determined from the total sum of users that it was acquired for. All others were free but some had additional features that could be bought to increase the functionality of the tool. Paid services were for example Calabash with its cloud service and MonkeyTalk with a professional version with added features.

The least supported criterion was that the already made tests for desktop versions of the web stores would work with little to no configuration on the mobile automation tool. This criterion practically needed the other criteria in order to work. Out of all the different tools this is only achievable with Appium.

6.7 Selected tool

After a great deal of thought and consideration we ended up choosing Appium as the tool of choice for this case. Our decision was based on the eleven selection steps that were introduced and the six client introduced criteria.

As seen previously (Table 3), Appium had all the features that needed to found on the tool based on the criteria. The tests are almost similar to the tests used in automated web site tests at Descom with minor changes to the code syntax; it uses the same scripting language and integrates well with the technologies already in use. The open-source nature of the tool was also a great factor.

There were only a few actual drawbacks to it. It is not the simplest tool to install and it needs plenty of separately downloadable parts in order to work as intended. Also users must be familiar with Python and Robot Framework. It is not intended to be used by the non-tech savvy people.

7 Setting up the mobile browser testing environment

7.1 Prerequisites

Since Appium tests are to be run utilizing Robot Framework it needs to be installed first. Robot Framework requires Python so that needs to be installed prior to installing Robot Framework. Python is easy to install using MSI file (in Windows) that can be downloaded from https://www.python.org/. Python 2.7.X was used in testing. Once Python is installed it is necessary to install pip (downloaded from https://pip.pypa.io/en/stable/installing.html) which is a package management system for software packages written in Python. Pip is installed by typing the following line into the Windows Command Prompt: *python get-pip.py*

After all this Robot Framework can be installed by typing into the Windows Command Prompt the following line: *pip install robotframework*

7.2 Appium installation instructions for Windows

Installation steps:

1. Install Node.js
2. Install Android SDK
3. Install Java JDK
4. Set the environmental variables
5. Install Appium using .exe
6. Install AppiumLibrary for Robot Framework

1 Install Node.js

In order to run Appium, node.js needs to be installed. Version needs to be at least 0.10 (currently 0.12.7). It can be downloaded as Windows Installer or Windows Binary file. Node.js installation also includes npm package manager. Install wizard can also add node.js and npm to the PATH environment variable. Otherwise the installation is fairly straightforward process.

2 Install Android SDK

The next step is installing the Android SDK. It can be downloaded at http://developer.android.com/sdk/index.html. In this case stand-alone Android SDK Tools were chosen since there was no need for Android Studio (IDE for developing on the Android platform).

After the file is downloaded run the .exe file and install it like any other program. It is a good idea to note where the Android SDK will install itself since that file path is needed later. After the installation is successfully completed the Android SDK Manager should automatically open.

From Android SDK Manager install the newest API Level (Appium requires at least API Level 17). (About Appium, Running Appium on Windows)

3 Install Java JDK

Java JDK needs to be installed and the latest version can be downloaded from http://www.oracle.com/technetwork/java/javase/downloads/index.html. In this case x64 version was used since the computer had 64-bit version of the Windows 8.1 Operating System.

Install process itself is fairly standard; just follow the instructions the installer provides and reboot the computer afterwards, if needed.

4 Set the environmental variables

To be able to run Android commands from Windows Command Prompt it needs to be added to the PATH environment variable. In this case a variable called ANDROID_HOME was created and value was set to: *C:\Users\<username>\AppData\Local\Android\android-sdk*. After this that variable can be used in PATH variable. Tools, platform-tools and build-tools need to be added to the PATH variable: *;%ANDROID_HOME%\tools;%ANDROID_HOME%\platform-tools;%ANDROID_HOME%\build-tools*

Java environment variable works almost exactly the same way. Variable called JAVA_HOME was created and its value was set to: *C:\Program Files\Java\jdk1.8.0_45. Then the PATH variable had the following line added: ;%JAVA_HOME%\bin*

5 Installing Appium

Appium can be downloaded from: http://appium.io/. Install process is quite simple; just click next a couple of times and specify the install location and whether the desktop icon should be created.

6 Install AppiumLibrary for Robot Framework

AppiumLibrary for Robot Framework can be installed using the following command: *pip install robotframework-appiumlibrary*

7.3 Configuration

It is possible to configure Appium using the graphical UI or using JSON files.

In order to run Android mobile web test the following capabilities must be defined: platformName and browserName. See Appendix 2.

More detailed explanation of the various parameters and capabilities can be seen on Appendix 3.

Emulator

When running Appium tests using emulator and the system runs on Intel® processor it is good idea to install Intel® Hardware Accelerated Executed

Manager (HAXM). It uses the system's computing power to run the emulator so it generally works much better. Although it can be installed via Android SDK Manager there is a chance it does not work as intended. If that's the case HAXM can be installed separately using Intel's own package. (About Appium, N.d.)

Since mobile web apps are tested Use browser must be ticked. If emulator is used the name of the emulator must be chosen from the drop down menu next to Launch AVD. Capabilities include Platform Name, Automation Name and Platform Version.

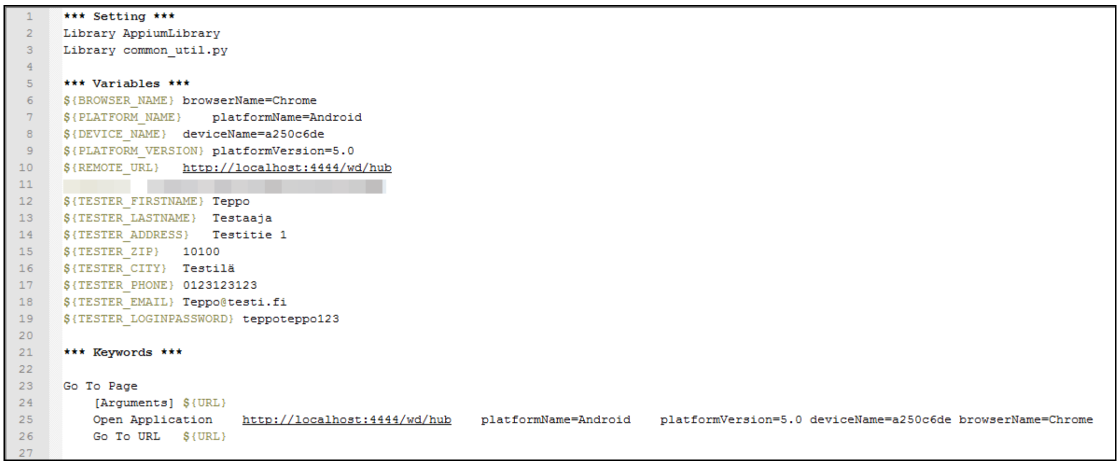
Real devices

Emulators are an adequate alternative, however, a real device reflects the experience that end-user will encounter much closer. A real device needs to be connected to the system that has the Appium server running. By typing adb devices into Windows Command Prompt device(s) should show up.

7.4 Writing test cases

After everything is installed and configured for Appium to work correctly it is time to write the test cases. Test cases for Appium are normally written in any scripting language that has a supported client library. Many popular scripting languages already have a client library. In this thesis we initially write the tests using Robot Framework which uses its own keyword-driven scripts and Python to define the more complex keywords. This is the preferred method at Descom for automating tests for web stores and we were fortunate that it works almost the same with Appium added to the mix. Process for writing them is generally similar for mobile versions of the web stores but with some new elements that need to be taken in to consideration.

Robot Framework with Appium



```
*** Setting ***
Library AppiumLibrary
Library common_util.py

*** Variables ***
${BROWSER_NAME} browserName=Chrome
${PLATFORM_NAME}    platformName=Android
${DEVICE_NAME}  deviceName=a250c6de
${PLATFORM_VERSION} platformVersion=5.0
${REMOTE_URL}   http://localhost:4444/wd/hub
[redacted]
${TESTER_FIRSTNAME} Teppo
${TESTER_LASTNAME}  Testaaja
${TESTER_ADDRESS}   Testitie 1
${TESTER_ZIP}   10100
${TESTER_CITY}  Testilä
${TESTER_PHONE} 0123123123
${TESTER_EMAIL} Teppo@testi.fi
${TESTER_LOGINPASSWORD} teppoteppo123

*** Keywords ***

Go To Page
    [Arguments] ${URL}
    Open Application    http://localhost:4444/wd/hub    platformName=Android    platformVersion=5.0 deviceName=a250c6de browserName=Chrome
    Go To URL   ${URL}

```

Figure 10. Robot Framework test case resource with Appium dependencies

Above is a figure of an example test case. It is basically a normal Robot Framework test suite with couple of differences. First of all, when Appium is used in conjunction with Robot Framework, the standard import for Selenium2Library is replaced by AppiumLibrary. This library needs to be installed either via pip with the command *pip install robotframework-appiumlibrary* or with setup.py with commands *git clone https://github.com/jollychang/robotframework-appiumlibrary.git cd robotframework-appiumlibrary python setup.py install*.

AppiumLibrary serves the same purpose as Selenium2Library which allows Robot Framework to communicate with the different browsers using Selenium WebDriver. Difference is that AppiumLibrary works with Android or iOS. The keywords are very similar between the two with minor differences mainly in the amount of them which is smaller in AppiumLibrary.

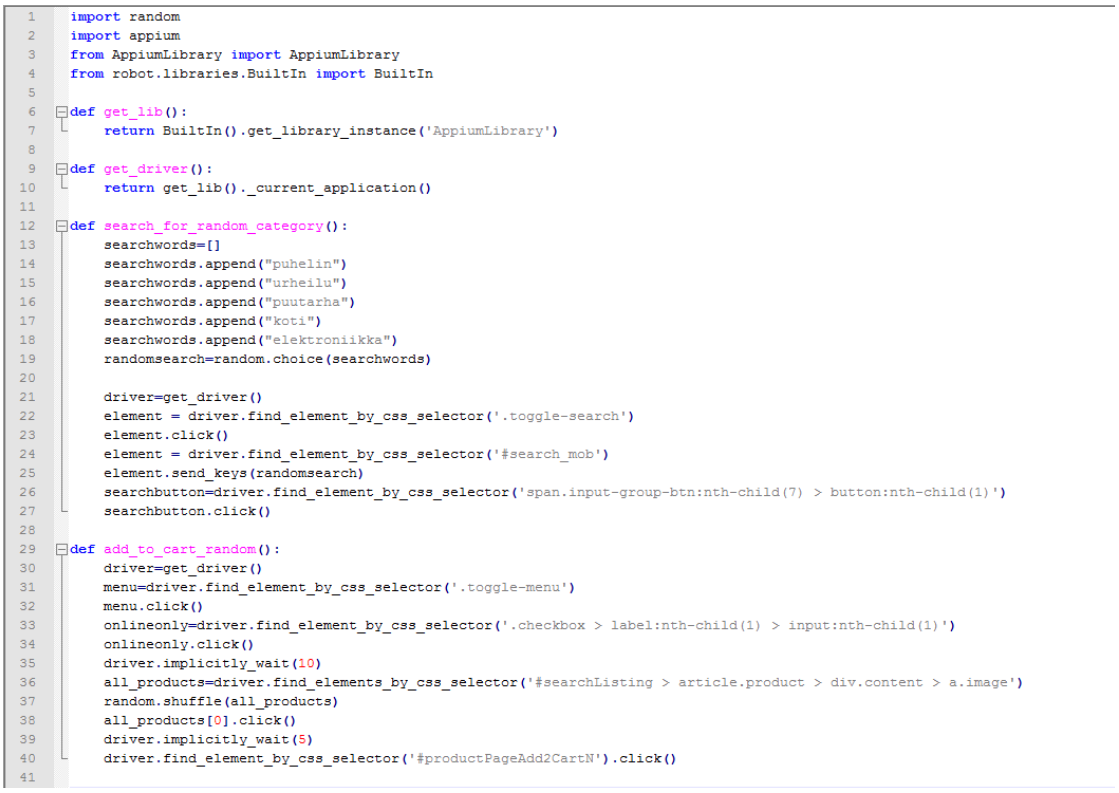
Figure 10 also shows the other factors that separate normal Robot Framework test case file from the one with Appium integrated. Appium needs some desired capabilities in order to guide the correct test case to a certain emulator or a real device that is wanted to execute the test. These capabilities are preferably written in to a keyword that is run every time a test is run. As it can be seen in the figure the needed capabilities are written in to the Open

Application keyword as parameters. The parameters that Open Application needs are in this specific order:

- URL of the Selenium Grid that is being used with Jenkins
- Name of the platform (Android or iOS)
- Version of the platform
- Name of the device that is going to execute the test
- The application that is tested (In our case the mobile version of Google Chrome)

Python file

Robot Framework is a very handy tool, however, it only handles very basic actions like clicking an element or inputting a text. It cannot handle intricate actions like randomizing or complex if statements. That is the reason why Robot Framework's keywords that contain those kind of actions are defined with Python in a separate file.



```python
import random
import appium
from AppiumLibrary import AppiumLibrary
from robot.libraries.BuiltIn import BuiltIn

def get_lib():
    return BuiltIn().get_library_instance('AppiumLibrary')

def get_driver():
    return get_lib()._current_application()

def search_for_random_category():
    searchwords=[]
    searchwords.append("puhelin")
    searchwords.append("urheilu")
    searchwords.append("puutarha")
    searchwords.append("koti")
    searchwords.append("elektroniikka")
    randomsearch=random.choice(searchwords)

    driver=get_driver()
    element = driver.find_element_by_css_selector('.toggle-search')
    element.click()
    element = driver.find_element_by_css_selector('#search_mob')
    element.send_keys(randomsearch)
    searchbutton=driver.find_element_by_css_selector('span.input-group-btn:nth-child(7) > button:nth-child(1)')
    searchbutton.click()

def add_to_cart_random():
    driver=get_driver()
    menu=driver.find_element_by_css_selector('.toggle-menu')
    menu.click()
    onlineonly=driver.find_element_by_css_selector('.checkbox > label:nth-child(1) > input:nth-child(1)')
    onlineonly.click()
    driver.implicitly_wait(10)
    all_products=driver.find_elements_by_css_selector('#searchListing > article.product > div.content > a.image')
    random.shuffle(all_products)
    all_products[0].click()
    driver.implicitly_wait(5)
    driver.find_element_by_css_selector('#productPageAdd2CartN').click()
```

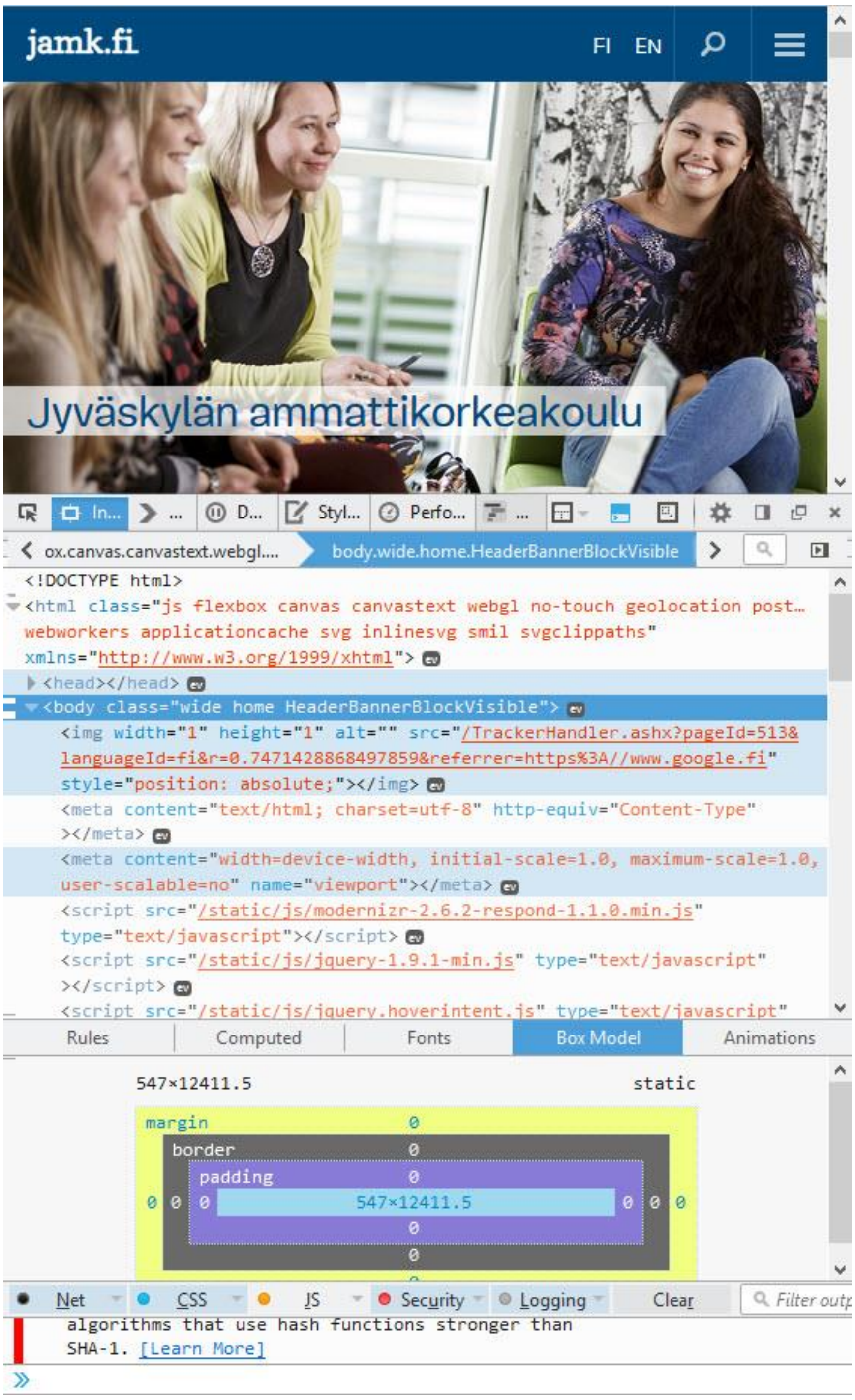
Figure 11. Python file for Appium test case

Like with Robot Framework, when using Python with Appium added to the fray, the changes to the file and to the scripting conventions are minimal. Appium must be imported by adding *import appium* to the top of the file. Also *from AppiumLibrary import AppiumLibrary* should be added as an import. These add the possibility for Python to interact with Selenium and its expansion WebDriver.

Also, as it can be seen above (Figure 11), first two functions are used to import instance of Appium Library. The code is almost the same as with normal importation of Selenium2Library when Appium is not used with minor adjustments. The *return get_lib()._current_browser()* in *get_driver()* is changed to *return get_lib()._current_application()*.

CSS locators that are used to find elements like buttons and texts in the keywords might be different in mobile resolutions than in desktop resolutions. Those should be checked via preferred method of CSS inspection. One simple way is to use a browser that has CSS inspection feature like Google

Chrome or Mozilla Firefox and resize the browser window to the same size as in the mobile device. This will reveal the hidden CSS locators that might only be seen at a certain resolution.



```
<!DOCTYPE html>
<html class="js flexbox canvas canvastext webgl no-touch geolocation post…
webworkers applicationcache svg inlinesvg smil svgclippaths"
xmlns="http://www.w3.org/1999/xhtml">
  <head></head>
  <body class="wide home HeaderBannerBlockVisible">
    <img width="1" height="1" alt="" src="/TrackerHandler.ashx?pageId=513&
    languageId=fi&r=0.7471428868497859&referrer=https%3A//www.google.fi"
    style="position: absolute;"></img>
    <meta content="text/html; charset=utf-8" http-equiv="Content-Type"
    ></meta>
    <meta content="width=device-width, initial-scale=1.0, maximum-scale=1.0,
    user-scalable=no" name="viewport"></meta>
    <script src="/static/js/modernizr-2.6.2-respond-1.1.0.min.js"
    type="text/javascript"></script>
    <script src="/static/js/jquery-1.9.1-min.js" type="text/javascript"
    ></script>
    <script src="/static/js/jquery.hoverintent.js" type="text/javascript"
```

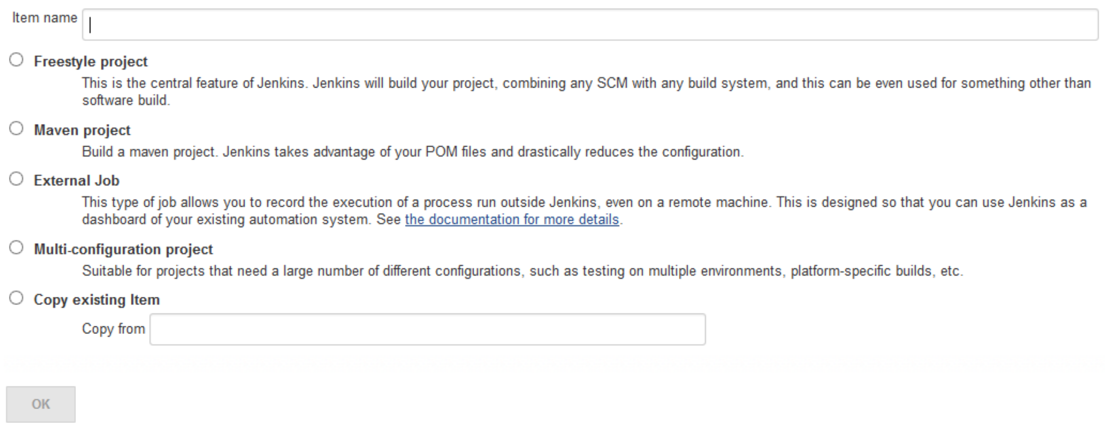
```
547×12411.5                                  static
margin 0
border 0
padding 0
0 0 0  547×12411.5  0 0 0
0
0
```

```
algorithms that use hash functions stronger than
SHA-1. [Learn More]
```

Figure 12. Firefox web element inspector scaled down to mobile size

7.5 Interaction with Jenkins

When test cases are written and ready to be executed they are integrated to the Jenkins CI system. This is done by creating a new item on the front page of Jenkins.



Item name

○ **Freestyle project**
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

○ **Maven project**
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

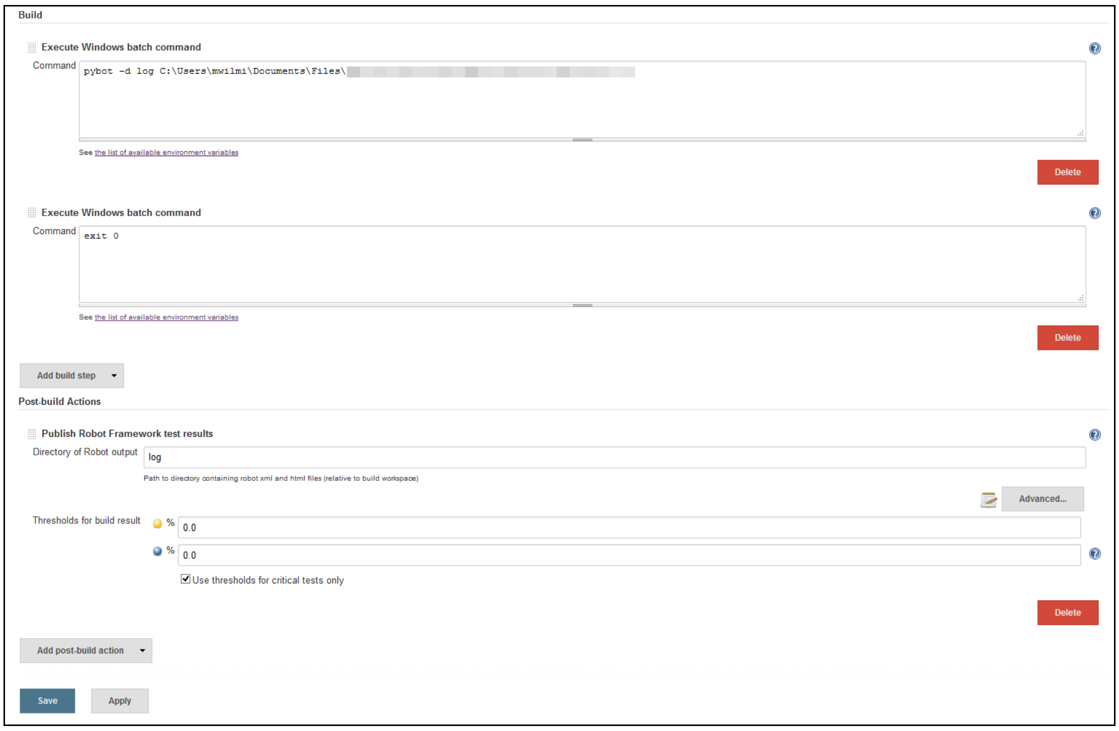
○ **External Job**
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See the documentation for more details.

○ **Multi-configuration project**
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

○ **Copy existing Item**
Copy from

Figure 13. Creating new Jenkins job

This opens a page with different kinds of possible items that can be added to Jenkins. For Robot Framework/Appium projects the Freestyle project option is chosen. The name for it that is shown in Jenkins job list can be assigned here also.



Build

Execute Windows batch command

Command

```
pybot -d log C:\Users\mwilmi\Documents\Files\[redacted]
```

See the list of available environment variables

Execute Windows batch command

Command

```
exit 0
```

See the list of available environment variables

Post-build Actions

Publish Robot Framework test results

Directory of Robot output: log

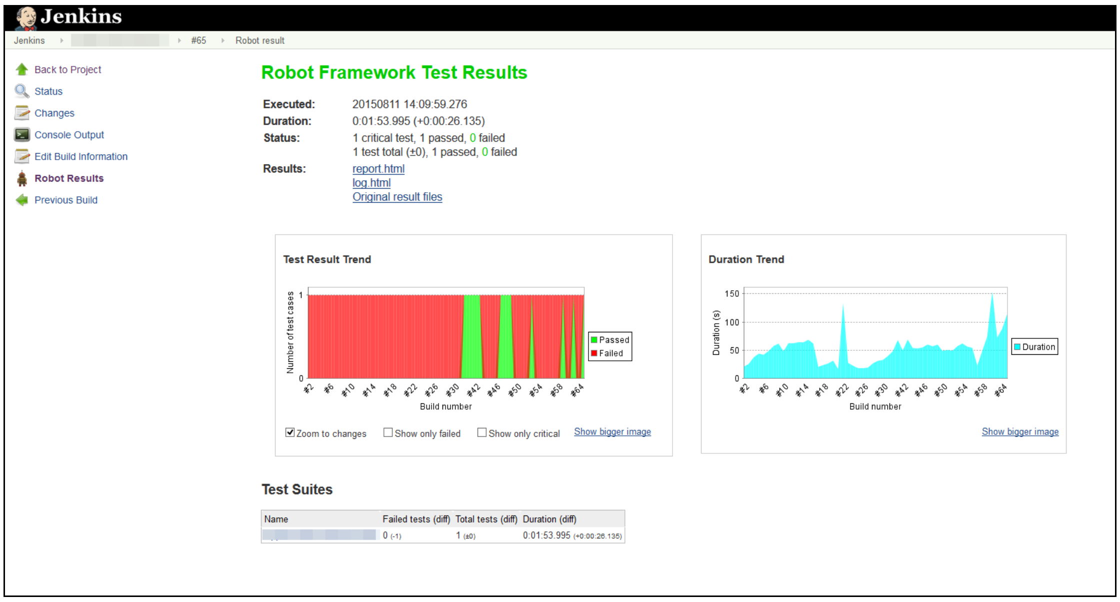
Path to directory containing robot xml and html files (relative to build workspace)

Thresholds for build result: % 0.0; % 0.0

☑ Use thresholds for critical tests only

Figure 14. Jenkins job configuration

The next screen that opens is the configuration for the newly created project. Important configurations that must be set correctly are the different build steps that can be seen above (Figure 14.). Firstly, Windows batch command should be executed with the pybot and the location of the Robot Framework test suite. This runs the test through Appium. Also another Windows batch command with *exit 0* is needed so that Robot Framework plugin can successfully log the results to the right files and display them through the Robot Framework plugin that is installed to Jenkins. Publish Robot Framework test results should be added to post-build actions so that the results are shown in Jenkins after the tests are run.



Jenkins › [redacted] › #65 › Robot result

Robot Framework Test Results

Executed: 20150811 14:09:59.276
Duration: 0:01:53.995 (+0:00:26.135)
Status: 1 critical test, 1 passed, 0 failed
1 test total (±0), 1 passed, 0 failed
Results: report.html
log.html
Original result files

Test Suites

Name	Failed tests (diff)	Total tests (diff)	Duration (diff)
[redacted]	0 (-1)	1 (±0)	0:01:53.995 (+0:00:26.135)

Figure 15. Jenkins's Robot Framework plugin

Robot Framework plugin is very handy for showing the results of the executed tests. Clear indicators in every project show the number of times the project has been run successfully, how many times the build has failed and the passing rate as a percentile. Also, links to the latest report.html and log.html are clearly shown increasing the usability of the plugin.

7.6 Running the test cases

When the tests are integrated with Jenkins it is time to finish setting up the test environment for running the tests. As Appium is already configured only couple of things are left to be done before testing can start. The mobile device that is used as the platform for the tests should be connected to the computer with the Appium. Developer settings and USB debugging must be enabled for test to work correctly. These options can be enabled from Android device by:

1. Accessing settings menu and from there "About Phone"

2. Locating "Build Number"

3. Tapping "Build Number" seven times. After three taps a pop-up will appear informing you that developer options will be enabled after four more taps.

4. After developer options are enabled they can be found from the options next to "About Phone"

5. "USB-debugging" can be found from within the Developer options

After those options are in use, the device can be connected via USB cord to the computer. If the phone is connected to the computer for the first time it will display a prompt informing this. Tapping "Ok" will confirm the connection. The phone is now ready receive tests from Appium.

To verify that the device is connected as intended simple line of command, *adb devices*, can be used. It lists all the connected android devices, both virtual and real. The name of the device connected should be seen there.

When the phone is connected and Appium is configured, all that needs to be done is to start Appium. Easiest way to do this is to launch it from the installed desktop program.

After everything is set up test cases can be executed via Jenkins. The project can be run with multiple different ways. Easiest way is to manually build the project from for example the front page of Jenkins. However, this is not ideal and should only be done when testing a new project. From the project configuration page it is possible to assign the test project to be built when certain event occur.

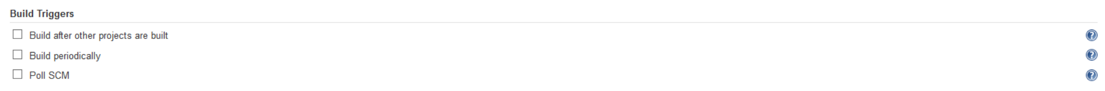


Figure 16. Jenkins job build triggers

These different build triggers can be seen in the figure above (Figure 16). For this case either building after other projects are built or building periodically is a great option. Building after other projects is clever because the mobile tests could be run after the desktop versions of tests are build bundling the regression/smoke tests to a neat bunch. This will give the results from both of the test environments to the developers roughly at the same time which will give them the idea of what is broken and needs fixing. However giving the mobile tests a certain timeslot and building them in their own time is a reasonable alternative too.

8 Conclusions

8.1 Results

The aim of the thesis was to investigate which mobile testing automation tool would be most suitable for Descom's needs and how it would be integrated to the existing automated testing systems already in use in few online web store projects. The thesis successfully answered to the research questions stated in chapter two, which stated this object of the thesis, thus being successful.

The thesis was able to prove that the chosen tool Appium suited the needs of Descom best. It was done by careful investigation, analyzation and comparison of different tools in the market. By this pedantic examination it was determined that Appium was the most suitable. It was also proven in the thesis that Appium is able to integrate to the existing testing environment by emulating it by the writers in their own computers.

Since the thesis was written in English it can be easily utilized in the foreign offices of Descom. It gives more value to this thesis.

8.2 Further development

This thesis focused on mobile testing automation for mobile web stores and the mobile side of it was executed only on Android. The system is set up so that it would be very easy to add iOS to the testing cycle in the future as Appium fully supports it. Without an access to Mac OS’s or phones from Apple thesis did not cover iOS side of Appium and it was not necessarily needed as mobile testing is only in its very early stages at Descom.

Appium is very flexible piece of software and if there is going to be any changes to the existing testing cycle that is used at Descom, it is most likely going to fit in the new premises also. Also, as there are currently plenty of different tools, scripting languages and processes used at different projects at Descom, Appium will most likely able to fit to most of them. If Descom also chooses to develop hybrid or native applications for mobile devices, Appium is up to that task.

8.3 Discussion

The working environment in which the thesis was written was really positive and helped to motivate the writing process in completely different way than if the thesis had been written at home. The fellow students that were writing their theses at the same time with the writers provided a kind of safety net that boosted morale and in a sense gave a shoulder to cry on when the process seemed exhausting. That kind of group mentality was invaluable for the success of the thesis. Also, Descom's employees were very supportive and they gave the writers guidance and pointers regarding the process of producing the thesis.

The writing process of the thesis was fairly straightforward although the fact that the thesis needed to be written in English was some-what surprising at first. All courses related to the thesis writing at school were assuming that the thesis would be written in Finnish. This obstacle was not insurmountable and gave the thesis writers chance to brush up their English.

The most lackluster part of the thesis was the fact that the planned pilot project for the mobile test automation tool did not pan out. This could have happened, if the writers would have been more active in pursuing the commencement of the pilot project. The writers could not test their results in actual projects but the environment was mimicked as close as it was possible giving the results the needed confirmation. Hopefully, in the future thesis will prove to be valuable to projects branching into mobile testing.

References

About Appium. N.d. Documentation for Appium. Accessed on 1 July 2015. Retrieved from http://appium.io/slate/en/master/?ruby#about-appium

About Jenkins CI. N.d. Accessed on 23 June 2015. Retrieved from https://www.cloudbees.com/jenkins/about

About MonkeyTalk Platform. N.d. Accessed on 5 July 2015. Retrieved from https://www.cloudmonkeymobile.com/monkeytalk-documentation

Appium Sponsors. N.d. Sponsors of Appium. Accessed on 1 July 2015. Retrieved from http://appium.io/sponsors.html?lang=en

Berg, Alan. 2012. *Jenkins Continuous Integration Cookbook.* Olton, Birmingham, GBR: Packt Publishing. Retrieved from http://www.ebrary.com

Certified Tester Foundation Level Syllabus. 2011. PDF on ISTQB website. Accessed on 15 June 2015. Retrieved from http://www.istqb.org/downloads/viewdownload/16/15.html

Chang, W. 2015. *Appium Library.* Accessed on 3 July 2015. Retrieved from https://github.com/jollychang/robotframework-appiumlibrary

Comparison among Black-box & White-box Tests. N.d. Software Testing Genius. Accessed on 3 August 2015. Retrieved from http://www.softwaretestinggenius.com/photos/wbtut1.JPG

Continuous Stories. N.d. ContinuousAgile.com. Accessed on 3 August 2015. Retrieved from http://www.continuousagile.com/unblock/cd_mobile.html

Dary, D., & Palotas, M. N.d. *Mobile Test Automation with Selendroid.* Accessed on 12 June 2015. Retrieved from http://www.methodsandtools.com/tools/selendroid.php

Descom's website. N.d. Home page of Descom. Accessed on 3 August 2015. Retrieved from https://www.descom.fi/

Fowler, Martin. 2006. *Continuous Integration.* Accessed on 12 June 2015. Retrieved from http://martinfowler.com/articles/continuousIntegration.html

How it Works. N.d. Selenium Grid. Accessed on 3 August 2015. Retrieved from http://grid.selenium.googlecode.com/git-history/22ed3ff910401af083bf06a4d13514f4c6a623ca/src/main/webapp/how_it_works.html

Huston, T. N.d. *WHAT IS REGRESSION TESTING?* Accessed on 17 June 2015. Retrieved from http://smartbear.com/all-resources/articles/what-is-regression-testing/

Introduction to Calabash. N.d. Documentation for Calabash. Accessed on 11 July 2015. Retrieved from http://developer.xamarin.com/guides/testcloud/calabash/introduction-to-calabash/

ISTQB Exam Certification. N.d. ISTQB Exam Certification website. Accessed on 16 June 2015. Retrieved from http://istqbexamcertification.com/

Johanson, C. 2013. *Avoiding the Mobile Test Automation “Gotchas”.* Pyramid Consulting 7 March 2013. Accessed on 6 July 2015. Retrieved from http://info.pyramidci.com/blog/bid/274460/Avoiding-the-Mobile-Test-Automation-Gotchas

Kananen, J. 2012. *Kehittämistutkimus opinnäytetyönä.* Tampere: Tampereen Yliopistopaino Oy – Juvenes Print.

Kawaguchi, K. 2015. *LTS Release Line.* Article in Jenkins Wiki. Accessed on 28 June 2015. Retrieved from https://wiki.jenkins-ci.org/display/JENKINS/LTS+Release+Line

Laukkanen, P. 2006. *Data-Driven and Keyword-Driven Test Automation Frameworks.* Master’s thesis. Helsinki University of Technology, Department of Computer Science and Engineering, Software Business and Engineering Institute. Accessed on 7 July 2015. Retrieved from http://eliga.fi/Thesis-Pekka-Laukkanen.pdf

Myers, G., Sandler, C., & Badgett. 2012. *The Art of Software Testing. 3rd.ed.* New Jersey: John Wiley & Sons, Inc.

Nadig, S. 2015. *Ad-hoc Testing: How to Find Defects Without a Formal Testing Process*. Accessed on 17 August 2015. Retrieved from http://www.softwaretestinghelp.com/ad-hoc-testing/

Online shopping. 2015. Wikipedia article. Accessed on 18 August 2015. Retrieved from https://en.wikipedia.org/wiki/Online_shopping

Pettit, N. 2014. *How to Test Mobile Website.* Article about mobile website testing. Accessed on 3 August 2015. Retrieved from http://blog.teamtreehouse.com/how-to-test-a-mobile-website

Plotytsia, S. 2014. *How to Choose the Right Mobile Test Automation Tool.* Accessed on 10 August 2015. Retrieved from http://www.testlab4apps.com/how-to-choose-the-right-mobile-test-automation-tool/

Robot Framework User Guide Version 2.8.7. N.d. Accessed on 2 June 2015. Retrieved from http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html

Sathyan, J., Narayanan, A., Narayan, N., & Vallathai. S. 2012. *A Comprehensive Guide to Enterprise Mobility.* 1st.ed. Infosys Press.

Selendroid. N.d. *Selendroid Selenium for Android.* Accessed on 12 June 2015. Retrieved from http://selendroid.io/

Selenium Documentation. N.d. *Documentation for Selenium.* Accessed on 10 June 2015. Retrieved from http://www.seleniumhq.org/docs/

SmartBear. N.d. *Why Automated Testing?* Accessed on 25 June 2015. Retrieved form http://support.smartbear.com/articles/testcomplete/manager-overview/

Software Test Automation Tools. N.d. Software Testing Tools. Accessed on 20 August 2015. Retrieved from: http://www.testingtools.com/test-automation/

Software Testing Fundamentals. N.d. Software Testing Fundamentals website. Accessed on 1 July 2015. Retrieved from http://softwaretestingfundamentals.com/

Software Testing Levels. 2011. Software Testing Fundamentals. Accessed on 3 August 2015. Retrieved from http://softwaretestingfundamentals.com/software-testing-levels/

Sridharan, M. 2014. *Guidelines To Choosing The Right Mobile Test Automation Tool.* Accessed on 10 August 2015. Retrieved from http://testhuddle.com/guidelines-to-choosing-the-right-mobile-test-automation-tool/

System testing. 2015. Wikipedia article. Accessed on 15 June 2015. Retrieved from https://en.wikipedia.org/wiki/System_testing

TestPlant. N.d. Accessed on 10 July 2015. Retrieved from http://cwbackoffice.co.uk/directory/orgprofile/default.aspx?objid=41348

Tomac, R. 2015. *Selenium2Library.* Accessed on 3 July 2015. Retrieved from https://github.com/rtomac/robotframework-selenium2library

Top 10 Mobile Testing Tools. 2015. Optimus Information. Accessed on 3 September 2015. Retrieved from http://www.optimusinfo.com/blog/top-10-mobile-testing-tools/

User scenario testing for Android. N.d. Accessed on 2 July 2015. Retrieved from https://code.google.com/p/robotium/

Vogel, Lars. 2015. *Continuous Integration with Jenkins - Tutorial.* Tutorial for Jenkins. Accessed on 27 June 2015. Retrieved from http://www.vogella.com/tutorials/Jenkins/article.html

Warner, Janine, LaFontaine, David. 2010. *Mobile Web Design for Dummies.* Indianapolis, Indiana: Wiley Publishing, Inc. Retrieved from www.books24x7.com

Welcome to Apache Maven. N.d. Accessed on 28 June 2015. Retrieved from https://maven.apache.org

White-box testing. 2015. Wikipedia article. Accessed on 14 July 2015. Retrieved from https://en.wikipedia.org/wiki/White-box_testing

Appendices

Appendix 1: Descom’s current test environment for web store smoke and regression tests

Currently, the smoke and regression tests for some of the Descom’s web stores are performed using Robot Framework with Selenium2Library and Python for defining more complex keywords. Continuous integration solution is Jenkins. Selenium Grid hub is run on Linux machine and nodes are on virtual machines running Windows Vista.

Appendix 2: Appium config.json file

```
{
  "capabilities":
      [
        {
          "browserName": "Chrome",
          "platformVersion":"5.0",
          "maxInstances": 1,
          "platformName":"ANDROID",
          "deviceName":"a250c6de"
        }
      ],
  "configuration":
  {
    "cleanUpCycle":2000,
    "timeout":30000,
    "proxy":
"org.openqa.grid.selenium.proxy.DefaultRemoteProxy",
    "url":"http://localhost:4723/wd/hub",
    "host":"localhost",
    "maxSession": 1,
    "port": 4723,
    "register": true,
    "registerCycle": 5000,
    "hubPort": 4444,
    "hubHost": "localhost",
    "role":"node"
  }
}
```

Appendix 3: Appium config.json file parameters explained

cleanupCycle = in ms. Sets how often the proxy will check if thread has timed out

timeout = in s. Time in seconds before the hub will end a test.

proxy = Class of the node.

url = IP address or the name of the Appium Server. Format: http://<url>:<appium_port>/wd /hub

host = IP address or the name of the Appium Server.

maxSession = How many tests can run concurrently in the node.

port = Appium port

register = Is either true or false. Determines if the node tries to register will register or not.

registerCycle = Determines how often (in ms) the node will try to register again.

hub = http://<address of the hub>:4444/grid/register. Node registration request is sent to this url.