Sebastian Muroma

Automated Acceptance Testing

Why and How to Implement?

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Thesis

2 December 2015

Helsinki Metropolia University of Applied Sciences

Author Title Number of Pages Date	Sebastian Muroma Automated acceptance testing: why and how to implement? 35 pages 2 December 2015
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Specialisation option	Software Engineering
Instructors	Tatu Kairi, Consultant Kimmo Sauren, Senior Lecturer

The goal of this thesis was to investigate why manual acceptance tests should be automated in a software project and how the automation should be carried out. The topic is of relevance because automated testing is one way to improve the quality of software.

Different methods, procedures, types and tools can be used to create automated testing. The challenges of automated testing are implementation, setting the right objectives and choosing the right tool. The benefits of automated testing are test execution speed, repeatability and cost savings in the long run. Also calculating the right return on investment is unique for every project and it should be bound to objectives.

Almost anyone can create automated tests, but creating them with minimal maintenance costs is a hard thing to master. The skills needed for manual testing are not the same as the skills needed for automated testing.

A health care device manufacturer has started to implement automated testing. Challenges are lack of experience in automated testing, too wide objectives and planning has been too optimistic. Creating automated test cases for the client has been costly. However, improvements in automation have been made.

Based on the results, the creation of automated testing has been burdensome, but the inefficient manual regression testing is brought forth. The purpose of automated testing is not to test the software only once, but multiple times. Implementing automated regression testing is deemed efficient, but for more specific results more extensive studying is needed.

Keywords	automated testing, acceptance testing, Robot Framework

Tekijä Otsikko Sivumäärä Aika	Sebastian Muroma Automaattinen hyväksymistestaus: miksi tehdä ja miten toteuttaa? 35 sivua 2.12.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tieto- ja viestintätekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaajat	Konsultti Tatu Kairi Lehtori Kimmo Sauren

Insinöörityössä tutkittiin, miksi manuaalinen hyväksymistestaus tulisi automatisoida ja kuinka automatisointi tulisi toteuttaa. Testiautomaatio on yksi tapa parantaa ohjelmiston laatua.

Testiautomaation luomiseksi voidaan käyttää erilaisia testausmenetelmiä, -menettelytapoja, -tyyppejä ja -työkaluja. Testiautomaation haasteita ovat toteutus, oikeiden tavoitteiden laatiminen ja toimivan työkalun valinta. Testiautomaation hyötyjä taas ovat testiajojen suoritusnopeus, toistettavuus ja pitkällä aikavälillä kustannussäästöt. Sijoitetun pääomatuoton laskeminen on jokaiselle testiautomaatioprojektille ainutlaatuinen, ja laskelmien tulisi pohjautua tavoitteisiin.

Vaikka melkein kuka tahansa voi toteuttaa testien automatisoinnin, mahdollisimman vähän ylläpitoa tarvitsevien testien automatisointi on kuitenkin vaikeata. Kustannustehokkaan testiautomaation hallitseminen on haastavaa. Testiautomaation toteuttamiseen vaaditaan erilaisia taitoja kuin manuaalitestauksen tekemiseen.

Insinöörityössä perehdyttiin tapaustutkimuksena terveydenhuollon laitevalmistajan testiautomaatioprojektiin. Haasteita projektissa olivat aikaisemman testiautomaatiokokemuksen puute, liian suuret tavoitteet ja se, että suunnittelu oli ollut liian optimistinen. Testiautomaation luominen oli ollut kallista, mutta automatisoinnissa tapahtui kehitystä.

Insinöörityön tulokset osoittivat, että testiautomaation toteuttaminen on ollut kyseisessä projektissa työlästä, mutta toisaalta esille nousi myös manuaalisen regressiotestauksen selvä tehottomuus. Testiautomaatiolla ei ole tarkoitus testata sovellusta kerran vaan useita kertoja. Tulosten perusteella regressiotestauksen automatisointi todettiin kannattavaksi, mutta jotta saataisiin tarkemmat tulokset, tarvitaan vielä tarkempaa tutkimusta asiasta.

Keywords	testiautomaatio, hyväksymistestaus, Robot Framework

Contents

1 Introduction 1

2 Software Testing Background 2

2.1 Testing 3
2.1.1 Methods 3
2.1.2 Procedures 4
2.1.3 Types 7
2.1.4 Relation to Software Development 7
2.2 Testing Tools 9
2.2.1 Record and Playback Tools 10
2.2.2 Scripting Tools 10
2.2.3 Hybrid Tools 11
2.2.4 Custom Tools 12
2.2.5 Domain Specific Language Tools 12
2.3 Automated Testing 13
2.3.1 Benefits 14
2.3.2 Challenges 15

3 Successful Automated Testing 17

3.1 Planning 17
3.2 Objectives 18
3.3 Tools 19
3.4 Determining Return on Investment 21

4 Automated Testing Project in Large Development Project 24

4.1 Background 24
4.2 Results 25
4.3 Challenges 28

5 Discussion 30

6 Conclusion 32

References 33

1 Introduction

Companies have been testing their software over the decades and there have been multiple automated software testing tools arriving to the markets (Meerts 2012). These tools have been created for the purpose of making sure that the software created works as is intended and to ensure better software quality. Automated software testing tools have multiple different ways to approach this problem: recording manual testing and play backing the recordings, scripting with a programming language and creating test cases with a domain specific language. Successful automated testing is still a rare thing in software industry (Graham & Fewster 2012, xxxi) – large and well-established companies still seem to rely on manual testing or no testing at all when developing software.

The goal of this thesis is to answer the following questions: why should manual acceptance tests be automated in a software project and how should it be done? The thesis also describes a real-life client case where automated testing was introduced to replace manual regression testing. This thesis introduces testing methods, procedures, types and tools for one to succeed in the implementation of automated testing and things to take into consideration. The client case brings perspective to the challenges in the automated testing.

2 Software Testing Background

Quality assurance is a set of activities for ensuring quality for processes for example software development, project management and requirements management. These activities include process definition and implementation, auditing and training. Once the processes have been defined and are in use they identify weaknesses in the current processes and correct those weaknesses to improve the process. (Software Testing Fundamentals 2010.) One great way to improve software quality is to test the software. Software testing is an activity to assure that the software works properly and that it does not contain any faults. It is one way to ensure high software quality; other methods include code review and good coding practices. Software testing is labour intensive and according to Memon (2002), it often accounts for 50 to 60 percent of total software development costs. Graphical user interface testing poses further difficulties that traditional software testing does not adequately address (Memon 2002).

In software testing, the software is evaluated against the specified requirements and thus verified and validated for correct functionality. While the software is being tested it is called Software Under Testing. Testing can be done manually, by automated means or combination of both and by anyone who has capability to test the software in question. These different procedures of software testing are done either by developers, manual testers or test automators. Test automators usually are the ones who implement the automated testing of higher procedures of software testing, although they can automate any of the testing procedures. They create test elements which in this thesis are defined as testing functions that can perform the automated actions against the Software Under Testing, for example pushing a specific button on the software or inputting text into an input field. These test elements can be scripts, parts of scripts or parts of libraries used in testing. A test harness consists of everything testing-related, including test scripts, testing data, documentation, environment variables and many more different testing-related components. Usually the whole test harness is created by test automators.

Coverage means the amount of code is being invoked through testing. When there is insufficient testing the coverage of the tests is usually minimal or non-existent. Insufficient testing is a major cause for software quality issues. When the software is tested poorly or not at all, one cannot be certain that the software can handle even its

basic functionalities. Testing should in minimum be targeted at the weak points of the software.

2.1 Testing

In the early days, software was simple and thus the testing was done by the developers. However these days, software is usually more complex and the developers might not have time to test the software which is under development (Meerts 2012). Because of this, software development has a need for dedicated software testers. Most of the testing done used to be manual and manual testers were needed to verify the necessary quality of the software and free the developers from testing.

In big software development projects, having only manual testers for regression testing is cost inefficient, due to the fact that there might be a need for hundreds of manual testers. This way one cannot get necessary quality assurance from testing the software in a reasonable time. (Graham & Fewster 2012, 129-130). Usually testing is done by hiring manual testers to execute the tests or releasing software for the customer without any testing. Releasing software with insufficient testing can result into software that has major faults, which can have a negative effect on the reputation of the company. It can be concluded that large and complex software cannot be fully tested by manual testing within a reasonable time or at reasonable cost; thus automated testing is in demand.

2.1.1 Methods

White-box testing, also known as glass box testing, is testing of software's internal structure (Dustin et al. 2009, 55). To perform white-box testing, one needs to understand the source code of the software while writing the test cases. The intention of this method includes testing functional requirements, internal security holes and expected output from the Software Under Testing. (Guru99 2015.)

Black-box testing is testing the software without looking at the internal code structure of the Software Under Testing and involves invoking system calls through an interactive interface. This means that testing validates the correct behaviour through the interface

from inputs and outputs as is illustrated in figure 1. (Guru99 2015.) Testing this way one cannot find all the defects — errors may not be reported to the software interface — for example defects in the error-reporting mechanism of the software. (Dustin et al. 2009, 55)

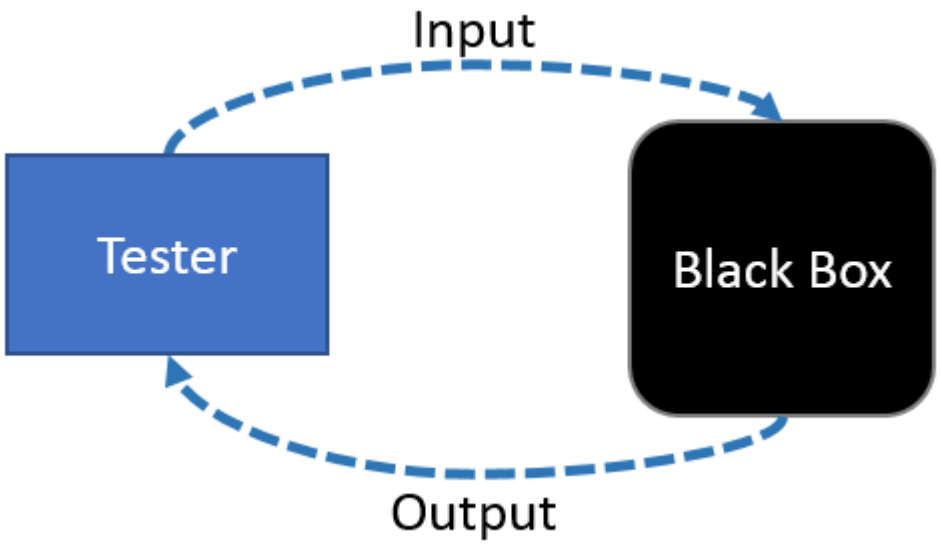

Figure 1. Black-box testing method (Guru99 2015)

Gray-box testing is testing the software using both the white-box and black-box testing methods together. The tester usually has some knowledge of the internal structures for the purpose of creating test cases, although test cases and testing is done with the black-box method from the outside. An example of internal structure knowledge is having access to internal data structures and algorithms. (Software Testing Fundamentals 2010.) Understanding of the architecture and underlying components of the application allows test outcomes to pinpoint to specific areas of the application. (Dustin et al. 2009, 56.)

2.1.2 Procedures

There are four different procedures to implement the mentioned methods for testing the software, which are unit testing, integration testing, system testing and acceptance testing (Software Testing Fundamentals 2011). These four procedures differ from one another by the level of detail they test the software and the state of the software's code

being tested. Figure 2 shows the test procedures from the bottom to up; these are generally executed at different stages of the software development from testing units of the code to testing complete software.

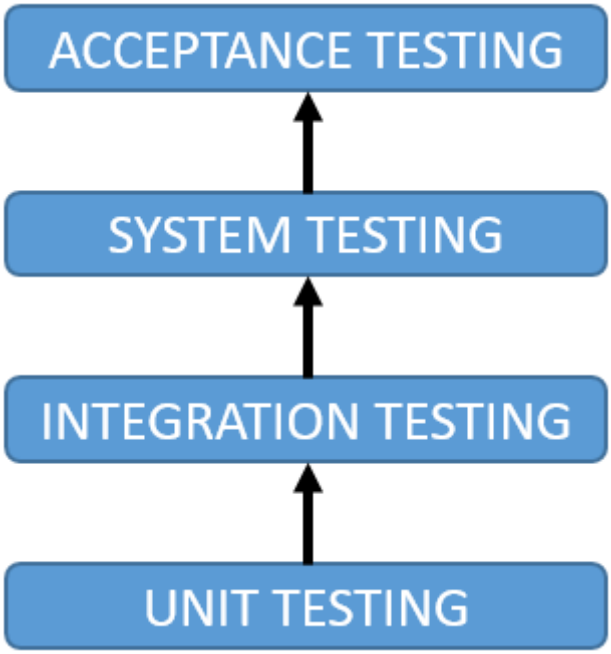

Figure 2. Software testing phases (Software Testing Fundamentals 2011)

In unit testing, the smallest parts of software are being individually and independently scrutinized to assure proper operation (TechTarget 2007). “Unit” is understood as the smallest testable part of software. For example, in object-oriented programming the smallest unit is a method. Usually unit testing is performed by using the white-box testing method and it is generally executed before integration testing. (Software Testing Fundamentals 2011 UNIT TESTING Fundamentals.) Figure 3 illustrates the scale of which unit testing is performed: the output of individual methods are verified against an expected result.

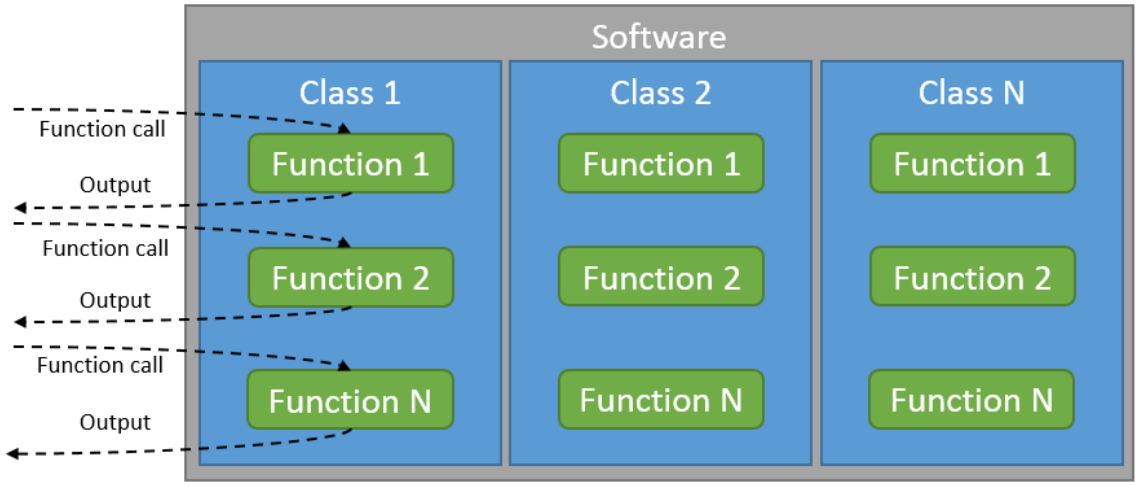

Figure 3. Unit testing in object oriented programming

In integration testing, the individual units are combined and tested together in a group. The purpose is to expose faults in the interaction between integrated units and any of the previously mentioned methods can be used: usually the method depends on the definition of unit. This testing procedure is usually performed after unit testing and before system testing. (Software Testing Fundamentals 2011.) Small software systems are often integrated as a whole software (where it is essentially system testing which is explained below) and tested in a single integration run. Larger systems typically involve several different combinations of smaller groups of integrations to complete the integration testing. (Janalta Interactive Inc 2015.)

System testing is the testing of fully integrated software, even including possible external peripherals for checking how the Software Under Testing interacts with all other software interfaces. System testing is generally performed with the black-box testing method and is performed after integration testing and before acceptance testing. The purpose of this testing procedures is to test the software in a way where the whole system is being exercised. (Guru99 2015.)

Acceptance testing is typically performed after system testing and before releasing the software. It is usually performed with the black-box testing method. (Software Testing Fundamentals 2011.) ISTQB (2014) defines acceptance testing as “[f]ormal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.”

This means that the software's compliance with the business requirements is evaluated, and whether it is acceptable for release is assessed (Software Testing Fundamentals 2011).

2.1.3 Types

Regression testing is to ensure that code modifications do not introduce new errors into already tested code (Memon 2002). This means that the software is tested in each development iteration. Regression testing is not a testing procedure due to the fact that it is done in all of the mentioned testing procedures. (Software Testing Fundamentals 2011.) It is typical that these tests are executed multiple times to ensure that the Software Under Testing is functioning as is expected after software modifications.

Exploratory testing is for finding defects usually in an ad hoc way. It is a good way to find faults in the software, because the software is being scrutinized in different ways compared to regression testing — where the same tests are run over and over again. The software should be investigated in no particular predefined order or it can be done in any testing procedure. For example the tester can use existing test cases as a base to begin the testing, extending it by executing steps outside the defined test case. Tests may be executed based on past experiences in testing similar software or utilising the knowledge of weak spots in the earlier versions of the software. Exploratory testing is creative work and is often not automated, due to it being executed in a detective manner, thus usually containing a great deal of random inputs. (Desikan & Ramesh 2008, 237.)

2.1.4 Relation to Software Development

The Waterfall model was introduced in the 1970's by Winston Royce (Meerts 2012). It is still a widely used software development model. In this model, each phase needs to be completed before the next phase can begin, the outcome of one phase acting as the input for the next phase sequentially. (Tutorials Point 2015.) Verification and validation model (V-model) is a modified version of the Waterfall model (Tutorials Point 2015). In this model, the phases are completed similarly as in the Waterfall model, but the V-model splits testing phases to smaller more specific stages. Figure 4 shows the phases

of the V-model. On the left-hand side of the figure, the phases performed are the same as in the waterfall development process, while adding more specific testing stages on the right-hand side. This model has two main phases: a designing phase and a validation phase.

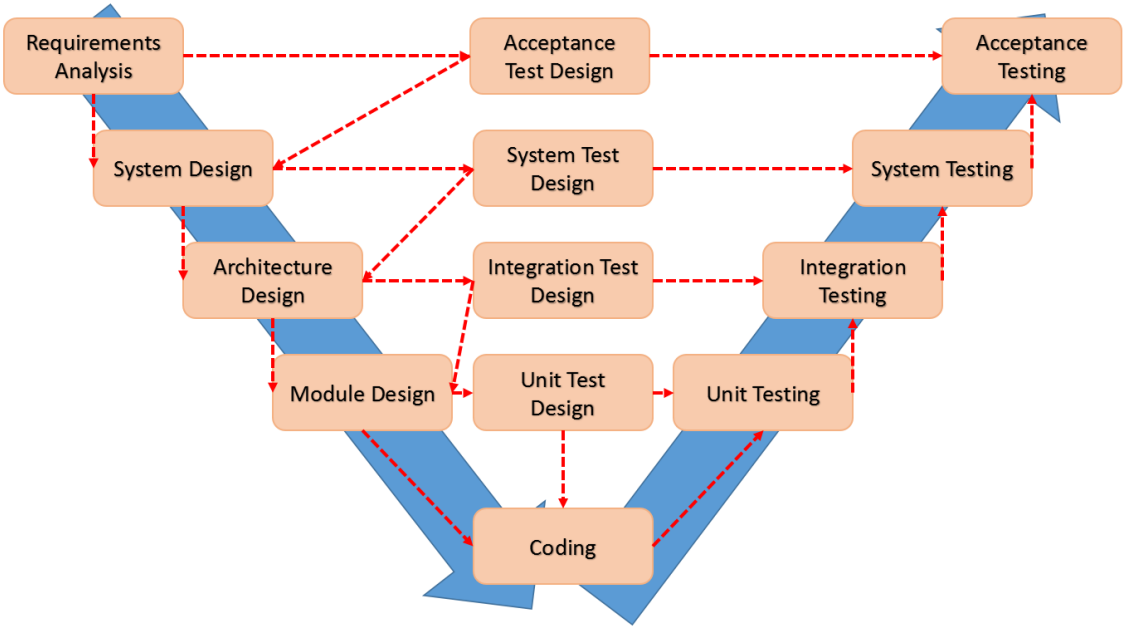

Figure 4. V-model (Tutorials Point 2015 V Model - SDLC)

In the Agile family of software development process models the way to develop software is usually done in small iterations. Each iteration has its own planning, development and testing phases. In this model, possible release versions of the software are developed frequently which may be deployed for the customer. (ISTQB exam certification 2015.) Figure 5 shows the development cycle of an Agile-like model; after developing part of the software, tests are created to make sure that the developed part works as is intended and that it is not faulted in further changes made in the development process.

The Agile family of software development process models and the V-model both highlight the importance of testing. In Agile software development, the software requirement changes can be handled more easily than in the V-model. However, in the V-model the requirements should be defined clearly before implementation; therefore it should be easier to estimate the end of the development project compared to Agile. Agile's strongest forte comes in when the testing of the developed features are

implemented right after they are completed; thus the automated testing value increases when tests are created as early as possible. Both development models have the planning, development and testing phases and are similar in that way, but the scale of these are different. In Agile software development, usually each stage lasts from hours to days whereas in the V-model the phases usually take weeks, months or even years to complete. Every kind of testing procedure may be executed in any development process; the way one develops the software does not bind the development project to a specific testing procedure or type.

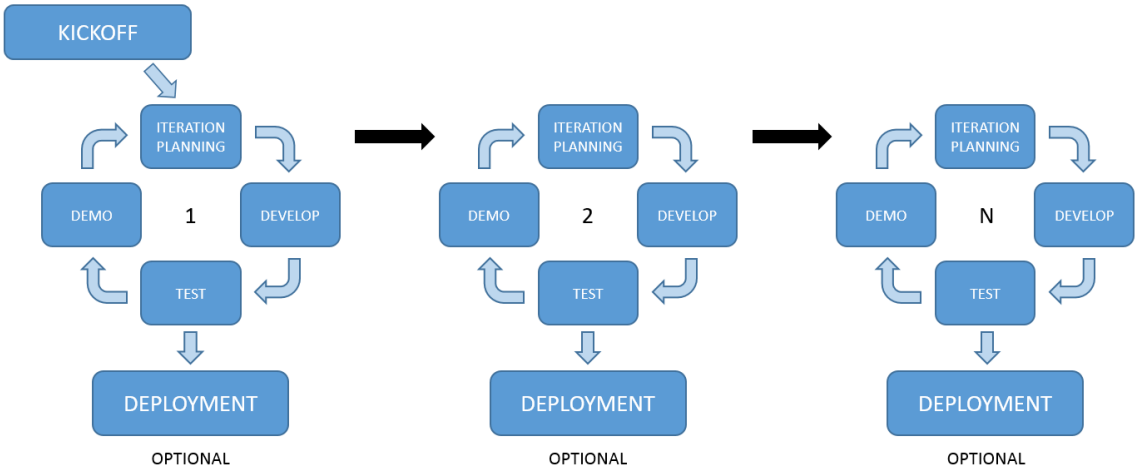

Figure 5. Agile-like model (ISTQB exam certification 2015)

2.2 Testing Tools

Tools for test automation have been around for more than 30 years: the first commercial test tool AutoTester was released for PCs in the year 1985 (Meerts: 2012). Thus, the idea of creating automated tests is not novel. Early software applications were used with a command-line user interface where users typed specific commands and the software performed corresponding actions. Test automators used and relied on test scripts with a collection of different command-line commands. These days software usually has a graphical user interface. Creating automated testing on the applications is harder than for command-line applications due to the complex environment with a computer mouse. (Li & Wu 2005, 4.)

Commercially available tools are usually platform dependent (Li & Wu 2005, 7). For example, testing tools developed for Windows do not necessarily work on the UNIX or

Linux operating systems. These tools often come with a tool-dependent scripting language; this limits automated testing by forcing the users to use the specific tool-dependent language (Li & Wu 2005, 7). When the tool is deemed unusable for testing the Software Under Testing, the scripts also become obsolete. It can be concluded that testing tools and the scripts one uses should be usable on multiple platforms and support multiple environments, as it guarantees longer life for the automated testing harness. There should be a tool for whatever needs there are for automated testing. Finding the right tool for automated testing is not a simple task to accomplish (Graham & Fewster 2012, 7).

2.2.1 Record and Playback Tools

Record and playback tools have been available a long time. These tools record the testing done by a manual tester and then it can be played back repeatedly. They had a huge success in the command-line and mainframe era, and they still have the reputation of the go-to automated testing tool even for software with a graphical user interface. (Graham & Fewster 2012, 87.) The well-reputed record and playback struggles to work properly on the graphical user interface environment, because traditionally they record the mouse click coordinates in the recordings. If there is a minor change in the application window, the test might start failing. At first the record and playback testing in graphical user interface environments were thought to be usable. (Memon 2002.) When there is a change in the user interface layout, the related test cases need to be re-recorded to work with the changed user interface. This maintenance effort increases workload and makes record and playback unfeasible as a long-term solution. (Memon 2002; Graham & Fewster 2012, 74.) There is a portability problem in record and playback which is that the recorded test cases usually run only in the environment which they have been recorded in (Li & Wu 2005, 7).

2.2.2 Scripting Tools

Scripting tools are software testing tools which use their own scripting or a full-fledged programming language to run the tests. These tools often require background in programming; custom scripting language raises the learning curve even higher (Li & Wu 2005, 12). Creating automated test cases with scripting tools may take more time

than using record and playback tools, but these test cases usually are easier to maintain, because they will not fail in minor changes in the Software Under Test. The test cases are usually documented separately from the test scripts; therefore, looking only at the scripts, it is not easily clear what is being tested in each step.

As an example of a scripting tool, the Selenium project (Selenium Documentation 2015) provides a set of different software testing tools for testing web applications. Selenium 2 is the main tool for creating tests with Selenium. Selenium supports most of the commonly used programming languages and operating systems for scripting. (Selenium Documentation 2015.) The Selenium 2 testing framework will appeal to programmers due to the fact that the test cases are created in a familiar environment: programming.

2.2.3 Hybrid Tools

Hybrid tools use record and playback for generating the test scripts in a scripting language which can be modified and add additional verification logic in them. Most of the modern record and playback tools have the ability to record the test cases into scripts. These testing scripts are often in custom scripting languages; thus a tester cannot effectively operate the record and playback tool due to a high learning curve of the custom language. Usually these tools require a great deal of time to master and to be fully utilized. There may not even be any useful information on the internet about these languages or tools; thus testers have to rely on just the instruction book or manuals that come with the tool. (Li & Wu 2005, 12.) In graphical user interface environments, testing scripts require more complex sequences of actions such as pressing a button, inputting characters, and then moving the mouse cursor. These actions usually require the exact coordinates of these test elements to be saved in the testing scripts. Test case automation often relies on the test automator to be able to create feasible test elements on the Software Under Testing. (Memon 2002.)

As an example of a hybrid tool, WinRunner (Hewlett-Packard 2014) is Mercury's hybrid tool for Windows. Users can choose to record test elements based on objects or on screen coordinates. They can then specifically add or remove graphical user interface parts from an existing graphical user interface map. Users can also interact with a test creation helper to check window and object bitmaps and insert checkpoints and test

data to inputs. The test scripts which consist of test elements are recorded in language named Test Script Language. With this tool one can use an external test data file to test against Software Under Testing, but creating this test data is done with the test data generator helper, and it is a manual procedure. (Li & Wu 2005, 26)

2.2.4 Custom Tools

There is always an option to create one's own testing tool from scratch as Li and Wu (2005, 9) suggest. The tool can then be tailored to be perfect for one's automated testing needs. This strategy to create a custom tool is the same as creating any other software. (Li & Wu 2005, 9.) The custom tool creation would take time and cost a lot of money for it to be complete; the tool created should also be tested for correct functionality, thus further increasing the time taken to develop it. Therefore, creating such a tool is not easy nor is it cheap, but the benefits of it are quite tempting. The testing tool is then specifically tailored for the Software Under Testing, thus supporting the environment of the software and ability to test the software correctly. Because creating one's own custom tool is neither a cheap or easy task to accomplish compared to using an off-the-shelf tool, the decision of making a custom testing tool should be the last choice as these things are dependant solely on the Software Under Testing. If there is no suitable tool for testing the software, then there may exist a need to make custom tool.

2.2.5 Domain Specific Language Tools

Domain specific language tools use natural language which test automators, manual testers or even managers can use to describe the test cases. However, the technical implementation is then implemented by the test automators. Steps which create the domain specific language are typically test elements or a combination of them. (Buwalda 2015.) These features enable readable test cases and make it easier for the non-technical personnel to understand how testing is done and what parts of the software are tested with the tool. Compared to record and playback, hybrid and scripting tools where the executable test cases are often in obscure byte data or in a programming language format. Domain specific language tools should support the ability to extend the existing libraries; this gives more flexibility to the automated testing

implementations. Thus extending an existing testing framework is also a possible choice while choosing a domain specific language testing tool. These types of tools are easier to use due to the natural language test cases, when comparing to record and playback, scripting and hybrid tools.

As an example of a domain specific language tool, Robot Framework (Robot Framework 2015) test cases are written in natural languages and contain keywords as test steps which are executed. The framework generates HTML document about the test results and steps. It also generates XML files, which can be used to create custom reports based on the results. The framework is easily extensible with Python or Java. The users can extend the existing pool of libraries by creating their own libraries. (Robot Framework 2015.)

2.3 Automated Testing

The purpose of automating manual testing is to speed up development lifecycles and increase the application reliability (Li & Wu 2005, 2). Automated testing is done either with record and playback, scripting, hybrid or domain specific language tool to create easily repeatable test cases. These usually give fast feedback and larger code coverage than what could be achieved with manual testing alone. Although companies still seem to rely only on manual testing or no testing at all, some companies have started to implement automated testing in their large software projects and have been successful with the automation. According to Graham and Fewster (2012, 6), companies might start automated testing due to a serious problem or near disaster caused by insufficient testing. Another catalyst might be an outsider's view (i.e. a consultant’s view or the view of other personnel outside the company) which can have an effect on starting automated testing. Even management can decide that automated testing is needed for the company’s survival (Graham & Fewster 2012, 6).

Skills needed to do good manual testing are not the same as the skills needed to implement good automated tests (Graham & Fewster 2012, 5). Companies that have already established manual testing and introduce automated testing; the manual testers may fear for getting unemployed due to being replaced by automation (Graham & Fewster 2012, 297). Automated testing, however, does not eliminate the need of manual testers (Graham & Fewster 2012, 243). Automated testing and manual testing

complement each other; automation is there to remove most of the repeatable regression testing from manual testers and let the manual testers to focus on exploratory testing and scenarios which the automated tests cannot handle.

2.3.1 Benefits

Automated testing has its benefits over manual testing but these benefits may not be easy to achieve. Regression tests can be run after each modification to verify the changes in the software has not broken anything (Dustin et al. 2009, 87). This usually makes verifying possible defect fixes and development lifecycle faster. Automated tests do not make mistakes even if they are run continuously and can be run repeatedly to find hard to replicate defects (Dustin et al. 2009, 87).

Automated testing increases the time that the Software Under Testing is being tested even when employees are not at office, for example at lunch time, nights and weekends. It can also cover more of the software; more data variations and test scenarios can be run in shorter time compared to manual testing. (Dustin et al. 2009, 87.) Increasing the coverage of testing usually guarantees better software quality by testing the features in a way which may not be tested manually, for example running tests 24 hours straight, using more varied test data or checking precise timing of certain actions. This also gives more time for the testers to perform creative exploratory testing.

Correct implementation of automated testing includes reducing the time and cost of testing the software (Dustin et al. 2009, 101). Return on investment can be calculated for everything which needs investment and it is meant to find out if the costs are smaller than the investment. In automated testing, the return on investment is meant to find out savings comparing it to manual or no testing at all. It is usually positive after multiple automated regression testing runs when comparing to manual testing (Graham & Fewster 2012, 109). As has been earlier stated; regression tests are executed multiple times and therefore they should be automated. Return on investment is normally calculated by comparing the time taken to execute tests between manually and automatically, but this approach to calculate return on investment does not take into account the time to implement and maintain the automated tests (Graham & Fewster 2012, 4). It is important to see whether automated testing has achieved what

was estimated to be achievable after automation has been established (Graham & Fewster 2012, 4); thus one can determine the real return on investment in the automated testing.

Graham and Fewster (2012, 3) imply that there is a common misconception to start successful automated testing: the only investment needed is the testing tool and therefore investment is not needed if one chooses an open source tool. Therefore, the misunderstanding is that the only investment needed for automated testing to take place is to get the needed equipment for it and giving it to the employees. Making no investments to automated testing can have a negative effect on the return on investment and the costs of automated testing may be higher than the estimated savings (Graham & Fewster 2012, 3). Like any other project the automated testing will usually be unsuccessful without investments and it generally needs applicable persons for delivering it and to maintain the existing tests.

The goal of automated testing is to reduce the number of tests done manually, but not remove the need of manual testing altogether. Usually when and automated test case has been implemented, no human intervention is needed for running it (Guru99 2015). The effectiveness of automated testing is that the Software Under Testing can be tested for 24 hours per day through the whole week with no human interaction. Dustin, Garrett and Gauf (2009, 47) say that manual and automated testing complement each other and that they intertwine; when regression testing is automated manual testers can focus more on exploratory testing, thus providing more coverage.

2.3.2 Challenges

Graham and Fewster (2012, 2) highlight one of the challenges in automated testing which is that the objectives for the automated testing are not defined clearly; thus the automated tests may not be targeted at the right things. Having clearly specified objectives it is easier to evaluate and achieve them. The decided test automation scope can be too large, too complex or too vague for the automated testing to be successful. (Graham & Fewster 2012, 2.) This can result into bad decisions made in the early stages of automated testing, such as starting the implementation of automated tests for scenarios that are not critical for the Software Under Testing. This may lead to implementing automated testing to parts of the software with low return on

investment. Possible poor automated tests increase maintenance of the test harness which in turn decrease return on investment in the long run.

Usually when companies decide to start automated testing, they forget that creation of the automation test harness is similar to software development and requires similar discipline (Graham & Fewster 2012, xxxii). The result of using non-programmers to implement automated testing leads to a low quality test harness with increased maintenance and may even make the company think that test automation is not for them (Graham & Fewster 2012, 5). Testing tools will execute the tests, whether the test cases test something or not (Graham & Fewster 2012, 2). For example a test case created does not verify anything, but only presses a few buttons. The problem with these cases is that the tool possibly tells only that tests passed or failed, without actually testing anything, and thus the tool can sometimes fool into thinking tests are checking and verifying the Software Under Testing (Graham & Fewster 2012, 12). This is why the automated test cases should be reviewed by a person who has knowledge of the testing tool in use and testing itself.

If non-technical personnel implements automated testing it may have problems, such as not testing what it was meant to test. Just because an automated test passes, it does not mean there are no problems (Graham & Fewster 2012, 369). Therefore the automated test cases should be reviewed with a scrutiny (Graham & Fewster 2012, 12). The reviewing should be done by someone who has expertise in test automation, knowledge of the Software Under Testing and the testing tool in use.

3 Successful Automated Testing

This chapter takes a look at factors which should help to create successful automated testing: planning, objectives, testing tools and how to determine return on investments. These chapters introduce what should be done before and at the time of implementation, to get the most out of the automated testing.

3.1 Planning

According to Graham and Fewster (2012, 1), support from managers is essential to achieving a planned return on investment and having a successful automated testing. Management is needed for defining realistic objectives, also providing sufficient and appropriate resources (Graham & Fewster 2012, 1). The test harness of the automation should have a good technical architecture, having right levels of abstraction to give flexibility, adaptability and minimize maintenance costs.

Graham and Fewster (2012, 5) suggest that when starting to implement the automated testing, it should be planned well, the plans should include time for experimentation and the planning should not take an unreasonable amount of time. When the automated testing is started, there should be a reasonable time to produce a good enough proof of concept. In automated testing, proof of concept means creating a minimal set of tests for the Software Under Testing, which can help to understand how to achieve long-term goals of automated testing. (Graham & Fewster 2012, 5.) Proof of concept means creating a very minimal showcase of a product and determining if the concept is viable for the intended use. This is a quick way to try and introduce new ideas. Depending on the project where the automated testing will take place, different approaches are needed: if there will be a lot of people implementing automated testing, the guidelines and standards should be defined as early as possible. However, if there are only a few persons the guidelines and standards are not as critical and can be defined later on. It would be preferable that the rules should be updated and improved constantly. (Graham & Fewster 2012, 11.)

The plan should define at least the first test cases to be implemented. It is important that these test cases yield high return on investment and are created for stable, infrequently changed part of the Software Under Testing. Therefore, one of the good

places to start is to automate the regression tests, thus ensuring that each revision of the software is tested. Usually the most repetitive tests should be automated, because manual testers tend to make mistakes when their focus is not at their best. (Graham & Fewster. 2012, 13-14.)

3.2 Objectives

Graham and Fewster (2012, 2) imply that having good objectives for automation is critical to have successful automated testing. It is necessary to understand that the objectives for manual testing and automated testing are different; for example a good objective for manual testing is to find a lot of faults (Graham & Fewster 2012, 2). However, this is not usually a good objective for automated testing, because a good and reliable automated exploratory testing is usually hard to accomplish and should not be the first thing to be automated. Also, if the target is to automate existing manual regression testing, the objective to find faults through automated testing is not achievable. Regression testing rarely finds new faults; provided previously working functionality has not been broken in the Software Under Testing (Graham & Fewster 2012, 2).

There are however some scenarios where automation does enable new faults to be found: automation may allow tests that could not otherwise have been run, because executing these tests manually is impractical. Implementing these tests automatically increases the coverage of the software being tested which enables finding faults in the software. Also, running automated tests for long sequences can reveal reliability faults. (Graham & Fewster 2012, 15.) Automated tests are easily repeated which makes reproducing flickering faults easier. These faults could otherwise go unnoticed by quality assurance and end up to the customer. Easy repeatability also gives the means to verifying that defects are correctly fixed. (Graham & Fewster 2012, 242.)

One of the key things to having successful automated testing is to have as many reusable test elements as possible, by making the test elements generic enough to be usable in multiple different scenarios. When the test elements are built and they are used in multiple places, it is worth the effort to make sure that they are as well built as they can be. (Graham & Fewster 2012, 12.) Usually the more these well-built test elements are used the better the return on investment on the effort will be. Although

there are situations when it is useful to have disposable test elements, usually individual test elements have a long life in automated testing (Graham & Fewster 2012, 12). If the test elements are generic and can be used multiple times it will make the creation of further automated test cases much faster later on (Graham & Fewster 2012, 100).

At least in large projects, objectives should be identified for the automation to ensure that the solution is directed to the essential parts of the Software Under Testing (Graham & Fewster 2012, 131). The biggest benefit from test automation comes from testing early and often (Graham & Fewster 2012, 234). Once the automation is done, it can be executed to test the software after every new software development change is made or even more often than that.

3.3 Tools

The test automation tool should support the test automators or other personnel who might be using the tool, for example business personnels, developers or testers. The tool used should have necessary features built in for automated testing or support the ability to extend as well as add features to it to accomplish interaction with the Software Under Testing. (Graham & Fewster 2012, 7.) The automated testing tool should support applications that are run on multiple computers, different types of operating systems and work with necessary network protocols (Dustin et al. 2009, 48). Another feature which is good to have is the ability to select specific tests to be run (Graham & Fewster 2012, 552); this gives the opportunity to target the tests for a recently changed part of the software for faster verification.

The tool is not usually the one that causes unsuccessful automated testing, although the tool or the personnel is often blamed for the failure. The tool that works for someone's automated testing may not work for others. (Graham & Fewster 2012, 7.) While evaluating which tool to use in automated testing, one should not be relying only on the success or failures of other automated testing efforts and what tool they used. Rather, the testing tool should be tried against the Software Under Testing and put more emphasis on its features instead of the reviews when deciding to take it into use.

A domain specific language framework, such as the Robot Framework (Robot Framework 2015), will make the creation of test cases easy as test cases are represented in plain language. Domain specific language also allows non-technical personnel to design the test cases as a clear collection of steps. Test automators can then automate these test cases as is necessary. Example 1 shows a generic test case for a web application, where a user logs in with given user account information and is greeted by the welcome page. The Robot Framework's keywords are quite self-explanatory, but it is sometimes necessary to add documentation to the keywords and libraries. Documenting and making the test harness easily accessible helps re-using the existing test elements (Graham & Fewster 2012, 12). This helps the implementation of further automated testing, as there is no need to "invent the wheel" again.

```
*** Test Cases ***
Valid Login
    Open Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
```

Example 1. Robot Framework example test case (Robot Framework 2015)

When creating automated testing the need of abstractions should be taken into account; this ensures the possibility to change some external library. With abstraction changes in the software have minimum impact on the test harness. A good example of this is to have necessary test elements mapped for the graphical user interface. The test cases should not directly refer to the tool or any other library in use; this has the possibility to invalidate the test cases when parts of the test harness are changed. (Graham & Fewster 2012, 9-10.) Figure 6 illustrates a desired test harness architecture which consists of the domain specific language; in the figure the libraries are used to communicate with the Software Under Testing. Manual testers or any non-technical personnel who has understanding about the testing objectives and the Software Under Testing should be able to create the test cases with a domain specific language. Correct abstraction reduces the need for technical knowledge of creating test cases (Graham & Fewster 2012, 5).

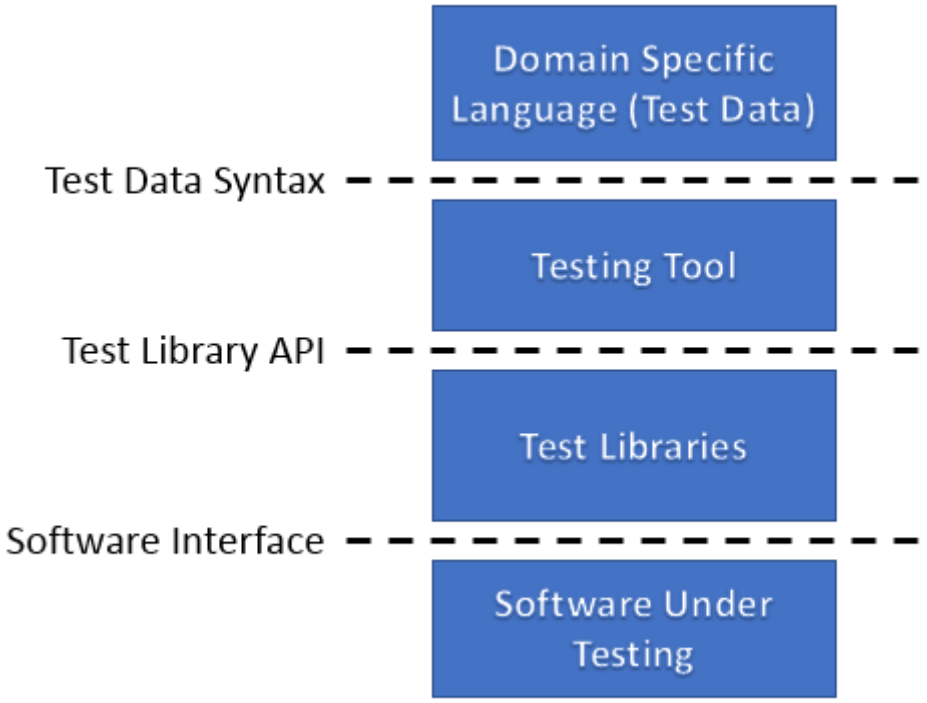

Figure 6. Abstraction Levels (Graham & Fewster 2012, 9)

3.4 Determining Return on Investment

It is possible to automate all regression tests but not all of them are worth automating (Dustin et al. 2009, 47), as some of the tests may not give positive return on investment. All manual regression tests do not need to be automated for the automated testing effort to be successful (Dustin et al. 2009, 51). Even though the possible target or expectation of the scale of the automated testing has not been met, it does not necessarily mean that the automated testing does not give positive return on investment.

An example of calculating return on investment from a medical X-ray automation testing project is seen in Equation 1; in this case, the calculation is done using the factor of defects found by automation testing and the cost of fixing the defect in the production. When calculating the return on investment with Equation 1 and the outcome comes out as higher than one, then the return on investment would be positive. The equation described in Equation 1 does not take into account the damage to the brand and customer satisfaction costs, only labour and hardware costs. Because the costs of fixing the defect after release vary in different projects, this may not be the

right way for everyone to calculate the return on investment in test automation. (Graham & Fewster 2012, 248.)

$$\frac{(Number\ Of\ Defects\ \times Cost\ Of\ Defect) - (Cost\ Effort + Remaining\ Cost)}{(Cost\ Effort + Remaining\ Cost)}$$

Equation 1. Return on investment calculation (Graham & Fewster 2012, 248)

In Equation 1, the fields are as follows:

- Number of defects are defects detected by reliability tests that would not have been detected by later test phases but that would be encountered by customers and that would require updates in production
- Cost of defect is average costs of labour and hardware for resolving a defect found in the production and re-releasing the product
- Cost effort is the total cost of the effort spent on the development of the test harness and the development of the test cases
- Remaining cost consists of other costs such as hardware purchased, hardware developed and licenses.
  (Graham & Fewster 2012, 248.)

This example of return on investment calculation is from a project where there were ten releases per year. The automated testing costs are compared to manual regression testing costs when there is no automated testing at all. The calculations take into consideration that the tests are run once per iteration. In Equation 2, on the other hand, the costs are calculated purely based on the hours to execute tests by manual versus automated means. After each year the cost of automated test runs are compared to manual test runs. (Graham & Fewster 2012, 109.)

$$Cost\ per\ test\ case = Test\ Cases\ \times [Describe\ test + Releases \times (Execute\ test + Maintain\ test)]$$
$$Cost\ per\ manual\ test\ case = T \times [A + L \times (B + F)]$$
$$Cost\ per\ automated\ test\ case = T \times [(A + C)\ + L \times (E + G)]$$

Equation 2. Return on investment calculation (Graham & Fewster 2012, 111)

In Equation 2, the variables are:

- A describes the time to create test cases for both manual and automated testing
- B is the time taken to execute one manual test
- C is the time taken to implement an automated test ready for execution
- E is the time taken to execute one automated test
- F is the maintenance time for one manual test
- G is the maintenance time for one automated test
- L is the number of iterations per year
- T is the number of test cases are run per iteration
  (Graham & Fewster 2012, 110-111.)

Equations 1 and 2 are ways to calculate return on investment for automated testing for different projects. In both of these projects there were already established manual testing before starting the automated testing. However, both of the projects had different costs on deploying new software for customers. (Graham & Fewster 2012, 109; 238.) It seems that the way of defining one's own return on investment derives from the objective of automated testing and each project may have their unique factors to consider.

4 Automated Testing Project in Large Development Project

This chapter discusses challenges and results in a development project where automated testing was introduced and where previously only manual testing had been carried out. The client is a health care device manufacturer, which started to implement automated testing of their acceptance testing for product line of hospital medical devices.

4.1 Background

The development project's goal is to release a new version of the medical device software and hardware to the markets. Earlier versions have only had manual testing for verification and validation. Automated testing has been introduced as a way to reduce the time verification and validation phase takes. The final goal is to get from a four year release cycle to a one year cycle. Another goal is to increase the quality of the medical device by testing it more broadly by testing the software with different data sets.

The goal of automated testing is to automate one third of the requirements automated which is 2,761 requirements in total. In medical technology, the validation and verification has to be thorough due to heavy regulations. The automated testing was estimated to be established in a year, but at the time of writing this thesis it was extended by another year. With the one year extension, new requirements and changes to old ones were introduced. The testing tool in use in the project is the Robot Framework: It was chosen for its ability to write the test cases in a natural language, for already available necessary libraries and for the ability to extend the tool.

The software development of the client company has been split into different teams and thus the responsibility to test their own area of development has been issued to each of them. They can decide by themselves which parts of their area will be covered with automated testing and the quality of automated test cases. A basic test environment consists of a medical device, a simulator, a computer and an internet connection.

4.2 Results

1,489 requirements out of the total 2,761 have been automated, which is more than half of the total automated testing target. The status of the project at the time of writing the thesis can be seen in Table 1. “Released” refers to those test cases which have passed the review process and can be assumed as done.

Table 1. Automated testing status requirements

Total requirements	**Automation Target**	**Released**	**Work In Progress**
8,624	2,761	1,489	96

Table 2 shows the comparison between manual and automated testing. The average verification test time per requirement in Table 2 takes into consideration the amount of time it takes to document the results. In automated testing, the results can be generated from the XML data files automatically; thus it is faster and there should not be any errors in the documentation. There have been scenarios in manual testing where the documentation of the manual test run has been incorrect; for example, there have been cases in which the date has been documented in a wrong format. The whole test run concerning the incorrect documentation has had to be executed again. This does not happen with the automated testing due to the test results being generated automatically. Also in Table 2 the average creation time for automated testing consists of the time it takes to describe, verify functionality and review the test.

Table 2. Time taken per requirement

	Manual	**Automated**
Average time it takes to test a requirement	1.35 hours	0.044 hours (2.6 minutes)
Average requirement test creation time	0	15 hours
Average maintenance time per requirement	0	0.5 hours

Table 3 shows the time it takes to test 1,489 requirements using the averages from Table 2. There are four different hardware configurations for the medical devices but

the software is similar in each of them. This is why in Table 3 only three medical devices are listed to be manually tested; however, all four devices should be tested with automated testing. Manual tests are run twice per iteration: there is a pre-verification run and then there is the verification run. In the pre-verification run the verification procedure is checked and validated to ensure a smooth verification run. If the process is valid, the verification run is executed. A full verification test run includes the number of medical devices and test runs, which differ from manual and automated testing.

Table 3. Test run times

	Manual	**Automated**
Requirements run	1,489	1,489
Needed test runs per verification	2	1
Medical devices to test with	3	4
Full verification test run time (hour)	12,068	261.5
Total time taken to complete testing (work day)	1,609	3,088

From Table 3 one can see that the full verification test run with automated testing is a fraction of the time a similar run takes manually. The difference between the total possible time to execute tests in manual and automated testing is, respectively, seven and a half hours per day versus 24 hours per day. The total time taken to complete testing with automation includes creation, maintenance and the full verification run time of the test cases. With the current automation status verification run time has been reduced by 11,806 hours. However, the total time it has taken to complete the automated testing for 1,489 requirements is 1,479 workdays more than just testing them manually. Although when the requirements are tested in multiple iterations the automated testing has its benefits as is seen in Figure 7. Automated testing costs more in the first iteration compared to manual testing. After two full verification runs have been executed, the automated testing total return on investment is positive compared to doing only manual testing.

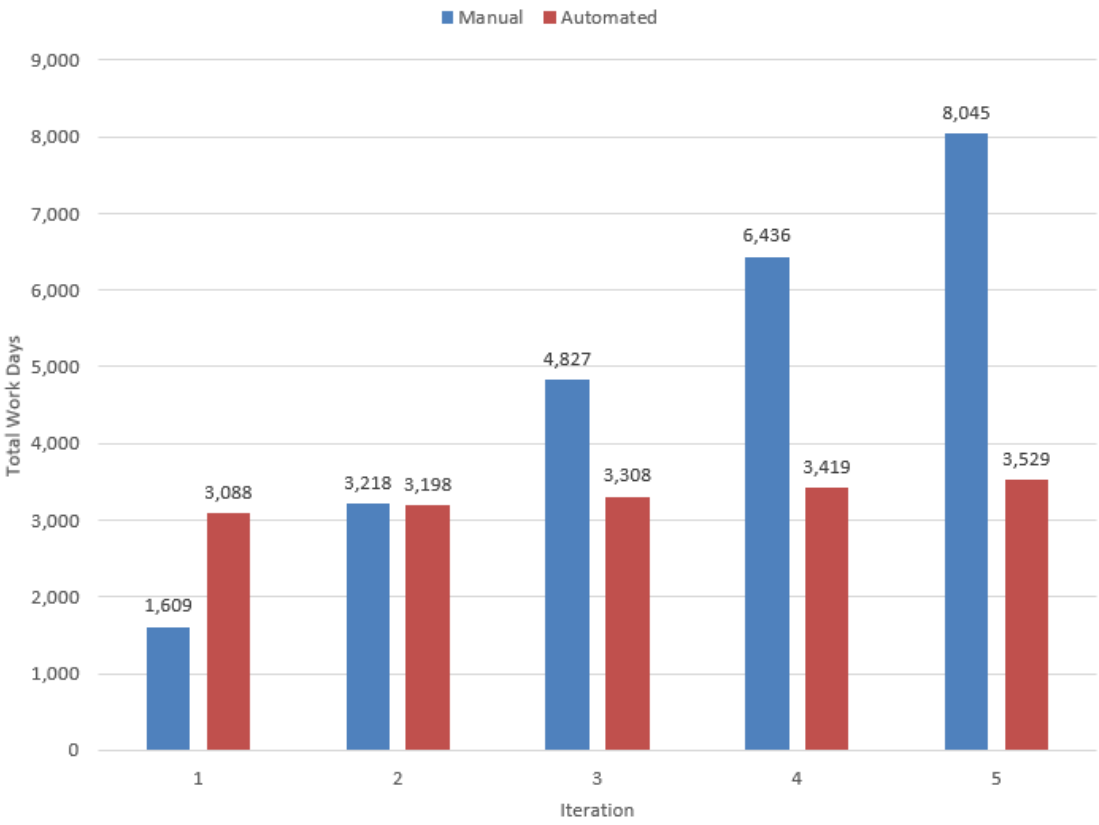

Figure 7. Time taken to test

The test execution time in automated testing is a great deal faster than manually; thus the time used to test manually increases rapidly. The equation used to calculate results illustrated in Figure 7 is presented in Equation 3.

$$\text{Total manual verification testing time} = A \times I \times V$$

$$\text{Total automated verification testing time in work days} = A \times [\frac{C}{7.5} + I \times (\frac{V}{24} + \frac{M}{7.5})]$$

Equation 3. Return on investment equations

In Equation 3, the variables are:

- A represents the requirements that have been automated at the moment
- I is the number of times the full verification run is executed
- V is the time taken to run full verification per requirement
- C is the time taken to create an automated test case per requirement
- M is the time taken to maintain automated test case per requirement

4.3 Challenges

There have been multiple challenges in the implementation of the automated testing for the client. The medical devices are complex; therefore creating automated test cases has been hard due to high learning demands to understand the software. One needs to understand medical abbreviations and how the medical devices work before implementing test cases. Also the testing environment is complex with different simulators and tools needed for testing. Manual testers have been implementing the automated test cases without proper knowledge of test automation; thus the implemented test cases have been poor and the creation of working ones has been slow. These challenges have increased the average test case creation time per requirement even further. These testers have since gone through automated testing trainings and are producing working test cases.

There have been changes in the requirements, for example a decision has been made to replace an existing feature by a completely new one. The original plan of the decided requirements to be automated has been re-evaluated, as some of the requirements could not be automated with a reasonable return on investment. A change in requirement may only be a rephrase of the requirement, or it might result in splitting the requirement to multiple new requirements. This increases the maintenance time: test cases are linked to requirements; thus the link needs to be fixed and the test cases need to be re-reviewed to ensure that they cover the new requirements.

There is a technical team which has been implementing libraries for different kind of simulators and for the medical devices. These libraries are necessary for communicating with the medical device and operating the simulators with the Robot Framework. Some of these implementations are either low quality, low level of abstraction or both. Low level libraries are directly used in the test cases, which add to the test creation time and maintenance time. At the time of writing this thesis, there are still old libraries which are cumbersome to use, but the new implementations have been better.

The technical team of the client company provides guidelines for the automated testing, although they are not themselves implementing the testing. These guidelines have been updated to a better condition, but they still are not enforced. These improvements

do not concern the already created test cases; thus the old test cases are not necessarily updated to be up to current standards. This will be detrimental in the long run for the overall status of the test harness.

The responsibilities split to different teams have led to that there seems to be not a single person looking at the big picture of the automated testing. For example, similar requirements across multiple teams could be automated together. There is a problem with the divided teams and them having different kinds of responsibilities. The problem is that the management is comparing the performances between teams. This has the effect that if one team helps another it has a negative impact in the performance of the team which lent the resources.

The reviewing of the automated test cases is done inside the teams and the reviewer might have no knowledge of how the testing should be done. This causes problems in the review stage when the reviewer does not understand the implementation of the test case. Also the test cases might be implemented poorly; thus the reviewer might not be able to recognize the mistakes made in the test cases. To minimize this, there has been change in the review process; there has to be a technical review embedded into it. This technical reviewer is familiar with the testing tool but might not be familiar with the testing domain.

Abstraction is necessary for the test cases to be easier to create. At the time of writing, the only abstraction between test cases and the System Under Testing is the navigation layer. However, no abstraction has been made for the simulators; thus the test cases directly refer to the needed simulator device. This leads to a problem in the long run as stated in chapter 3.3; if simulators are changed it affects directly to the test cases, rather than just in the abstraction.

The test cases may fail for finding a fault or due to the instabilities in the testing environment. In the client project, there are multiple different factors that might cause overall failure of test runs: there are network connectivity issues, the simulator might get stuck or the software can crash. The software can crash for having a 100% CPU load or because there is a defect in the software. Performance crashes happen due to automated tests stress the system more than manual testing, especially with the older

hardware. These flickering failures are hard to recognize and fix, due to the failures not being easily repeatable.

5 Discussion

The goal of the automated testing described in the previous section was to implement testing for 2,761 requirements in one year. However, in a year only 1,489 requirements have been automated. Thus the expectations have not been met. The expected time to complete an automated test case for a requirement was estimated to be faster than it is. The client's management thought that the manual testers could implement automated testing almost as fast as they had performed manual testing. The project can be deemed as a failure because it did not succeed in the goal which was set for it and had to be extended by a year.

However, the already implemented automated testing is faster than the manual testing of the same requirements. The 11,806 hours removed from the full verification run is a great deal of time saved; also the automated tests cover a wider area of the Software Under Testing. The time taken to achieve the automated testing for 1,489 requirements is 1,479 work days more than it would have been to execute a full verification run with manual testing. These results of the automated testing project confirm that automated testing should not be implemented if they are going to be executed only once. Automated testing yields barely positive return on investment after two full verification runs; thus it confirms that the strength of automated tests are repeatability and execution speed. This confirms that the regression testing should be automated.

The automated testing results do not take into consideration the time taken to build the test harness. The total average time to create and review one test case for a requirement might be even more. There are multiple different factors that can increase the total time it takes to create automated testing for a requirement, but these do not increase the time it takes to run a full verification test run with automated tests.

The automated testing project succeeds using the right testing tool for the task, which is the Robot Framework. Otherwise, the project has not been ideal for successful automated testing: management of the client company seem to be more interested in

creating competition between the teams than support them, as some of the requirements cannot be automated even though they were chosen to be; thus the planning has not been successful. While the automated testing was started, the possible limitations of it were not clear for the ones who planned and estimated the testing. The automated testing should have been introduced as a proof of concept with a couple of different automated test cases to get an in-depth view about the possibilities and restrictions of automation. It can be concluded that the automated testing project was started with a small amount of information about the challenges of automation.

The time spent working with the software and the environment has given more insight into the medical devices domain, thus helping with further implementation of automated testing. The project has also improved my skills as a test automator and the usage of the Robot Framework. Also the improvements in the automated testing tools have lowered the learning curve for one to implement automated testing. The Robot Framework has enabled non-technical persons to implement automated testing in the testing projects, although they might need to have trainings to correctly utilize the testing tool.

6 Conclusion

This thesis has studied why automated testing should be implemented and how it should be done. To illustrate software testing, this thesis at first introduced different methods, procedures, types and tools to test software. The tools were further divided into five categories in the automated testing project; the tool used in the project is a domain specific language tool called Robot Framework. Then automated testing was introduced, as well as its benefits and challenges. Subsequently, the means to succeed in automated testing were dealt with and a client case with results was analysed.

The thesis informs companies about the challenges of automated testing and also introduces means to avoid the challenges. The thesis can be used for example to identify problems in one's own automated testing project and also give perspectives for such projects. Further studying is required to get a better understanding of the cost differences between manual and automated testing.

References

Buwalda H. Key Success Factors for Keyword Driven Testing [online]. Logigear; 2015. URL: http://www.logigear.com/resources/articles-presentations-templates/389--key-success-factors-for-keyword-driven-testing.html. Accessed 5 October 2015.

Desikan S, Ramesh G. Software Testing Principles and Practices. Delhi: Dorling Kindersley; 2008.

Dustin E, Garrett T, Gauf B. Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality. Boston: Pearson Education Inc; 2009.

Graham D, Fewster M. Experiences of Test Automation: Case Studies of Software Test Automation. Crawfordsville: RR Donnelley; 2012.

Guru99. Automated Testing: Process, Planning, Tool selection [online]. Guru99; August 2015.
URL: http://www.guru99.com/automation-testing.html. Accessed 29 September 2015.

Guru99. What Is Black Box Testing? [online]. Guru99; August 2015.
URL: http://www.guru99.com/black-box-testing.html. Accessed 24 September 2015.

Guru99. What Is System Testing? [online]. Guru99; August 2015.
URL: http://www.guru99.com/system-testing.html. Accessed 24 September 2015.

Guru99. White Box Testing - Ultimate Guide [online]. Guru99; August 2015.
URL: http://www.guru99.com/white-box-testing.html. Accessed 23 September 2015.

Hewlett-Packard. HP WinRunner (WR) Software Version [online]. Hewlett-Packard; 2014.
URL: http://support.openview.hp.com/encore/wr.jsp. Accessed 14 October 2015.

ISTQB. Glossary [online]. ISTQB; 2014.
URL: http://www.istqb.org/downloads/glossary.html Accessed 24 September 2015.

ISTQB EXAM CERTIFICATION. What Is Agile Model - Advantages, Disadvantages and When to Use? [online]. ISTQB Certified tester; 2015.
URL: http://istqbexamcertification.com/what-is-agile-model-advantages-disadvantages-and-when-to-use-it/. Accessed 25 September 2015.

Janalta Interactive Inc. Integration Testing [online]. Techopedia; 2015.
URL: https://www.techopedia.com/definition/7751/integration-testing. Accessed 24 September 2015.

Li K, Wu M. Effective GUI Testing Automation: Developing an Automated GUI Testing Tool. Marina Village Parkway: SYBEX Inc; 2005.

Meerts J. The History of Software Testing [online]. Testing References; 2012.
URL: http://www.testingreferences.com/testinghistory.php. Accessed 22 September 2015.

Memon A. GUI Testing: Pitfalls and Process. IEEE Computer Society Press 2002;35(8):87-88.

Robot Framework. Robot Framework. [online] Robot Framework; 2015. URL: http://robotframework.org/. Accessed 29 September 2015

Robot Framework. Robot Framework User Guide [online]. Robot Framework; 2015. URL: http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html. Accessed 6 October 2015.

Selenium Project. Selenium Documentation [online]. Seleniumhq; September 2015. URL: http://www.seleniumhq.org/docs/. Accessed 29 September 2015.

Software Testing Fundamentals. ACCEPTANCE TESTING fundamentals [online]. Software Testing Fundamentals; January 2011. URL: http://softwaretestingfundamentals.com/acceptance-testing/. Accessed 24 September 2015.

Software Testing Fundamentals. Gray Box Testing [online]. Software Testing Fundamentals; December 2010. URL: http://softwaretestingfundamentals.com/gray-box-testing/. Accessed 24 September 2015.

Software Testing Fundamentals. Integration Testing Fundamentals [online]. Software Testing Fundamentals; January 2011. URL: http://softwaretestingfundamentals.com/integration-testing/. Accessed 24 September 2015.

Software Testing Fundamentals. Software Testing Levels [online]. Software Testing Fundamentals; January 2011. URL: http://softwaretestingfundamentals.com/software-testing-levels/. Accessed 24 September 2015.

Software Testing Fundamentals. Software Quality Assurance [online]. Software Testing Fundamentals; December 2010. URL: http://softwaretestingfundamentals.com/software-quality-assurance/. Accessed 18 October 18 2015.

Software Testing Fundamentals. UNIT TESTING Fundamentals [online]. Software Testing Fundamentals; January 2011. URL: http://softwaretestingfundamentals.com/unit-testing/. Accessed 24 September 2015

TechTarget. unit testing definition [online]. TechTarget; 2007. URL: http://searchsoftwarequality.techtarget.com/definition/unit-testing. Accessed 24 September 2015.

Tutorials Point. SDLC -Waterfall Model [online]. Tutorials Point; 2015. URL: http://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm. Accessed 25 September 2015.

Tutorials Point. V Model – SDLC [online]. Tutorials Point; 2015.
URL: http://www.tutorialspoint.com/software_testing_dictionary/v_model.htm. Accessed 25 September 2015.