UNIVERSITY OF VAASA

FACULTY OF TECHNOLOGY

SOFTWARE ENGINEERING

Petteri Siponen

AUTOMATED TESTING VIA GRAPHICAL USER INTERFACE

Master´s thesis for the degree of Master of Science in Technology submitted for inspection, Helsinki, 20 April 2016.

Supervisor	Prof. Jouni Lampinen
Instructor	Prof. Jouni Lampinen
Inspector	M.Sc. EBA, M.A. Laura Lappalainen

PREFACE

This thesis has been a great challenge and it has taught me a lot. I want to thank my family and friends, my professor Jouni, and my girlfriend Kirsi supporting and giving feedback.

TABLE OF CONTENTS page

PICTURES page

TABLES page

VAASAN YLIOPISTO
Teknillinen tiedekunta
Tekijä: Petteri Siponen
Diplomityön nimi: Testausautomaatiota graafisen käyttöliittymän kautta
Valvojan nimi: Prof. Jouni Lampinen
Ohjaajan nimi: Prof. Jouni Lampinen
Tutkinto: Diplomi-insinööri
Koulutusohjelma: Tietotekniikan koulutusohjelma
Suunta: Ohjelmistotekniikka
Opintojen aloitusvuosi: 2007
Diplomityön valmistumisvuosi: 2016 Sivuja: 83

TIIVISTELMÄ:

Diplomityö käsittelee testausautomaatiota graafisen käyttöliittymän (GUI) kautta testattaessa Android-sovellusta käyttäen mustalaatikkometodia. Työn laajuus on rajattu GUIn testaustekniikoihin, jotka käyttävät mustalaatikkometodia ilman, että käytetään lähdekoodia. Myös muita GUI-testaustekniikoita ja ehdotettuja lähestymistapoja tutkitaan, erityisesti mobiilitestaamisen osa-alueella. Työn tavoitteena on löytää ratkaisu testiskriptin ylläpitämisen ja testiautomaation ajamisen tehostamiseksi. Toisena tavoitteena on löytää parhaita käytäntöjä testausautomaatiolle testattaessa sovellusta sen käyttöliittymän kautta.

Työ suoritettiin kaksivaiheisena. Ensimmäinen vaihe koostui kirjallisuustutkimuksesta keskittyen kolmeen tutkimusalueeseen: Android GUI-testausautomaatio, mobiili GUI-testausautomaatio, ja muu GUI-testausautomaatio. Tulosten selvittyä, ne jaettiin kahden otsikon alle: mobiili GUI-testausautomaation löydökset ja muut GUI-testausautomaatio löydökset. Työn toisena vaiheena oli tapaustutkimus, jonka aiheena oli Android-sovelluksen GUI-mutaatioiden testaaminen visuaalisen GUI-testauksen avulla.

Työn tuloksena löydettiin parannus graafisen käyttöliittymän kautta tehtävään testaamiseen käyttäen mustalaatikko-ympäristöä. Visuaalinen GUI testaus –metodin käyttö tuo tavoiteltua tehokkuutta verrattaessa lähtötilanteeseen, jossa Record and Replay -tekniikkaa oli alunperin käytetty. Myös havaitut lasilaatikkotekniikat antavat potentiaalisia tapoja kehittää testausautomaatiota graafisen käyttöliittymän kautta tehtävään testaamiseen hyödyntäen esimerkiksi GUI Ripping -tekniikkaa mallintamaan GUI sekä automaattisesti luomaan testitapaukset sekä -skriptit. Jatkon kannalta olisi tärkeää tutkia hahmontunnistuksen sekä koneoppimisen mahdollisuuksia kehittää GUIn kautta tehtävää mustalaatikkotestaamista. Myös sitä tulisi tutkia, kuinka GUI-muutokset voisi jakaa testityökalulle, jotta se voisi oppia ja ylläpitää oppimaansa tietoa. Lisäksi visuaalisen GUI-testaamisen ja GUI Ripping -tekniikan sulauttamista tulisi tutkia lisää, koska näiden yhdistäminen toimivaksi kokonaisuudeksi mahdollistaisi tukevan tavan testata sovellusta GUIn kautta.

AVAINSANAT: Testausautomaatio, GUI, Mustalaatikko, Android, Mobiili

UNIVERSITY OF VAASA
Faculty of technology
Author: Petteri Siponen
Topic of the Thesis: Automated Testing via Graphical User Interface
Supervisor: Prof. Jouni Lampinen
Instructor: Prof. Jouni Lampinen
Degree: Master of Science in Technology
Degree Programme: Degree Programme in Information Technology
Major of Subject: Software Engineering
Year of Entering the University: 2007
Year of Completing the Thesis: 2016 Pages: 83

ABSTRACT:

The subject of the thesis is automated testing via graphical user interface (GUI) when testing Android application by using black-box method. The scope of the thesis is limited to GUI testing techniques using black-box method without using the source code. Also other GUI testing techniques and proposed approaches will be studied especially in the field of mobile testing. The objective for this study is to find solution for making the maintenance of test script, and test automation runs more efficient. Other objective is to find best practices for test automation via GUI.

The research of this study was executed in two divided phases. The first phase was literature research focusing on three different areas: Android GUI Test Automation, Mobile GUI Test Automation, and Other GUI Test Automation. After found the results, they were arranged into two titles: Mobile GUI test automation findings, and Other GUI test automation findings. The second phase was a case study of using Visual GUI Testing technique for testing Android GUI mutations.

The outcome of this study was the found improvement for testing via graphical user interface in black-box environment. Using Visual GUI Testing method gives efficiency in testing via GUI compared to the initial situation where Record and Replay technique was initially used. Also the found white-box methods are giving considerable best practices for improving test automation via GUI e.g. using GUI ripping for modelling the GUI and generating automatically test cases and test scripts. The future studies could focus on how the pattern recognition and machine learning could help GUI testing in the field of black-box techniques. Other future study aspects could focus on how the GUI changes could be informed to the test automation tool to learn and gain knowledge of the changes. Also the implementation of Visual GUI Testing and GUI Ripping should be studied more for making a robust method for GUI testing.

KEYWORDS: Test Automation, GUI, Black-box, Android, Mobile

1. INTRODUCTION

Quality plays a big role in software development and it has risen to a significant role. The reason behind this growth is that users are very quality driven and it is easy to switch the software or application to another if you aren't happy to its content or quality. The product needs to be tested and quality assured properly to meet the product specification and also the consumers' expectations. Mobile markets have given a full range of different choices for the user and this is why it is easy to switch to using a "better" application.

There has always been awareness of testing and examples have shown what can go wrong if something hasn't been tested properly. Testing is needed mainly to save the loss of money. In mobile markets, as in any other markets, testing is needed to secure that everything is working properly. If your product isn't, the user will find another application that will work and the company will loose the customer and their money.

The supply of mobile device on the markets started the rapid growth of mobile applications markets at the same time. This phenomenon grew the competition in the application markets. Latest report from Ericson shows that at the end of year 2015 there was 7,3 billion smartphone subscribers globally and growing yearly around three percentage (Ericson Mobility Report 2016). This tells the size of the force there is to move the markets and it has become this big very quickly. It also enables the possibility for the customer to switch the application if it doesn't satisfy.

Now that we have electronic devices and different kind of gadgets around us has made us very quality driven consumers. People are keen on the quality and also very instantly changing the product or application to the next one if they are not happy with it. As said earlier this leads us to testing and why it's important to ensure the level of quality of the product. To reach the level of preferred quality, the product needs to be tested properly and it takes time and money. To cover fully all the main areas and most urgent testing tasks, it can be costly and on most of the products the overall costs need to be cut to the minimum. This means also cutting the costs from the testing. As it is commonly known the testing is needed earliest as in the beginning of the specification of the software development and throughout the whole development life cycle. Reason why testing should be paid attention in the very beginning of the process is because of the costs of the found bugs are low and better quality can be reached easier. All this and the

repeatable manual tasks have led the testing into the automation, the next level of testing. With automation the testing can be done in a more efficient way and this also saves money and time. Normally test automation is used for example to speed up the testing, to break the software more quickly or more effectively in a repeatable manner, or help execute complex test cases in a more timely manner. Of course the test automation isn't the solution to minimize the entire manual testing, but it is a great effort to lower the budget in one way. And if we focus on test automation and one of the issues that brings the costs up is the maintenance of the scripts and test cases. Test cases need to be checked and updated accordingly, if there are any changes in the product or design documents and some parts of product have been developed and changed. The maintenance of test cases and test scripts takes a lot of time and it needs to be done continually as the product will change and develop rapidly.

The premise for this Thesis is to find best practices to reduce the need of script maintenance and make the test automation more automated and intelligence than it is currently. In this thesis the focus will be on Android application's Graphical User Interface (GUI) test automation maintenance focusing on functional regression testing using black-box testing without instrumenting the source code. The object of this research is to find a more intelligent way to use test automation when testing GUI in a repeatable manner. The automated GUI testing can be done recording the actions for the test case by using the application in test and then modifying the scripts. Normally there are several different test cases recorded by using test automation tools. When some update has been made to GUI and it normally changes the GUI layout some how, for example one main button has changed its position. In this case the recorded test cases using this button are no longer valid and they need to be updated to match with the current new version of the GUI. The scripts need to be updated manually or the whole recording needs to be done again. This iteration of maintenance doesn't end after one update and it continues as long as the development is running. This means a lot of work is needed for maintenance. Making the maintenance of scripts more efficient would also enhance the whole development process. The main focus will be studying the existing studies to improve test automation via GUI when using black-box approach. Also non-black-box methods will be considered whether they are proven to increase the GUI test automation efficiency.

The scope of this thesis will be limited to Android application's GUI's functional testing with test automation. In the section of Software Testing And Test Automation,

the basic information of the areas of testing, test automation, GUI and GUI testing, and Testability and Quality will be presented.

Research will be carried out in two sections and these are: Literature research and Case Study. Research focus is on literature research and it will be complemented with case study of a test automation tool and bring the available information together to give the best solutions for making the maintenance of scripts in test automation more efficient when testing Android application. These results will provide guidelines for test automation to make the test automation maintenance more efficient. This makes the maintenance easier and reduces the manual work. The whole study will be explored from the perspective of GUI test automation when functional testing Android applications in black-box environment.

2. SOFTWARE TESTING AND TEST AUTOMATION

This chapter will present the basis for this research of test automation via graphical user interface. The main areas in this work are Software Testing, Test Automation, Graphical User Interface, Approaches to GUI Test Automation, and Testability and Quality.

Software testing chapter will present all the information needed from the point of view of this thesis. The main areas to be focused are black-box testing, regression testing, end-to-end testing, acceptance testing, and GUI testing. In the chapter of Test automation the objectives of test automation will be presented and GUI test automation is also studied. Graphical User Interface chapter presents the basics of what is graphical user interface and what it is consisted of. In the chapter of Approaches to GUI Test Automation some of the approaches or techniques are presented and studied a bit more in detail. The last chapter in the section of Software Testing And Test Automation will be Testability and Quality. This chapter presents the ground for what is quality and testability in overall.

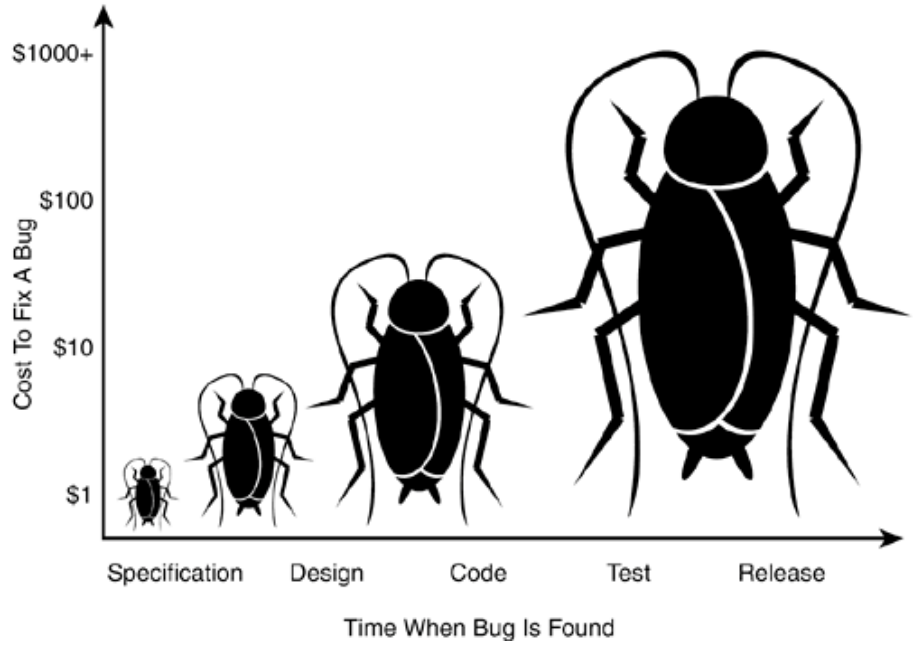
2.1. Software Testing

Information systems and software systems have extended their range widely to business and consumer products and as part of the daily process. Nowadays people are used to use different kind of systems or applications and quite many have experienced the situation when software is not working the way as expected. In case of malfunctioning software it can lead to several problems for example loss of money, time, or reputation. Worst case could be injury or death caused by software error. A human being with a mistake can cause these program defects. (International Software Testing Qualification Board 2011: 11–12.)

To avoid this kind of issues it is important to bring the testing as a part of the software development process and test as thorough as possible already from the start of the process. Before going any deeper to the testing, let’s start from the history. Where the testing started and how the name “bug” has established its role in software development.

The computer error has got its nickname, the bug, early in the 1947 when the computers were very big as filling the whole room. This occasion happened in Harvard University where the greatest computer at that time, Mark II, was built. The computer was on its run of different phases and it suddenly stopped. Technicians tried to find the problem and after a while they found a moth deep inside the computer. It had died to the high voltage and caused the computer to malfunction. This little moth was the reason to call computer errors as computer bugs. (Patton, Ron 2006: 11.)

Software bug can be defined with the specification of the product. If the software doesn't do something the specification has declared, or the software does or doesn't do something it shouldn't or it hasn't mentioned in the specification. Then of course, if the software has some usability issues like difficulties to understand, it is processing slow or hard to use then these actions can be also called software bugs. (Patton 2006: 15.)



Picture 1. Fixing the bugs can increase dramatically over time (Patton 2006:18).

It is important to start testing early in the software development process to avoid the big costs of a bug found in a later state of development, or worst, after the release. Bugs can be found at any state of a development process and early found bugs are cheaper to fix as it is shown in the Picture 1. If the bug is found while writing the specification it could cost $1. If the same bug were found in test round, it would cost from $100 to $1000. After release this bug could be from thousands to millions of dollars depending on the bug and at what state it would be found. (Patton 2006: 18.)

When software is being specified and the development starts, there are different states the product goes through in the software development life cycle. When the software has been defined and the specification has been made the testing starts also. During this cycle there are several of different type of testing done and here are couple of examples what kind of testing it is.

One of the first tests is *unit testing*, which will be done to different units e.g. software modules, subprograms, subroutines, classes or other procedures in the program that are smaller parts of the whole software. This way the software will be tested starting from the smaller units rather than testing the whole product at once. This way of testing also helps debugging when the bug has been found and specified to exist in a specific module or unit. The main purpose of unit testing is to test if the function of the unit is contradicting a specification of another function or interface defining the unit in test. Most of the unit testing is white-box oriented because it's feasible when you have a smaller set to focus on. Also you only need to find certain specific type of bugs while focusing on one unit than a bigger part of software. (Myers, Glenford J., Corey Sandler & Tom Badgett 2011: 85–86.)

White-box testing is one approach to test the software. White-box is opposite to black-box testing and is used for unit testing as mentioned previously. White-box testing may not be feasible if executed to bigger modules than smaller units of software. In white-box approach the tester knows what happens after giving an input to the unit or module and can observe how the outcome is produced. Whit this access to the code the tester can examine it and take some clues to his testing and this way find some weak points in the code. When executing the white-box approach it is possible to become biased and perform the tests to match the code's operation. This way you would loose the objective approach to test the software. (Patton 2006: 55–56.)

After unit testing the software by testing it in pieces, these pieces will be put together into larger ensembles. At his point the units have been tested and shouldn't bring up any surprises. Next step from the lowest level testing is *integration testing*. Integration testing is done against group of units that have been integrated to work as a bigger set. This integrated set will be tested properly so that they work together and no bugs have been found. This process will continue incrementally putting together the units of the software and test that they work together. After all the units have been integrated and

tested the entire software has been put together and it is ready for next step, *system testing*. (Patton 2006: 109.)

After all the unit and integration testing is done the system testing can start. During system testing the software will be tested against its functional and structural stability and also the non-functional requirements will be tested, such as performance and reliability. If there is requirement for the software to interoperate with other software systems in collaboration, this will be tested during this phase. There can be organized a system integration testing, in case there are several systems to interoperate with and the interoperation may be very complex. System integration test is part of the integration testing. (Watkins, John 2001: 59.)

No we have presented some of the most used and most important parts of testing software. The field of testing contains many different phases and approaches of testing. And it depends on the project and budget what testing is explicitly needed and afford to run on the software. Here are couple of more phases and approaches to be mentioned.

One of the most important test areas is the software security. This area needs to be focused well by planning the testing very carefully to cover every corners of software security. Testing needs to use risk-based approaches so that they are heavily studying the architecture of the software and at the same time having the mind set of and attacker (Potter & McGraw 2004). Other important areas to test are compatibility and performance of software in order to function on different environments.

In compatibility testing, it is needed to make sure the software is interacting the way the user would expect when using software on a device that is supported by the software. So the compatibility means the software is interacting and sharing correct information with other software or system. This can be anything from copying text from text edit software and pasting it to another software, or mobile application working on different versions of the same operating system. The scope of software, platforms and devices is huge and to compatibility test all these can be a big task to achieve. It is normal to limit the scope to be more efficient. The key things, when limiting the set to be most effective, are popularity, age, type and manufacturer. With these criteria it should be easier to limit the scope of software, platforms and devices to be more reasonable. (Patton 2006: 141.)

Software performance is one of the important measures of quality software. If the software is functioning well and there aren't any errors with performance, the user doesn't notice anything and the software can carry out all the tasks without any delay or irritation of the user. In the opposite situation the user experience can be very irritating or even causing financial loss to the company. When testing the application performance all the different levels of application need to be tested. The higher level is the client, the software visible to the user, and hosting services. At the lower level are servers that run the software and network infrastructure. And if there are third-party service providers integrated, those need to be taken into account as well. Software performance can be measured with key performance indicators, service-oriented and efficiency-oriented indicators. Service-oriented indicators are availability and response time and efficiency-oriented are throughput and capacity. Availability is measuring the time of the software is available to the user. Response time is measuring the time to respond to the user request. Throughput is a rate of events occurring within a given period of time. And the capacity is presenting the used theoretical resource capacity. (Molyneaux, Ian 2014.)

Here was explained the testing in software development in a higher level and without going into the details and explaining all the methods and approaches of testing. Next subchapters will open more of the testing from the point of view of the research are of this thesis and explaining also more of the methods needed in GUI testing.

2.1.1. Black-Box Testing

Black-box and white-box testing was mentioned previously in this thesis and white-box was explained briefly. Here we are going to dive more into the black-box testing, because it is the main approach of GUI testing in this thesis. There are also two types of testing the black-box approach can be executed and these will be also explained later in this chapter.

Black-box testing approach is opposite to white-box testing. When using black-box approach the tester knows only what the software is supposed to do and what is the output if given a certain input to the software. Black-box testing doesn't give an opportunity to view the code to see how the software is operating. This limits the testing to test the software more from the user's point of view. (Patton 2006: 55.)

As mentioned there are two types of testing that can be used with black-box testing. These are static and dynamic testing. Static testing is a way of testing something that isn't running, e.g. examining and reviewing. Dynamic testing is then the "normal" way of testing software by using it in action. (Patton 2006: 56.)

Static black-box testing is the software specification testing and examining it for bugs. This static testing is performance of a high-level review of the software specification. It is not meant to find any specific bugs. Finding large fundamental problems or oversights are the main goal in static black-box testing. For static testing it is important to do background research on who will be the software users and research on similar software to have some knowledge of the competitive software. (Patton 2006: 56–57.)

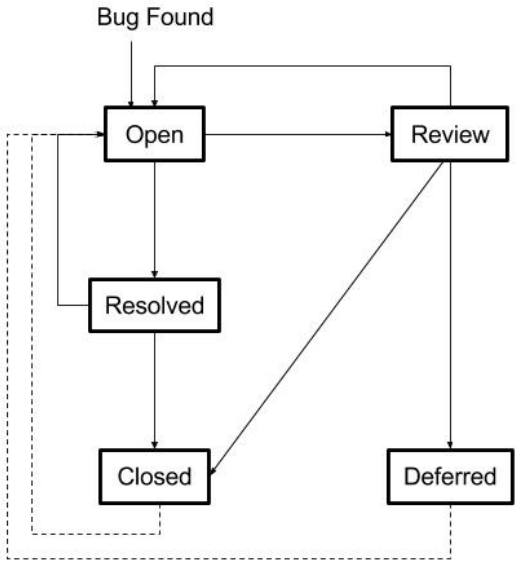
Now that we have covered the static black-box testing we can move on to dynamic black-box testing. Dynamic testing is basically testing of a running software and acting as an end user. As mentioned previously black-box approach doesn't give the opportunity to look into the code. The dynamic black-box testing approach is leaning on these requirements. This approach is also named behaviour testing because it is testing the behaviour of the software while it is used. For testing with dynamic black-box approach the software documentation or specification is needed to know how the product will behave. If given an input A, you need to know the output B so that you know the software is working properly. With the specification you can start creating the test cases for the testing. Black-box testing has two fundamental approaches that are test-to-pass and test-to-fail. With test-to-pass is meant to test the software in its normal behaviour and with out pushing it to its limits. Test-to-fail then tries to break the software and pushes the limits. (Patton 2006: 64.)

International Software Testing Qualifications Board's Foundation level syllabus presents techniques of black-box testing. First of the techniques is Equivalence Partitioning. With equivalence partitioning the inputs to the software are evaluated by their behaviour and then put into the groups that are expected to behave in a similar way. Other technique presented is Boundary Value Analysis. This analysis is useful technique with equivalence partitioning to test the boundaries of every partition. The boundaries are the maximum and minimum of a partition. Decision Table Testing is one technique to test system requirements that include logical conditions. Also the internal system design can be documented this way. The strength of this technique is that it creates the variety of different combinations of conditions that wouldn't normally be

executed during normal round of testing. Two last techniques presented are State Transition Testing and Use Case Testing. The state transition testing is a technique to test the different states of the software and the transition between these states. A state transition diagram can be created from the software behaviour. This technique is very useful and used mainly in embedded software development and in technical automation. Use case testing is then testing the software, as it would be actually used in real-world. Use cases can be described in abstract level and test cases are derived from the use cases. This technique is very useful in acceptance testing and it can also discover integration bugs. (International Software Testing Qualifications Board 2011.)

2.1.2. Regression Testing

Regression testing is a big part of test automation when you are running the specific test cases multiple times during the application life cycle. The application runs trough a great amount of test automation and these runs are checking the feature, or other content updates hasn't created any errors in the application i.e. running the regression tests.



Picture 2. These are the main states of the bug life cycle (Patton 2006:302).

Before going into the regression testing we need to explain a bit what is the life cycle of a bug. In *Picture 2* it has been presented in a very generic way and it can vary a lot in different companies how the bug life cycle runs. Main steps for the bug, after it has been found and reported, are Open, Resolved and Closed. During these steps the bug

will be reported and fixed and the fix will be regression tested to be able to push the bug to the Close state. There are also two additional states in the picture: *review* and *deferred*. In Review state the bug will be reviewed if it needs to be fixed. The deferred state is for the bugs that are review to be fixed but are not urgent and can be fixed later. (Patton 2006: 302.)

Regression testing can be divided in two "approach" where regression testing is needed. In overall the regression testing is always testing by using the tests that have been already executed at least once before (Tamres 2002: 223). The first one to be mentioned is regression testing a fixed bug. After a bug has been found, it will be reported and assigned to the programmer and the bug will be fixed. Before the bug can be pushed to closed-state it will be regression tested to assure it has been fixed correctly and it's not creating any new bugs. (Patton 2006:303.) Mainly the regression testing is known to use when e.g. an existing feature has been updated or modified and it needs to be tested whether it has created any errors or it is malfunctioning with other features integrated with it. This way it needs to be ensured the function or feature is working as it was in earlier versions. (Tamres 2002: 223.)

Ideally the regression tests should be run after every change made in the application or to its environment. This way the regression testing could be implemented to be part of the development and maintenance processes. The multi-level regression testing could be executed by first regression testing on the unit level and after that on the integration level. There are benefits when using this type of approach. First of all, when run the same tests on different levels it can reveal new bugs. The components can be run at the same time and the time used to detect the errors can be minimized. (Holopainen, Juha 2014: 8.)

2.1.3. End-to-End Testing

End-to-End (E2E) testing is one of the testing techniques as the unit and integration testing are. E2E testing is focusing on black-box testing and executing tests from the user point of view. Functional testing, which has been proven to detect effectively faults, is the main method on E2E testing. (Paul, Ray 2001: 211.)

When comparing E2E to unit and integration testing, E2E is not focusing on any single units as in unit testing. E2E testing is closer to integration testing but as said previously

E2E testing is application testing from the end user point of view and acting like a real user without focusing on testing any specific subsets of the application. When proceeding to E2E testing, the unit and integration testing should have been already executed and approved. E2E testing can be applied to any development processes because of its ability to be independent from any development process models and it can be also implemented in early states of the process to meet the requirements. (Paul 2001: 212.)

E2E testing follows the same steps of executing the tests for example in unit and integration testing. The steps are test planning, test design, test execution, and result analysis. (Bai, Tsai, Paul, Shen & Li 2001.)

2.1.4. Acceptance Testing

Acceptance testing is the one of the last tests done to the product and is ran after system testing. If some company has ordered the product, the real end user often executes these tests or at least Acceptance testing is done on the customer's site as a showcase the product is developed according to the requirements. (Tamres 2002: 223–224.)

Acceptance testing is the validator for the given requirements and the final check to verify the product is ready for release. Test cases in Acceptance testing are covering all the main functionalities and requirements that an expected user will proceed when using the product. (Tamres 2002: 223–224.)

Agile testing is one approach to whole idea of testing being a part of the agile development environment. This means that everyone is part of creating the test plan, even the customers by defining use cases and attributes for the test cases. In agile development cycle the customer starts acceptance testing the product already in early states of the cycle. This way the developer can get valuable information and testing becomes more integrated part of development cycle. In agile development process named Extreme Programming the testing has been divided in to two phases: unit and acceptance testing. Also tests need to be written before coding starts. As the customer designs the tests for acceptance testing he or she also executes the tests. This is because of the objective point of view and for assuring that the product meets the requirements. (Myers et al. 2011: 178–187)

Acceptance testing is the last checkpoint for the product either before its release to live environment in mobile applications or if delivering a product to a customer, then this check is for ensuring the product is in line with the specification and requirements.

2.1.5. GUI Testing

Graphical user interface (GUI) can be tested in couple of different ways. For example the focus can be either on unit testing the GUI or testing the system and its logic by focusing on the functionality via GUI. Both are important and complex tasks to perform.

GUI is combined of several different components that are also units. These units can be tested individually. Also GUI needs to behave correctly and for this category there's also a list of different kind of behaviors. Units inside the GUI are usually called elements or objects and they include all the buttons, windows, text boxes, menus etc. Examples of behavior in GUI are mouse clicks and movements, value displaying, disable, etc. All of these can be tested as different units before integrating them all to build up the GUI. (Hamill, Paul 2004.)

Other GUI areas to be tested are graphics, responsiveness, usability, and scalability. These are normally tested at least on low- and high-end devices to ensure the product is running correctly on different hardware and behaving as expected with the provided environment.

As mentioned, functionality testing is another way of testing GUIs. GUI is based on event-driven functionality and every input given to the GUI creates an event flow to the event listener and it will perform an action that GUI will display. Functional testing the GUI will give confirmation that the product is behaving correctly when given user inputs to the system. (Ruiz, Alex & Yvonne W. Price 2008.)

Automation is rolling in with a big force. But manual testing is still needed, at least on GUI testing. Manual GUI testing is useful way of revealing bugs. Manually found bugs can increase adaptability by leading to another similar bugs that haven't been found. Another thing that automation cannot measure or test is the usability and user experience of the product. For catching the usability issues might need an experienced tester, but still this is a task for manual testing. Manual testing is still needed, but the

automation has taken its place. It has been developed different kind of tools and methods to run test automation tests. GUI testing is taking a great amount of testing effort and that is why there are several different tools available. The tools aren't just for executing test, but also e.g. generating test cases, GUI model builder, and other supportive tools for test execution. (Yang, Xuebing 2011: 28.)

2.2. Test Automation

This chapter of test automation will present what is test automation and how it can help testing applications. Different tools for test automation will be presented. Objectives of test automation and GUI test automation have their own subchapters and these titles are looked into in a more thorough manner.

As we know, no testing or test automation is for free because of the amount of work to be done to achieve related benefits. To success in test automation you need more than just the tool acquisition. Robust background for test automation can be build with good test process, stable environment and realistic time budgeting. Here are some points to achieve success in implementing test automation: (Tamres 2002: 226.)

- Significant amount of time is needed first to program the test scripts and then to maintain the test suites.
- Programming experience is needed when programming test automation scripts.
- Tester needs training with tools to create right tests for automation.
- Good management is required to control the content related to tests and all files included.
- Automation is not guarantee of finding all existing bugs.
- New tests are needed while software develops and existing tests can get old and can't find new bugs.
- Awareness of that test tools themselves are software and can have their own bugs. (Tamres 2002: 226.)

Testing can be time consuming and repeatable work and automation is a great effort to make these tasks more efficient and less time consuming. Not to mention that with automating testing you reduce the risk of not finding the bugs. One other benefit of test

automation is that the regression testing can become easier. The main principals of using test tools and automation are *speed* - running test with tool can do the job many times faster, *efficiency* - while you set your tool to run tests you may also use your time to focus on cases the automation can't handle, *accuracy and precision* – human can make mistakes and focus can loose after running hundred of test cases, but a tool will perform the same work as planned without loosing the focus, *resource reduction* – with a tool it is possible to expand the resources needed by simulating the real environment, *simulation and emulation* – e.g. software or hardware can be simulated by a test tool to have a normal interference with the product in test, *relentlessness* – test automation and tools have all the energy to run tests you assign to them. Even this sounds very promising, but automation isn't always the right call. The manual testing can be in some situations the right choice and test automation shouldn't be applied. The whole test automation environment needs also a lot of time to be built and it doesn't come up in a second. Test tools are for making testing more efficient and faster, it's not a solution to put a computer to do your work while you're resting. (Patton 2006: 231–233.)

Patton (2006: 233–239) presents some tools for test automation. These tools are *viewers and monitors, drivers, stubs, stress and load tools, interference injectors and noise generators, and analysis tools.* A brief presentation of these tools will be given in this section next.

Viewers and monitors are tools for seeing the events behind the software user interface. White-box testing tools that help to see which part of the code or functions are being used and are they behaving as they should. One example of a viewer is a communications viewer, which enables to see the raw protocol data from i.e. the network. It works by observing the information and presenting it to another computer where the tester can observe and analyse the possible bugs. (Patton 2006: 234.)

Patton (2006: 235.) says that: "Drivers are tools used to control and operate the software being tested." With a driver it is possible to create a sequence of wanted actions to be executed. Drivers can execute programs or act as a mouse or keyboard for example by adding another computer to be the driver and give the needed inputs. (Patton 2006: 235–236.)

Stubs are also one type of drivers. They can receive or respond information and stubs are known as a white-box testing method. Patton gives an example that a printer can be

replaced with a computer, which will act as a stub and then the output can be analysed thoroughly. (Patton 2006: 236.)

Stress and load tools are for exercise of the software ability to handle great amount of stress and load. The system resources are playing a big role and available memory, disk space and files are affecting how the application is surviving. This could be done manually, but the tools are more efficient and helpful when testing this kind of part of the product. Load tools are more known in web environments by testing for example the servers load ability by simulating a situation with required set of connections and hits. (Patton 2006: 237–238.)

Interference injectors and noise generators have similarities to previous tools presented. For example the viewer tool can be modified to get an interference injector by switching the viewer with a computer that can view and alter the communication line's data. This interference injector can simulate communication errors of all types. (Patton 2006: 238.)

The last tool category to be presented is analysis tools. This category holds in different kind of everyday tools that makes it easier to proceed with daily tasks and can save time. Patton mentions couple of software types that are in this category: word processing, spreadsheet, database, file comparison, screen capture and comparison software, and for example debugger. (Patton 2006: 239.)

Common tasks in test automation are also *capture-playback tools*, *comparators* and *test execution tools*. The idea in capture-playback tools is to record events for example keystrokes, mouse activity and output on the display. This and other GUI test automation tools will be presented in chapter 2.3.2. Comparators help in determining if the test case is passed or failed by comparing two data sets. Test execution tools are for running the whole execution of the test. This includes application's initialization, data sending to the application, output capture that will be evaluated by comparator and finally results are filled in to the log. (Tamres 2002: 225–226.)

In cases that test is relying on the manual interaction, tests are plan to run only once or infrequently, or test evaluation is hard to program and it could be easier for a person to evaluate. These are examples of situation when test automation might not be good to be taken for consideration. Efficiency is not reached and automation isn't offering the excepted additional value in these cases. (Tamres 2002: 226.)

By test automation, regression testing can be made easier with possibility to run already existing tests whenever it is needed and quicker than human to perform it. With automated tests are manually made errors avoided for example typing mistakes and lapses of concentration. To fulfil requirements of the good test automation is an experienced tester needed for defining and creating good test cases. (Tamres 2002: 226.)

2.2.1. The Objectives of Test Automation

Objectives of test automation can be seen as cost and test execution time reduction, greater test coverage, and quality of testing can improve by the action of test automation. And when these objectives are reached it also can indicate greater product quality, product is more reliable and therefore can be shipped to market faster. But at the end, the main objective for automating testing is most of the time the cost reduction. (Mulder, D. L., & Whyte, G. 2013: 168.)

Moving from manual testing to test automation can improve the overall process of software development. Automation is also a great way to reduce the time on regression testing because the test automation can execute tests in a faster pace than manually executed (Rauf, E. A., & Reddy, E. M. 2015: 949). Rauf's and Reddy's research (2015: 955) also stated other benefits of test automation that are for example improvement in productivity, scripts were able to run on different platforms, and generally the test automation can be said to be fast, reliable, reusable and comprehensive.

As mentioned earlier automating testing can reduce the time used in testing. Tests can be set to run independently and tester can use the time for example to run more complex manual tests or increase the test coverage by adding more test cases for test automation. The provided possibility to run different tasks simultaneously by the test automation saves time and makes testing more efficient and test automation can run the tests without any break. (Cervantes, Alex. 2009.)

One thing that speaks for the test automation is the major possibility to make errors when testing and verifying results manually. Test automation can limit the human factor out. Other fact is the software development is mainly maintenance of existing code and for testing this means repeating the tests, i.e. running regression testing. This needs time

and effort and test automation can be very useful in this task. (Beizer, Boris 1995: 233–235.)

2.2.2. GUI Test Automation

There is a different kind of tools for automating GUI testing and each tool can provide different kind of ways to approach making the testing more efficient. In this chapter these approaches will be presented.

First approach to be presented is *Record and Playback*, also known as Capture/Replay. This method is one of the simplest ways to automate GUI testing. As the name can tell, you first need to record the wanted actions on the screen and then the recorded test case can be played over and over again, when testing is needed. Record and playback tools can record keystrokes or mouse actions and then play the actions in wanted playback speed. Also the mouse movements and clicks can be adjusted to be either absolute or relative to a specific window, this can be helpful because of the window position can change. (Patton 2006: 240–243.)

Next type of GUI test automation is to use *test monkeys*. This approach is a way to simulate what users could do. Test monkeys are tools that randomize the inputs given to the GUI. This is also why test monkeys aren't for running planned test cases. There are different states of monkeys: dumb, semi-smart, and smart monkeys. The dumb monkey is the most simplest and most straightforward tool to simulate random user inputs without knowing the software or its state at all. Next one of the monkeys is semi-smart monkey. Compared to dumb one, the semi-smart can e.g. maintain a log file of the actions. This would make it easier to find the cause of a crash bug if needed. The last type of monkeys is a smart monkey. This type of a tool can take the random testing to a more intelligent level. Smart monkey can act more like a user by knowing the states of the software and perform real action without randomly clicking around. This can give an enormous benefit while a big amount of time can be compressed to a fraction of it. (Patton 2006: 245–250.)

Model based GUI testing is one method of GUI test automation. With this approach it is needed to know the GUI states and events and models will be created based on this information. Model based GUI testing tools are able to run random test cases or then specific known cases if wanted. (Yang 2011: 31.) The GUI test models, which embody

the GUI's behaviour, are created either manually or by using an automatic tool for this purpose. The test cases can be created with the information of this model that gives the event sequences. A great advantage of model based approaches is the ability to generate concise test cases. (Bae, G., Rothermel, G., & Bae, D. 2014.)

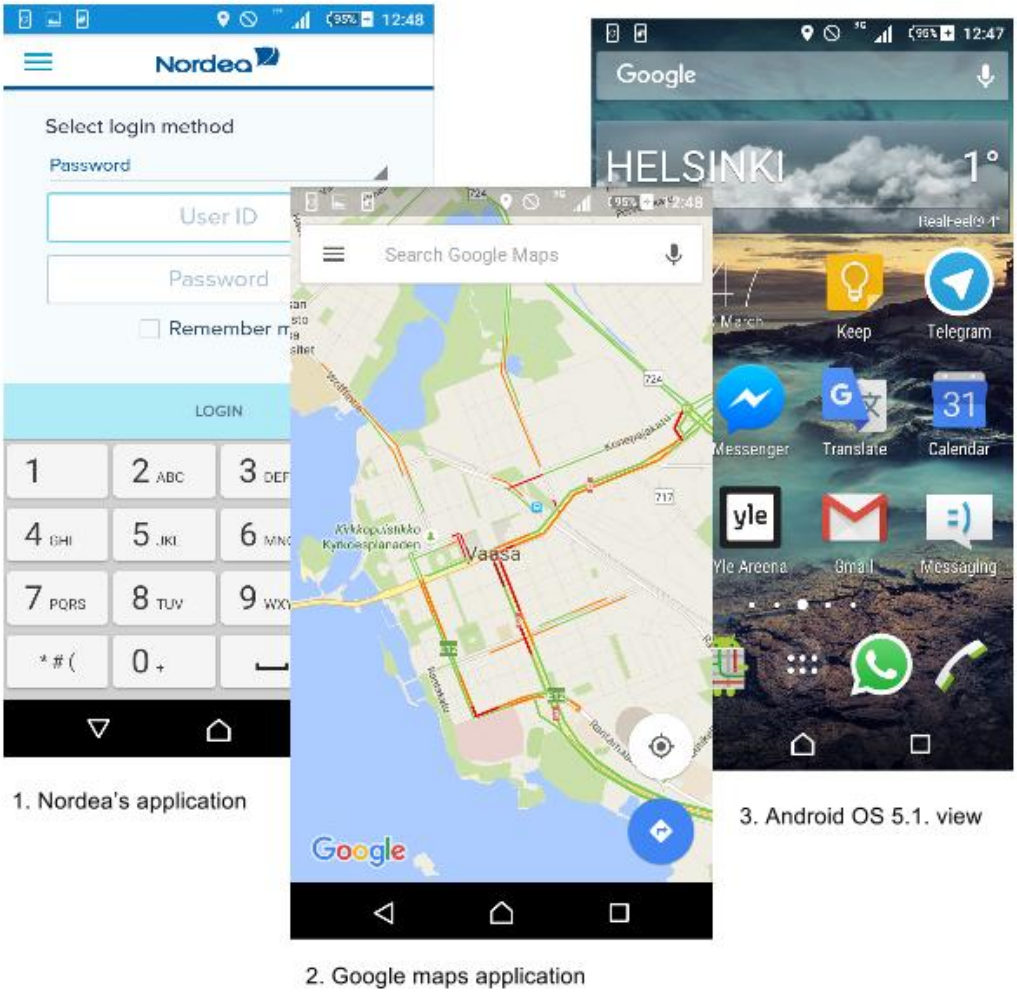
One model based approach is *event-flow model*, which gathers all the events and their interactions. This model's behaviour can be compared to control-flow model where the execution paths are represented, or a data-flow model that is representing all the memory location's uses and definitions. Event-flow model focuses on GUI events and event sequences. This way GUI can be seen as a hierarchy of events and all the different execution paths of events can be tested. (Memon, Atif. M. 2007: 138.)

Artificial Intelligence (AI) methods are taking more and more a bigger step in software testing. The techniques of AI can improve the quality and reduce the costs. Couple of examples of using AI in GUI test automation is test case regeneration after GUI has changed, use of an automated oracle for mapping the user interface behaviour, and AI planning techniques and genetic modelling have been used in improving the automatic test generation. (Rauf, Abdul M., & Alanazi, Mohammad N. 2014.) Also A. M. Memon presents a way using AI planning in GUI regression testing. The primary goal is to detect the changes in GUI and update the test cases by regenerating and rerunning them (Last, Mark, Kandel, A., & Bunke, H. 2004: 52–55).

2.3. Graphical User Interface

Instead of using command line to give input to the software, the user interfaces have been a while around by using graphical user interfaces (GUI) for presenting information and taking user input to the software. The GUI's nowadays are consisted of different particles and features. These are usually presented as objects, widget or elements. These are common to our day-to-day behaviour for helping to use daily activities and the GUI has for example menus and buttons. The other main elements used in GUIs are windows, scroll bars, drop-down menus, and informative icons. Depending on the GUI, it can be accessed by using the either keyboard, mouse, or, like in mobile devices, using touch input. The exact definition of graphical user interface is presented next: (Xuebing 2011: 49–50.)

> *A Graphical User Interface (GUI) is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events which are from a fixed set of events, and produces deterministic graphical output. A GUI contains graphical objects; each object has a fixed set of properties. At any time during the execution of the software, these properties have discrete values, the set of which constitutes the state of the GUI.* (Xuebing 2011: 52.)



Picture 3. Different mobile GUIs.

Today's graphical user interfaces are very intuitive and rich. As presented in Picture 9, the three different screenshots are showing the variety of the GUIs. The banking application (1) is using integrated keyboard. Google maps (2) is simplistic and presenting the map that can be zoomed and swiped by touching the screen. And Android OS 5.1 (Sony Xperia Z3 Compact) is also presented (3) and there can bee seen a basic home screen with application icons, and time and weather widgets.

GUI isn't just the objects and widgets the end user sees. There is another aspect of GUI when developing a product or application. The GUI can behave differently if the environment changes. This is something to be focused on when developing and testing application. First of the main aspects that are affecting GUI is *platform*; every platform has its unique features to be taken into consideration. Different operating system versions of the platform are affecting also and on the mobile side they can have major differences that will affect the application. Also in mobile the range of devices is extraordinary, and different screen sizes and specification are affecting too. (Raut & Tomar 2014: 4–6.)

The software is developed following some architecture and GUI based application, especially on mobile, are normally following Model-View-Controller (MVC) architecture and this architecture has been versioned a lot. The basic idea behind is that the *model* manages the data storage, *view* will present the data to the user, and *controller* is responsible of controlling the user input. MVC can be very adaptive and meet the demands of high usability. (Luostarinen, Manner, Määttä & Järvinen 2010: 51.)

2.4. Approaches to GUI Test Automation

Approaches to GUI test automation will give an exploration to three different approaches of automating GUI testing. Every approach will give example of using test automation via GUI. The first one is focusing on use of framework in GUI test automation. The second one is Pattern Recognition And Computer Vision in GUI Testing. The last one is presenting GUI Ripper –technique for traversing the GUI structure.

2.4.1. Framework Use in GUI Test Automation

Test automation frameworks are used in testing and the purpose of frameworks is to make testers work a bit more efficiency. With a framework you can provide tools and services to support for testing activities. Framework offers an infrastructure for the test automation environment and can help develop test automation solutions. (Cervantes 2009.)

The simplest framework is the *Capture/Replay*, also can be known as *record and playback*. In this approach the user gives inputs to the system and these will be recorded. The inputs can be e.g. mouse movements or keystrokes. The recording will create a script which can be played back to run the test. These tests are simple to be created without knowledge in programming. But on the other hand, regression and maintenance can be error prone if the GUI changes and makes the test cases obsolete and fail. (McMaster, Scott, & Memon, Atif M. 2009.)

Another approach is *Data-driven scripts*. This framework is normally application-specific and the scripts can be created either manually coding or capturing by running the application. Data-driven script will use the given external data to deploy it to the input fields in the application. Negative side of this framework can be the hard coded components or fragile recognition strings that are in the script and these can fail by paralyzing all the scripts or even the whole application. Data-driven scripts framework is quick and easy to implement, but needs dedication in maintenance. (Nagle, Carl 2000.)

Test automation framework can be *keyword-driven* or also known as *table-driven*. This approach can typically make the framework application-independent. The tests are executed by using data tables that contain keywords. These keywords are describing the actions wanted to perform. They can also contain data for the application input fields and this data can be used also as a verification data. Application-independent framework will ensure the maximum reuse of the scripts and they can be easily kept updated. All though implementing keyword-driven framework most probably will be hard and time-consuming, but it will show the effort. (Nagle 2000.)

Hybrid test automation framework is combination of data-driven scripts and keyword-driven testing approaches. Keyword-driven architecture provides libraries and utilities for data-driven scripts by making them less prone to errors and to be more compact. (Nagle 2000.)

2.4.2. Pattern Recognition and Computer Vision in GUI Testing

Computer vision is trying to recognize shapes and details from the picture or object. The optimal case would be when the computer vision can analyse and fully understand the investigated object, like little child can name all the animals from the picture. Humans

can recognize people from the pictures and tell their feelings, or humans can separate and recognize the object from any background. This is partly due to the ability to perceive our surroundings in three-dimensional way, but there hasn't been found any explanation to how our visual system works. The field of computer vision has been researching for long time and researchers have developed mathematical techniques to work with three-dimensional view and recognize objects in the perceived surrounding or an image. (Szeliski, Richard 2010: 3.)

Pattern recognition has got its foundation in the 1960s and 1970s. With the gathered knowledge the modern-day challenges are tackled with the existing theories, ad hoc ideas, intuition and guessing. With pattern recognition it is possible to point out wanted or searched patterns from the investigated target. Kuncheva presents a classic example of use of pattern recognition where mail needs to be sorted by the digits that are handwritten. Examined target or object will have a corresponding class, which will define more about the found pattern. (Kuncheva, Ludmila I. 2014: 1.)

For recognizing objects or patterns from the picture or video can be difficult and lot of research has been done for computer vision and pattern recognition. Luckily there has been lot of improvement and research has come up with different kind of solutions. One common thing in computer vision and pattern recognition is that there always needs to be a model or description of investigated pattern or image. Because of this Shapiro & Stockman states that: "*many experts will say that the goal of computer vision is the construction of scene descriptions from images.*" Problem-centeredness and fundamental issues are still present in the field of computer vision research. (Shapiro, Linda G., & Stockman, George C. 2001: 1.)

Chang, Yeh and Miller (2010) have researched the use of computer vision in GUI testing. They are presenting an approach for automating GUI testing in their paper. This approach uses images to test the GUI by specifying the components to be tested. With given input events and images, the visual test scripts can be generated automatically.

Although the GUI test automation is very error prone task, this presented approach presents Sikuli, a tool that uses computer vision for creating GUI test automation scripts. After creating the test script and given the images, the Sikuli will look for the image from the GUI and execute the given actions. After this the tool will observe the outcome and verify the result or GUI feedback with the specified image. Sikuli provides

also a library of functions to help automating GUI tests. (Chang, Tsung-Hsiang, Yeh, Tom, & Miller, Robert C. 2010.)

Chang et al (2010) approach using Sikuli is a way to use computer vision for GUI test automation. In this approach the tool seeks the object either from the specified area or from the whole window. It is not explained how this tool processes the comparison of investigated object and image.

As Szeliski (2010) and Shapiro et al (2001) describes the computer vision is well researched and it still has taken only baby steps when compared to humans and animals ability to observe and perceive our surroundings. These baby steps are still giving a lot to the science and people in their daily tasks. In GUI test automation it has started to prove some advantages. Most of the comparison tools are comparing the object and investigated image by pixel operation and this can be quite inefficient and needs some level advanced logic behind the comparison process.

2.4.3. GUI Ripper

As known, testing needs test cases, which need to be maintained. Test cases can be created either manually or automatically. A tester does the manual work and automatically created test cases can be done with the model derived from the system specification. This isn't though that common to use the automatic test case creation because of the overall form of system specifications. For generating a test case to test GUI you can use Capture/Replay tool to interact with the system. It can take 20–30 minutes to generate a normal GUI test case. This kind of test case typically includes 50 different events. (Memon, Atif, Banerjee, Ishan, & Nagarajan, Adithya 2003: 260.)

Memon et al (2003) have described an approach to automatically generate test cases for GUI testing. This technique is called GUI Ripping: "*Reverse engineering the GUI's model directly from the executing GUI*". This dynamic approach can be used to a system's GUI that is currently active. From the launch of a system GUI ripping starts and it walks through all the windows and paths. It gathers all the available information, like properties and values, from the GUI elements in the current window. GUI's structure and execution behaviour will be formed with this technique. GUI's execution behaviour will be represented as *event-flow graph* and *integration tree*, and structure as *GUI Forest*. The information got from the executed system will be verified by the test

designer and after that the model is ready to be used in test case generation. (Memon et al 2003: 260–261.)

GUI Forest is the first ripping outcome of a system's GUI. Window structure of the GUI is now presented as hierarchic nodes in the forest. Each node contains window's state and it's widgets with values, and properties. (Memon et al 2003: 262.)

To be able to use the GUI forest for creating test cases, it is needed to have modelled structures for representing execution behaviour. These structures are called *flow of events*. GUI is consisting of different components and events are triggering different components. Flow of events is representing this behaviour and a flow graph is indicating a GUI component's flow of events. Event-flow graph is then presenting all the possible communication between events in a component. Integration tree is then describing the event-flows of GUI components and their communication with each other. (Memon et al 2003: 263–264.)

The approach that Memon et al (2003) described was strong act for creating new technique for generating GUI test cases automatically. Their focus was still quite narrow in the study and didn't cover the major platforms. Now there as been new development and research, and GUI ripping has reached Android platform as well. Amalfitano, Fasolino, Tramontana, Carmine and Memon (2012) have presented *AndroidRipper*, which is extending the work done on ripping. (Amalfitano, Fasolino, Tramontana, De Carmine & Memon 2012.)

With automated technique, AndroidRipper, an Android application can be tested via graphical user interface. In developing this technique the work of model-based GUI testing has been exploited in the study. AndroidRipper uses the GUI ripping technique, which was mentioned previously in this chapter. By traversing the application's GUI it systematically and automatically gathers the information from new events faced for generating test cases and executes them. This extended, configurable GUI analysis technique can examine the graphical user interface and notice crashes while application is running. (Amalfitano et al 2012: 258–259.)

In their study Amalfine et al (2012) compared the results of AndroidRipper to Monkey, which uses random testing approach, and AndroidRipper showed to be more feasible and cost-effective. Also compared to in detecting defects and covering the code,

AndroidRipper was more successful. The application tested was WordPress for Android, an open-source application. (Amalfitano et al 2012: 260.)

2.5. Testability and Quality

People are nowadays expecting a lot from the devices and software they are using. They have become very quality driven and aware and they are expecting that the software is working as intended. Other point of view is the requirements for the software or application that need to be implemented. When these are filled the product can be considered a quality product.

In following chapters the quality and software testability is presented. These chapters are not focusing on the end user point of view, it though will be presented, and focusing more from the technical side and in overall from the development life cycle side.

2.5.1. Testability

Software development has normally focused on filling the requirements of the version under development. This philosophy may lead to a point when testability decreases. One way to make software easier to test is to focus on design aspects by maximizing cohesion and minimizing coupling, this will make it easier to test the software. By cohesion it is meant how much the functionality is affecting other modules or functions. Coupling means how dependent, in number, the module is of others. If the software architecture and code is very coupled, it makes the tests also highly coupled. This brings it to the point where tests need to be updated continually and are fragile. The same situation can be considered when the code cohesion is low. When software has been designed and developed following best practices, less time will be used for unit and regression testing of the system. When the software has the cohesion and coupling on a proper level the goals of higher readability, maintainability, and testability can considered to be succeeded. (Alwardt, Mikeska, Pandorf & Tarpley 2009: 178–179.)

Other aspects of testability have been studied and next some of the main points are presented. The literature review has gathered the information and the data has been break down to four main categories, which are fault tolerance, exception handling, handling external influences, and miscellaneous issues. For *fault tolerance* there has

been proposed algorithms to find the weak spots of the software, also the fault observatory improvement could be improved with code instrumentation. Another aspect is to focus on events and their redundancy. For achieving greater testability, the events should be triggered in smaller groups or even individually. Then it would be easier to focus on the exact event. For increasing the testability the operational errors should be cut out, and software internal state should be observed. *Exception handling* can help in improving testability by improving observability and controllability. One proposed approach is to handle the captured exceptions in fault handlers and distinct them properly. Coding language has been also stated to affect the testability of an application. Another proposed approach is to implement the built-in testing features to the system as a testability improvement. *Handling external influence* is one aspect in building robust software and coding base. This can bring improvement in testability when coding has been done in a modular way. Architecture is also important aspect of testability and one proposed method is to implement test layers into the architecture for enabling the robust automated test runs. For dealing *miscellaneous issues*, an approach is proposing a method to be built in a system for detecting faults and false alarm reduction. Another proposition, for increasing testability, is that developers should follow strict coding rules for excluding miscellaneous issues. For excluding external issues, an approach proposes to be ran the tests on the original hardware. This way the test cases and requirements can be analyzed and a standard way of working can be followed. (Hassan, Afzal, Blom, Lindstrom, Andler, & Eldh 2015.)

2.5.2. Quality in Software

Vital components of successful software development and testing are presented next and what quality means in software. Tamres (2002: 213–218) has mentioned main character of software development to succeed in building quality software. These characters are requirements, project management, software configuration management, software quality assurance, and review and inspections. These titles are walked through next.

Establishing a common understanding between software developers' and the customers', good and clear *requirements* are inevitable. With requirements the system capabilities are represented from the user point of view and they give guidelines for designing the software. The system requirements define the specification for example

for internal interfaces, design boundaries, and performance issues. The final product will be evaluated by the requirements and testing validates the developed software comparing it to the given requirements. The testing is carrying out the validation that ensures the product holds the right requirements. With requirements the test cases can be defined and test cases will help simulating the input and output conditions. Early phase requirement reviewing defines if the requirement is reflecting correctly customer's needs. (Tamres 2002: 213.)

Project management is one of the most important things in the development process. Project management manages schedules and resources, and continually observes the status of the product. Project management in software development comes up in two ways as project planning and project tracking. What comes to software project planning it creates plans for performing software engineering and for managing the software project. Here are some key factors for good project planning: (Tamres 2002: 214.)

- Clear project objectives
- Definition of the development strategy
- Risk and risk mitigation plans
- Resource assignments
- Deliverables of the product
- Project is defined to smaller sections
- Schedule definition
- Standards to hang onto
- Clear communication between different stakeholders (Tamres 2002: 214.)

For sufficient visibility of actual process can be provided by software project tracking. If project's performance differs substantially from the planned activities the management can take actions to correct the situation. For tracking progress: (Tamres 2002: 214.)

- Major and minor milestones
- Forecasting and identifying of possible issues
- Planning actions for the situations if a problem occurs
- Observing the accuracy estimation, planned vs. actual progress (Tamres 2002: 214.)

Project managing is in an important role from the point of view of software testing. The testing is more than the test execution and consists of different phases and tasks. This normally takes a great segment of time from the whole schedule. When project management is well prepared and planned, the testing and quality assurance has the required time for their tasks. (Tamres 2002: 215.)

Software configuration management is responsible of the environment of software development and keeps track of the changes in different components during the development cycles. Software configuration management (SCM) handles all the documented data of the software, including the source code. SCM consists of different tools and these are version control, configure control, change control, build control, and process management. The use of SCM is using version control, which will keep track of the changes code in each component. Configure control gathers all the information and documentation for building a specific version of the software. Also the source code dependencies are memorized in these cases. Change control keeps track of all the changes related to the source code. Build control is for building the software automatically by collecting the information of the modules needed to rebuilding for that time. Other important part of SCM is process management that follows the progress on different level and evaluates the progress. When building a specific product environment software testing benefits from using configuration control. Also the documents and scripts related to testing can be managed with the help of configuration management. (Tamres 2002: 215–216.)

Software quality assurance is an aspect ensuring the quality in software and software development. Software quality assurance (SQA) is evaluating the software processes so that the tools, procedures, or techniques are not affecting any issues and affecting the software quality. Auditions are the main task of ensuring the development process and software quality are on the required level. Tamres states the SQA is held as a separate unit from the testing. (Tamres 2002: 213.)

Reviews and inspections are important from the quality point of view. It is important to find the fault in earliest state as possible. The reviews and inspections should be made on every phase of the development to find the bugs in earliest state. The different types of review are technical review, where the smaller groups are discussing of product related work. Walkthrough is type of a review and it guides through a document or code

the designer has normally written. Software inspections are the strictest type of review and they are constructed of a structured process and include checklists, comparing standards to reviewed items, reports, statistics, and records of areas shown weaknesses. Reviews and inspections are made to some type of documents and this is helping testing to produce test cases and test the documents to achieve greater quality in software. (Tamres 2002: 213.)

The previous chapters have been focusing on the quality in developing the software. Another point of view is the quality the user experiences. This is called quality in use. The measures of quality in use have been stated in standard ISO/IEC 9126. These quality categories are functionality, reliability, usability, efficiency, maintainability, and portability. Bevan states in the study that the software quality in use is measured as the ergonomic of the system and can be measured as effectiveness, productivity, and satisfaction of the users. (Bevan, Nigel 1999.)

3. LITERATURE RESEARCH

The research is divided into two sections: the literature research and the case study. The first one is literature research, which will scope the available literature in the field of GUI test automation from the point of view of test case and script maintenance and regression testing of an Android application.

3.1. Overview

This structured literature research will search and evaluate the available literature and gathers all the information, which is relevant for this research. Following channels were used for information retrieval for collecting relevant studies: Scopus, abstract and citation database peer-reviewed literature; Nelli, information retrieval portal for searching online material provided by Vaasa University; Google Scholar, bibliographic database with web search engine indexing metadata or the full text of scholarly literature. After searched the content from these information retrieval channels, the databases that were used are: Springer, Science Direct, ACM Digital Library, and IEEE.

Literature research is limiting the search to following search terms:

- “GUI Test Automation Android”
- “GUI Test Automation Android Regression”
- “GUI” AND “Test Automation” AND “Android”,
- “GUI” AND “Test Automation” AND “Android” AND “Maintenance”
- “GUI” AND “Test Automation” AND “Android” AND “Regression”
- “GUI” AND “Test Automation” AND “Mobile”
- “GUI” AND “Test Automation” AND “Mobile” AND “Regression”
- “GUI” AND “Test Automation” AND “Smartphone” AND ‘Regression”
- “GUI” AND “test automation” AND “regression”
- “GUI Test Script Maintenance”
- “GUI” AND “Test Script” AND “Maintenance’”

When executing literature search on Scopus, the search content was also limited to “Computer Science” for excluding irrelevant matches. After the search in each literature retrieval channel and also including a search of most active researchers on the field,

more than hundred articles were found. This group of articles were reviewed by reading titles and abstracts and excluding irrelevant papers out. From the first round survived 69 articles were selected to have a more specific review of content.

3.1.1. Statistics of Android GUI Test Automation Literature

Searching papers from Scopus with search terms of “GUI test automation Android” and limiting the search to “Computer Science”, the result was twelve documents. After adding “regression” to the search terms, the search gave zero documents corresponding to the search terms. When searching from Nelli, 79 articles were found in total corresponding to search terms “GUI AND Test Automation AND Android” and “GUI AND Test Automation AND Android AND Regression”. Google scholar had 122 articles with the same search terms and adding “Maintenance” to the terms. After reviewing the results by title and abstract the number of relevant articles were 30 articles.

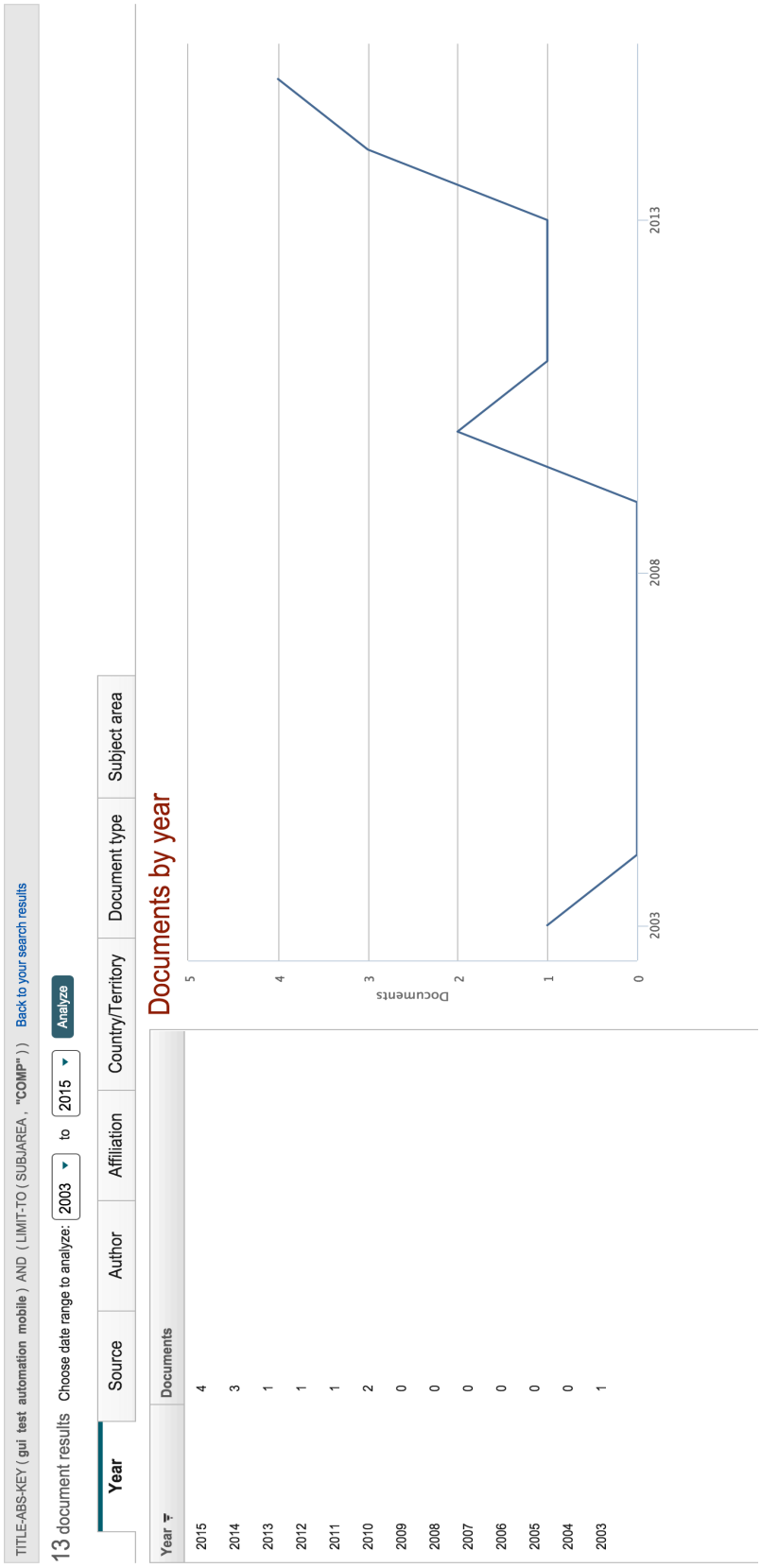
In the picture (Picture 4) the Scopus search is presented in diagram. The diagram presents the breakdown by years. The articles have been published after 2011 and the spike can be seen on the year 2014. Scopus also gave relevant information of the authors and this contributed the research on the field of the thesis is focusing.

Picture 4. Scopus search result with search terms: “GUI test automation Android".

3.1.2. Statistics of Mobile GUI Test Automation Literature

The focus of this thesis is on Android platform. For covering also other ideas and published research material from other platforms, the data was collected also from other mobile platforms and other GUI research. This chapter presents the statistics data referring to other articles related to overall mobile GUI test automation.

For developing the content search to be more efficient, only Scopus and Nelli were used in this search because the Google Scholar wasn't proven to be that accurate. The search terms that were used in Scopus were "GUI test automation mobile". This search gave in total 13 hits and they distributed by the publishing year (Picture 5) giving the weight to the beginning of the current century. The Spike is on the year 2015. In Nelli the search terms used were: GUI, Test Automation, Mobile, Regression, Smarthphone. These searches gave in total 229. The data was sorted by the relevancy and limited to 6 articles by reviewing the title and abstract. Most of the articles were the same as when searching content for previous chapter.



Picture 5. Scopus search result with search terms: “GUI test automation mobile".

3.1.3. Statistics of Other GUI Test Automation Literature

In this chapter other GUI test automation specific literature statistics are presented. Other GUI related research content is also studied to find possible solutions made for other platforms and learn from the these studies. Most of the GUI related studies aren't focusing on the Android and therefore this area needs to be researched carefully.

In this search also Nelli and Scopus were used and graph from the Scopus is presented in the Picture 6. This graph presents the breakdown of found content by the content's publishing year. Search terms used in this search were "GUI test automation regression" and the search was limited to computer science. From Nelli the searches were made focusing on GUI test automation regression and GUI test script maintenance. With these terms 455 articles were found. This amount of hits were considered to be unreasonably big and search result was filtered by the subject of the articles content, feature in Nelli, to exclude others than GUI, Software Testing, and Regression Testing. With this action the outcome of the search was 85 articles in total. After reviewing the articles by the title and abstract, 32 articles were selected for more detailed review.

Picture 6. Scopus search result with search terms: “GUI test automation mobile".

3.2. Literature Findings for Mobile GUI Test Automation

This chapter gathers all the found studies that were considered relevant for this research from the areas of Android GUI test automation and mobile GUI test automation. Other relevant points for focusing were black-box testing and test case and test script maintenance. Literature has been gathered and divided into different titles for reviewing them in a more structured way.

3.2.1. Capture/Replay

Liu, Chien-Hung, Lu, Chien-Yu, Cheng, Shan-Jen, Chang, Koan-Yuh, Hsiao, Yung-Chia, and Chu, Weng-Ming (2014) have developed an approach, which uses the Capture/Replay method for GUI test automation. In their work they are converting Android application's user events into test scripts of Robotium framework during the application run-time. Robotium is open source framework for Android test automation. The method for capturing the occurring events happens by overwriting the Android UI event handlers. All the event handlers are managed by the View component, which is also responsible for drawing, and the application under test needs to be instrumented by a mock layout that will execute the overwriting of UI event handlers. After the mock layout has captured the events, they will be analyzed and then test scripts are automatically generated. Any rich UI gesture events can be captured for test script generation. This approach is also extended to include assertion insertion so that the results of execution can be verified. The verification happens during the application run-time. For replaying the recorded tests, Robotium will execute them. Robotium is extending the AndroidTestCase, which is a JUnit-based testing framework. This means that every operation has been simplified, but still every operation consists of several low-level actions of AndroidTestCase. For replaying the recorded script, the test script is compiled and executed on the target device. Afterwards the application will send the results. (Liu et al. 2014)

3.2.2. Test Case Generation

Anbunathan, R. and Basu, Anirban (2014a) are proposing an approach in their work to use test scheduler and test cases as APKs. They have developed a framework for this purpose. In this framework an Android APK is developed to run the test cases. All the testing is done in Android environment without any connection between computer and

test device. Architecturally there is test scheduler and test cases. Scheduler is its own APK and every test case is build as an own APK. Scheduler invokes the test cases and monitors the execution. Test cases can be created in three different ways, depending on which Android building blocks have been used (Activity, Service, Async task). The options are short duration test cases, long duration test cases, and cases to be run in background. With this proposed approach all the Android resources can be used and it enables the possibility to create complex test cases. (Anbunathan, R. and Basu, Anirban 2014a.) From the test script point of view this approach can be considered as script-less test automation as the test cases are managed as APKs in the Android environment and it hasn't been presented how to maintain the test scripts.

Another test case generator has been proposed in study of Mariani, Leonardo, Pezzè, Mauro, Riganelli, Oliviero, and Santoro, Mauro (2011). This approach proposes a black-box testing tool for automatic generation of test cases. The tool is named as AutoBlackTest and it is focusing on crash and regression testing. AutoBlackTest is using machine learning to explore the available features and their combinations. The learning tool used is Q-learning that draws a model of the application's states, state transitions, and transition triggers. Q-learning uses two different parameters to analyze and learn the applications most relevant scenarios. This gains the experience during the learning process of the individual application's behavior. The learning part can be divided into state extraction, agent actions and behavior, and reward function. State extraction is presenting the state of the GUI visible to user. From this state all the widgets relevant information will be gathered. Agent is the executor in Q-learning and it has different actions to use. These actions can be either simple or complex actions. With a simple action agent a button can send click action. A complex action includes multiple widgets and deeper level of GUI actions. Agent behavior is guided to discover new ways to use the application for exploring relevant scenarios. These relevant scenarios are most likely to be highly-rewarding scenes. Rewarding and reward function are giving feedback to the agent of how well the application has been executed. This can be measured how the agent has affected the GUI with the actions. High rewarded actions are affecting the GUI change its state by producing totally new view compared to previous one. This kind of valuable action will give a great reward and increase the Q-value of the current action. After the learning has been executed and the model of the application has been built, test suites and cases can be created. There are plenty test cases that can be added to test suite, but if wanted to refine the size and time used the

test suite, Q-value can be used to include the most relevant cases. This value can be adjusted to fit the needs of testing. (Mariani et al. 2011.)

3.2.3. GUI Rippers and Crawlers

Memon, Banerjee, Nguyen and Robbins (2013) are focusing on in their paper to the GUI ripping evolution and how the GUI ripping has changed the GUI test automation. GUI ripping has been published first time in 2003 to improve the testing of GUI and to automate the manual work. Also an open source tool has been developed for GUI ripping, GUI Testing frAmewoRk (GUITAR), and this tool have been also extended to several platforms including mobile and web. GUITAR consists of four components: GUI Ripper, Graph Converter, Test Case Generator, and Replayer. GUI Ripper and Replayer will need platform specific plugins when operating on different platforms. As Memon et al. (2013) explains the history of GUI ripping and it's affect on GUI test automation, the great variety of studies can be observed from the publications. In this chapter some of the proposed approaches has been reviewed and considered relevant from the research point of view of this thesis.

GUI ripping has been used in many proposed approaches and Amalfitano, Amatucci, Fasolino, Gentile, Mele, Nardone, Vittorini and Marrone (2014) have also proposed an approach using AndroidRipper, a GUI ripping technique for android GUI testing, to improve code coverage. Amalfitano et al. have suggested improving the code coverage with using information of pattern from different levels, and generating additional test cases automatically. In this approach AndroidRipper is in a major role of testing the GUI of an application under test. AndroidRipper needs access to the source code to execute testing properly and creating the model and base for test cases. For creating the patterns this approach is proposing to focus on three different sources. These are software, devise and model. The flow for this proposition starts with AndroidRipper traversing the application for generating the model of the GUI and creating the first block of tests. Analyze of the GUI model will be done for finding possible matches with System Patterns (different levels' structures and behavior repetition. After a match has been found and a rule created for the match, a Test Specification Model will be created. Test Specification Model is presenting the GUI model and rules the Test Specification Patterns are describing. Then TestSeqGenerator, tool for transforming source models into analyzable languages with model transformations, creates test sequences into a language for AndroidRipper framework. Now AndroidRipper can stress test the

application with the set of generated tests. If the new generated test reveals yet uncovered code, it will be added to the repository. The AndroidRipper, when allowing process iteration, can update GUI model. This procedure ends after iteration there hasn't been found any test cases to be added to the repository. (Amalfitano et al. 2014.)

Anbunathan and Basu (2014b) are proposing a method for test automation to crawl the GUI menu path recursively for detecting occurring crashes. In this proposal Anbunathan et al. (2014b) are discussing about non-instrumentation of the source code by using UI automator to learn the GUI objectives in the APK under test. The crawler is also an APK; build on the elements like a normal Android application. The architecture also includes UI automator jar file, which contains two functional modules for learning and re-running. Learning module gathers information of the GUI objects, like GUI ripping does. Learning module also clicks the texts and observes will it open a new pop-up or page. Every UI objects are handled in the specific manner to the current widget. This way every widget can be specified and learned. Report of the test execution will be exported to Microsoft Excel file and the tracked path and widget information will be stored to Sqlite database. For ensuring whether a crash has occurred or not, the crawler compares the process names before and after starting the execution and giving input, item clicks, to the GUI. (Anbunathan et al. 2014b.)

Wang, Liang, You, Li, and Shi (2014) have focused in their study on Android GUI traversal and they are presenting tool called DroidCrawler. This tool is run on a host PC and application can be tested on a device or emulator. DroidCrawler architecture includes three main blocks. These blocks are GUI Extractor, User Interaction Emulator, and Crawl Controller. GUI Extractor gathers all the information of existing components and their properties in raw data. The list of GUI widgets and their attributes are extracted from the raw data. User Interaction Emulator is sending user inputs to the device while application is traversed from UI to another. Crawl Controller is following the GUI ripping strategy for building GUI tree where nodes are presenting GUIs and edges are presenting the user interactions between GUIs.

Many graphical user interface testing techniques are executing model-based testing method. Wu, Yumei, & Liu, Zhifang (2012) are focusing their research to create less work and constraints when generating model for GUI with Optical Character Recognition (OCR). OCR is using image processing to collect information of the characters. OCR approach is implementing the ripper method for traversing the GUI.

OCR Ripper is based on checking the caption strings from the GUI. The idea is that every widget contains a string. OCR is used for calculating the positions of the strings by modeling algorithm. After recognizing the possible widgets, click events are sent to the application on these positions. After the click event the graphical change is calculated for possible opening windows for example. If action has happened after sending a click event, the event and text item will be recorded as a widget and gather all the related information. (Wu et al. 2012.)

3.2.4. Automated Mobile Testing as a Service

Cloud services have provided, with a great variety of mobile devices, a new test model Test as a Service (TaaS). User can access this cloud environment at any time. Villanes, Bezerra Costa, and Dias-Neto (2015) have proposed a framework for this type of testing. They have named it as Automated Mobile Testing as a Service (AM-TaaS). This framework follows AQuA's (App Quality Alliance) test criteria for providing mobile application test automation. Test as a Service can be also known as Cloud Testing and it is most familiar of its scalability. TaaS is a web service with full cloud based test automation availability. It can execute testing on any layer of cloud computing (Infrastructure, Platform, or Software as a Service). Virtualization use, devices are emulated, and cloud infrastructure in use, are Automated Mobile Testing as a Service framework's main characters. AM-TaaS architecture is built on six components. First one is Cloud Controller, which manages authentication, reporting, and resources. Another component is Hypervisor. Hypervisor is responsible of virtual machines and the status of resource machines. Virtual machine is also an important component of AM-TaaS architecture and its main task is to allocate two other components as sub components of the Virtual machine. These components are Mobile Emulator Manager and Test Automation Manager. Mobile Emulator Manager runs the Android Operating Systems and emulates the environment of the requested OS version and unique features for the hardware and software. Test Automation Manager controls the test automation criteria and runs the MonkeyRunner on the emulated devices that the Test Automation Manager provides. MonkeyRunner is Android SDK based GUI test automation tool. The architectures last component to be presented is the Front End of the service. Front end can be accessed with any machine by using available web browser. When accessing the service, user will upload the application file (.apk) to the service and inform if there are some dependencies. After this the types of testing will be chosen and the device models. Mobile Emulator Manager starts the devices and Test

Automation Manager executes the test scripts on these devices. After the test run has been finished, user will be given a report of the test execution. (Villanes et al. 2015)

3.3. Other GUI Test Automation Related Literature Findings

The literature research was divided in three main areas of focus. Android and Mobile GUI test automation were discussed in the previous chapter and in this chapter the third, other GUI test automation related literature findings, will be reviewed. The reason for bringing this set of research was to research GUI test automation literature in overall and try to find possible solutions to be implemented on mobile side, and specifically on Android GUI test automation and bring additional information to solve the main issues of this research in the current thesis.

3.3.1 Test Case/Test Script Maintenance

In this chapter the focus will be on test cases and scripts and how the maintenance of them could be executed more efficiently. Scott McMaster and Atif M. Memon (2009) have proposed an approach that would use heuristics based framework for solving the GUI test case maintenance problem. This way the obsolete test cases could be updated to function with the updated GUI. This framework's key factors are: easy definitions of heuristics, heuristics are fitted to the sets, and facilitate experimentation support. Proposed tool to fit this framework is GUIAnalyzer, java based tool for analyzing applications with Swing toolkit. GUIAnalyzer has a specific AbstractHeuristic concept that serves for specific heuristic implementations as a base class. This component also decides if two components from different GUI models are a match. (McMaster et al. 2009)

Xun Yuan and Atif Memon (2010) have proposed a method for generating test cases fully automatically. This proposition is presented as a feedback-based technique. The feedback is gathered from the GUI in couple different manners. The first one creates the seed test suite from the smoke tests, which are the test cases of two-way GUI event interactions generated by the model-based algorithms. Another important feedback gathered is GUI run-time state that will be stored. GUI event-interaction graph (EIG) is used for gathering the information for the seed test suite. While the test suite is executed, the GUI widgets run-time state will be stored. This information is used for

recognizing the relationship, Event Semantic Interaction (ESI), between event pairs. The way an event can change another event's execution behavior can be recorded with the information of the relationship between events. Event Semantic Interaction Graph (ESIG) is formed automatically from ESI relationships. ESIG is then used to build the new set of test cases automatically. (Yuan & Memon 2010.)

Another approach is proposed in the work of Daniel, Luo, Mirzaaghaei, Dig, Marinov and Pezzè (2011). Their approach is giving an equivalent method as automatic GUI test repair approaches are using. In Daniel et al. (2011) approach the method is a proposal of white-box approach while in previous studies it has been made using black-box approach. In the black-box approach the GUI test repair automation is trying to deduce the changes in two different GUI versions. Daniel et al. (2011) white-box approach is refactoring GUI to understand the changes and translating the changes into code. Main target is to give developers an easy way to automatically refactor the GUI and get the mapping information for the GUI test script update. This precise mapping information will specify the evolution of GUI and enables test scripts update automatically. This proposed approach also excludes the checking of all test scripts. GUI changes will direct to the test scripts that need to be repaired. (Daniel et al. 2011.)

One proposed approach to be reviewed is SITAR Test Case Repair by Gao, Chen, Zou and Memon (2015). ScrIpt repAiRer (SITAR) is a reverse engineering technique for repairing obsolete test scripts automatically. The focus is on low-level test scripts. SITAR works to form a view of low-level scripts to function with higher-level models. After this the comparison happens between model-level test cases that are usable with the help of low-level test scripts abstraction. For the repair, SITAR uses GUIs broken event flow graph (EFG - GUI Ripping) models for combining the information to create new usable low-level scripts. The specific required information from the EFG is the logical sequences, which will be repaired to connect the test scripts to the right statements. SITAR is mainly automatic tool, but in some specific cases it might need human tester's input for repairing the script. After SITAR has repaired the test scripts, they can be executed automatically to run the test automation for the GUI. (Gao et al. 2015.)

3.3.2. Visual GUI Testing

Visual GUI Testing is comparable to Capture/Replay technique as they are both script based testing techniques. The difference is that Capture/Replay is using code or coordinates of GUI objects and Visual GUI Testing is an image recognition technique for searching bitmap objects in the GUI as buttons and windows. With this image recognition approach Visual GUI Testing is able to act as a user and test the application in a manner of black-box testing and still handling layout changes of the tested GUI. (Börjesson, Emil, & Feldt, Robert 2012: 350.)

Börjesson and Feldt (2012) are studying Visual GUI Testing in their paper. Their research focused on solving whether Visual GUI Testing is a better solution for acceptance and system testing in test automation or not when comparing to Capture/Replay tools. The study was executed comparing commercial tool and open-source tool Sikuli. Testing happened in industrial environment by testing non-animated security-critical software. As an object, both tools were set to automate and execute five test cases for the system under test. As an outcome, both tools were able to reach the set objects. The test case execution was also more efficient by decreasing the execution time 78 percent. This lowers the costs of regression testing and the time used for test case execution. Börjesson and Feldt (2012) state that the Visual GUI Testing can be used in system regression and acceptance testing. It though doesn't replace manual testing yet, but gives a great effort in improving efficiency. The subject for consideration in Visual GUI Testing is the handling of unexpected events from the system. (Börjesson & Feldt 2012.)

Alégroth, Feldt, & Olsson (2013) have presented a case study of the migration from manual testing to test automation using Visual GUI Testing. The company they were studying had used previously unit testing and Capture/Replay test automation tools and had problems with achieving in their tasks. Transition to using Visual GUI Testing affected test execution speed to improve, automated tests found unknown faults in addition to the faults found previously in manual testing. This can be also considered raising the software quality. One other benefit to be mentioned was that the Visual GUI Testing can be applied to test any software without the dependency of a specific operating system or programming language. (Alégroth, Feldt, & Olsson 2013.)

The previous papers of Visual GUI Testing have been using tool called Sikuli. The study written by Alégroth, Nass, and Olsson (2013) will present another tool, JAutomate, and its comparison to a CommercialTool, which name is not revealed, and Sikuli. This commercial Visual GUI Testing tool, JAutomate, was released in 2011 and is innovated by Michel Nass. Compared to other tools, JAutomate has functionality to record and replay the test case in addition to image recognition. This can be helpful in script generation. Although, the study cannot say, what is the maintenance costs of the scripts in JAutomate. The outcome of the study is summarizing that all the compared tools are competitive with each other. JAutomate has benefits over the other tools like several algorithms for image recognition, backwards compatibility, and scripts can include images. All the compared tools have different set of properties and for that reason can be used in different contexts. (Alégroth, Nass, & Olsson 2013.)

One of the latest studies on the Visual GUI Testing is a case study of comparing Visual GUI Testing with component-based GUI testing. The objective was to study the flexibility of GUI testing. The study executed the evaluation with two tools, GUITAR, from the point of view of component-based testing, and VGT GUITAR, from the Visual GUI Testing point of view. VGT GUITAR was a prototype of implementation of Visual GUI Testing to the GUITAR tool. Both studied techniques are differing from each other significantly as the component-based testing is fast, robust, and automatic with the limitation to the programming language of the tested application. Whilst Visual GUI Testing can be executed on every application with a GUI, but the fragile and slow execution of tests is limiting Visual GUI Testing technique. In the case study Alégroth, Gao, Oliveira, and Memon (2015) divide the study in two different sections. The first one is an experiment where they created 18 different mutations, faulty versions, of a simple Java application that they developed for the purpose of this study. The second one was a case study where they used both tools to test three open-source applications. As the focus was in this study on system and acceptance testing, the results showed that Visual GUI Testing is more applicable for acceptance testing and the component-based technique for system testing. Component-based technique also needs the access to the library of the application's GUI, and still it doesn't know what is rendered for the end user. The researchers are stating to combine these two techniques to create a more flexible way of testing GUI. VGT GUITAR is a step towards this combination, but as the study shows, it doesn't yet fully fill the requirements. (Alégroth, Gao, Oliveira, & Memon 2015.)

4. CASE STUDY - PREPARATION AND PLANNING

Previously in this paper the available literature has been researched in the field of GUI test automation and focused on finding new ways to improve GUI test automation maintenance problem when using Record and Replay (Capture/Replay) technique, which is a black-box method. This technique can be considered to be quite old and inefficient for robust, flexible and rapidly changing GUI's nowadays.

In this chapter, the case study will research a tool for implementing a new way of testing Android application functionality via its GUI. The aim is to implement a new black-box testing approach for testing Android application without instrumenting the application under test. One of the promising approaches, and chosen for this study, is Visual GUI Testing. This approach will be studied in this case study and evaluated whether it is suitable for Android application testing. The studies, reviewed in chapter 3.3.2, have been focusing on either web or desktop application Visual GUI Testing and therefore they cannot give a specific answer to this case study, but they are giving a great base to build on.

4.1. Definition of the Case

This thesis is focusing on GUI test automation of an Android application and how the GUI changes are affecting the test execution, test cases and test case maintenance. In the Literature Research chapter (3.) all the available studies of the GUI test automation have been filtered, reviewed and divided under two different categories: Android, and other GUI test automation. Now that we have all the available literature reviewed, we will supplement this thesis with the study of a GUI test automation tool. Because of the primary focus in this thesis is on literature research, this case study will be kept narrow and focusing only on simple test cases of testing the application with the GUI test automation tool.

To my knowledge and after thorough search, there wasn't found any tool specified to Android or mobile black-box testing and therefore the JAutomate, which was presented earlier in chapter 3.3.2, has been chosen for this study. JAutomate is currently the most promising and versatile for executing Visual GUI Test automation. As the tool is not

compatible on running Android or mobile applications, an emulator needs to be used. Xamarin Android Player was chosen for this purpose.

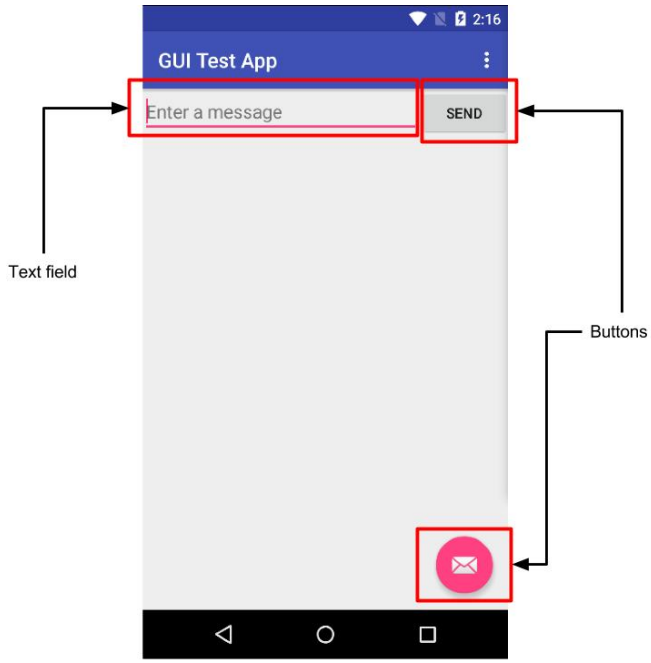
The background for this study comes from the record and replay technique. This technique is based on recording the actions and then replaying them in purposes of testing. The technique isn't the most agile and will break if there is any major or even any changes on the GUI layout. This study now investigates the Visual GUI Testing, as a comparable to record and replay technique, for updating this technique to the current century and for the needs of GUI testing. Main areas to be studied are: will the Visual GUI Testing succeed after the GUI has been changed, and will the JAutomate tool work in the environment where it will be stressed.

The case study now consists of the test environment, which is built on JAutomate and Xamarin. JAutomate is a commercial test automation tool, can be used also with a free licence, that uses Visual GUI Test, and record and replay techniques. JAutomate can be used for automating web and desktop application testing. For executing test automation on Android application we need to emulate the device and for this purpose Xamarin Android Player is used. Tested Android application will be simple with a few functionalities. This limits irrelevant faults from the testing and gives the opportunity to focus on the area of the case study. As the purpose of the case study is to observe the tools behaviour when GUI changes will occur, the application under testing will be modified and mutations of the GUI will be created for the testing purposes. The designed test cases will be the executed on the mutations and observed whether the tool succeeds testing the changed GUI.

4.2. Case Execution Definition

The case study will be executed on MacBook Pro (OS X 10.10.3, 2,6 GHz Intel Core i5, 8GB 1600MHz DDR3) with test automation tool JAutomation (version 1.0) and Android device emulator Xamarin Android Player (version 0.6.5). Emulated device will be Nexus 5 with OS version of Lollipop (OS 5.1, API 23). Application under test will be a prototype of an Android application called GUI Test App (version 1.0). The prototype was developed for the purpose of this thesis using Android Studio (version 1.5.1) tool. The GUI Test App has very limited functionality for enabling the focus only on the GUI changes and testing the mutations.

The study will start by creating the test cases for the application. After this phase, the application will be first launched by using the Xamarin Android Palyer emulator and then tested by using the test automation tool JAutomate. All the test cases will be recorded with the JAutomate and using the first (original) version of the application. After the cases have been recorded their function will be verified by running against the original version of the application. After this, all the mutations will be tested against the recorded test cases and results will be documented.



Picture 7. GUI Test App's original view and GUI objects are presented.

GUI Test App, presented in picture 7, has simple GUI and functionality is following the same design. The application contains two separate functionalities. First function is in the upper part of GUI and consists of two different objects. The objects are text input field "Enter a message" and button "Send" for submitting the text. After text insertion and button click, application will present the submitted text in another window. The other functionality in the lower part of the GUI is a button with the icon of a letter presenting as an email. After this button is clicked, application presents a message "Email is disabled" to the user.

Based on the documentation on the functionality of the GUI Test App, the test cases are presented in Table 1. The test cases are created based on the picture 5 and the

documentation of the functionality explained previously. Test case design is focusing on the functionality from the GUI point of view and there for the test cases are not going in to the details e.g. testing all the different input field's equivalent classes.

Table 1. Test cases for executing GUI functionality tests in GUI Test App.

Case #	Test Objective	Steps	Test Data	Expected result
001	Email button	1. Click the button with the letter icon		Text "Email is disabled" is presented to the user
002	Send button	1. Click the button with the text "SEND"		Empty "My message" view is presented
003	Send message - string	1. Insert text to the "Enter a message" field	"Hello there"	
		2. Click the "SEND" button		The inserted message is presented in "My message" view
004	Send message - int	1. Insert integer to the "Enter a message" field	89485	
		2. Click the "SEND" button		The inserted message is presented in "My message" view
005	Send message - char	1. Insert characters to the "Enter a message" field	"#€%/#€%&°^	
		2. Click the "SEND" button		The inserted message is presented in "My message" view

As the purpose of this case study is to measure how the built environment and Visual GUI Test tool can handle the GUI changes. For this reason ten different mutations will be created based on the original view of the GUI Test App application. These mutations are focusing on modifying the GUI layout and the widgets. The planned mutations are presented in the Table 2 in a text form. All the mutations are consisting of modification of the widget's layout, position, text, or removing the widget from the GUI.

Table 2. Different mutations of the GUI Test App.

Mutation #	Mutation Objective
001	"SEND" button has been removed
002	Email button has been removed
003	Text input widget has been removed
004	"SEND" button position has changed
005	Email button position has changed
006	Text input widget position has changed
007	Email button has changed its icon
008	"SEND" button's text has been changed to "SENT"
009	Text input widget's hint text has changed to "Enter a text"
010	"SEND" and Email buttons are overlapping

4.3. Objectives of the Case Study

The objectives for this study are to see how the Visual GUI Test automation tool manages to test the mutations of the GUI Test App and is the test environment suitable for testing Android applications functionality via GUI. Other objective is to study whether this tool and environment are fixing the problem of obsolete test cases in record and replay technique after minor GUI changes.

Expectations regarding the test cases and mutations, the tool should find the widgets after minor changes. But when there are major changes or the widget is missing, the expectation is that the test tool and the case will fail. One of the greatest expectations is the environment to work, and building a way to use Visual GUI Testing on Android application.

5. CASE STUDY - EXECUTION AND RESULTS

This chapter presents the results of the defined case study in the chapter 4. The case study has been divided in two different sections to separate the preparation state and the execution state. This part of the study will give a thorough view to the execution of the study. The execution will be documented and presented in this chapter under the first subtitle. The raw outcome of the case study is then revealed under next subtitle. Analyse of the gathered results and observations are stated in the last subtitle of this chapter.

5.1. Execution of the Case Study

The case study started by recording the test cases with the original version of the GUI Test App. The recording followed the steps in addition to the test cases for testing the GUI Test App's functionality:

1. Executing the Xamar Android Player
2. Starting the emulated device
3. Open the GUI Test App
4. Follow the test case steps

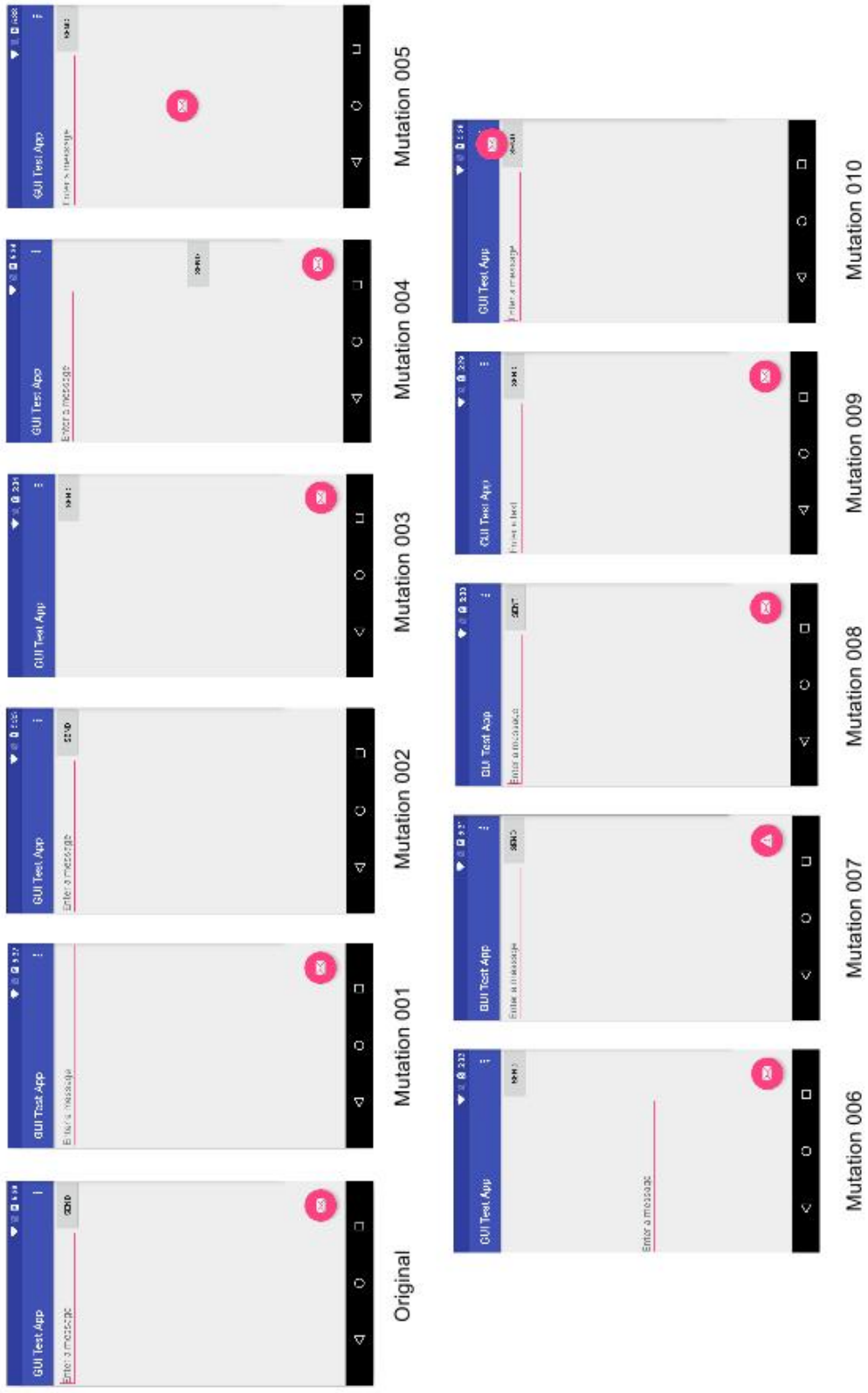
In the beginning of recording the test cases occurred some problems with taking screenshots. JAutomate took false images. Also launching the Xamarin Android Player and emulated device was very slow and the recorded test cases were optimized and wait calls were added to the existing script. The script was recorded to open the Xamarin Android Player application and execute the emulated device. Some user errors occurred in the beginning of the process e.g. realized the Xamarin should not run on the background when running the test case from the beginning. This caused the test case fail. Xamarin also failed to record test cases few times.

For the test case recording all the extra features were dropped out from the emulated device, for example live wallpaper and screen lock. These were causing test cases fail, but these minor issues were noted in the early state of the execution, and settings were configured. Next the test case 001 was tested against the original version of the GUI Test App. After 12 iteration rounds for adjusting the wait calls in the script, the test case was still failing radically. The wait calls were now waiting for 60 seconds and the

process was still failing. The recording process was changed to exclude the steps one and two. The new recording process started from the already running emulated device first opening the application and the executing the steps of the test case.

While recording other test cases some specific modification were needed. One was that a wait call needed to be inserted before verification or otherwise the test would fail. This took a while to be noticed. It was also observed the emulated device didn't recognize the inserted characters when inserting them using "Type" function. The characters were needed to be inserted using emulated device's keyboard by clicking the buttons individually. One JAutomate feature was also noted when changing the compared image in the script; the image was replaced in every place the script was copied.

Next the mutations of the GUI Test App were made for testing the recorded test cases against these mutations. While developing the mutations no changes were made and all the mutations were created according to the plan in table 2. In picture 8 all the different mutations are presented.

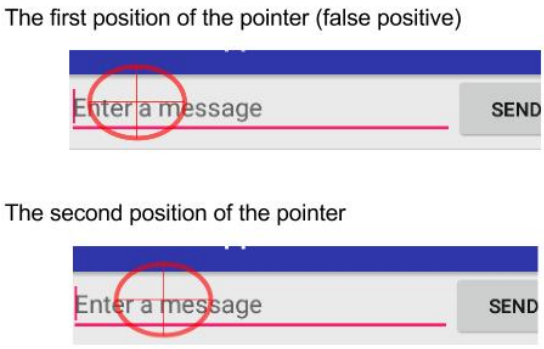


Picture 8. GUI Test App mutations.

After the test cases and mutations where created, the matrix of them was prepared for documenting whether the test case failed or passed. Each version of GUI Test App was first installed on the emulated device and then each test case was executed. At the point when all the test cases were ran the GUI Test App was uninstalled and next version was clean installed. Between each test case execution, the application was killed from the background and the device was set to present home screen.

First the test cases were ran for the original version of the GUI Test App to ensure the cases are recorded as planned. After this the first mutation was installed and test cases ran against it. All other mutations were also ran as planned.

Some issues occurred during the execution of different mutations. In mutation 003 the removal of text field caused the application crash. The text field was removed and this caused it. This was specific to test case 002 where no input was inserted to the text field. When the text field object was hidden, no crash occurred. In mutation 004 an unexpected failure of a test case appeared when the send button was hidden under the emulated device's keyboard. Mutation 006 showed how fragile the image comparison is; only a slight change on the pointer failed the test. Also the in the mutation 009 the sensitiveness of the pointer was recorded. It wasn't expected that the test cases 003–005 will pass, but it seemed that the change wasn't that small and it didn't affect during the comparison. The pointer area hasn't changed radically and therefore it doesn't fail. The position of the pointer was studied and after the half of the text didn't match the reference picture, the test case failed. In the picture 9 the upper image is presenting the case when the test will pass. The lower image would fail. The text in the images is from the reference images and the tested text was "Enter a text". The mutation 010 seemed to be the interesting because the test case 001 failed but the other test cases didn't. Even the email button is partially on top of the send button; it doesn't fail the test cases where the send button is used. When testing the email-button (test case 001) the pointer area had only very minor colour changes and it affected the failure. Compared to Send-button, even changing the position of the pointer when clicking the Send-button didn't cause the test case fail. When the pointer was located as top on the Send-button it still recognized the button and clicked it. This time it clicked accidentally the email button and test case failed because of this. The cause of this odd behaviour wasn't found.



Picture 9. Pointer comparison between failing and succeeding case.

5.2. Observations and Obtained Results

Recording the test cases was easy and simplistic with JAutomation and Xamarin Android Player. After iterations of using the tool and emulator some steps were excluded and the recording made easier. The learning curve was observed while using the tool and environment. JAutomation provides efficient way of using Visual GUI Testing by taking screenshots of the object to be clicked. The GUI of the JAutomation is built on using pseudo code and images. No actual scripting or coding is needed. This helps recording the test cases, and images are giving an efficient way to see what will be tested. Some minor issues were faced but these didn't affect the process after optimizing the process and excluding the opening of the emulator and emulated device.

Table 3 is presenting the results whether or not the test case passed. Here are the cases that showed some exceptions. The mutation 004, where the Send-button position has been changed, revealed unexpected issue. The Send-button was places horizontally to the right side and vertically to the centre of the GUI. After activating the text field object, the device keyboard appears and overlaps the Send-button. This caused the test cases 003–005 to fail. In mutation 006, the text field object had been repositioned. This also caused test cases 003–005 to fail. Reason for the failure was the pointer position in the original place contained also the top bar inside it. When the text field object had been moved, there weren't the same surroundings. This minor difference in the pointer caused these test cases to fail. Mutation 010 had also unexpected results. The Send-button ignored the overlapping Email-button totally. The expectation was that either the all test cases pass, or then they fail. But for some unknown reason, the Send-button overlapping with Email-button, didn't affect the pointers comparison.

When removing the object from the GUI, the test case outcome was as expected. Position changes didn't affect drastically the test case execution and the tool succeeded searching the objects. When Email-button icon was changed, the tool didn't recognize the button. The text changes were a bit harder. In Send-button case, the tool recognized the change immediately. But when studying the text input field, the tool failed the tests. The pointer seems to be sensitive in some corner cases.

Table 3. Results for the test run of each mutation.

Test Case # / Mutations	001	002	003	004	005
Original	P	P	P	P	P
Mutation 001	P	F	F	F	F
Mutation 002	F	P	P	P	P
Mutation 003	P	P	F	F	F
Mutation 004	P	P	F	F	F
Mutation 005	P	P	P	P	P
Mutation 006	P	P	F	F	F
Mutation 007	F	P	P	P	P
Mutation 008	P	F	F	F	F
Mutation 009	P	P	P	P	P
Mutation 010	F	P	P	P	P

Other observation was that when a button was missing or it had been repositioned the tool searched the button for more than 40 seconds. If tool didn't find the object, it requested manual verification. This was observed from the beginning of execution the test cases and it continued during every mutation and test case.

5.3. Analysis of the Results

The case study was executed following the prepared definition. The results of the case study were mostly according the expectations. Exceptions occurred on mutations 004, 006, and 010. In mutation 004, the issues can be fixed just hiding the device keyboard and cannot be considered a major failure from the environment and tool point of view. In mutations of 004 and 010, the issue was with the pointer. The comparison algorithm seems to be very sensitive and the error could be outlined in mutation 004 by reducing

the area of the pointer so that the area doesn't contain any particle from the outside of the objective. The mutation 010 needs more investigation because the minor changes in the tool or in the GUI didn't give any conclusive feedback. The pointer either ignores the Email-button, or the comparison algorithm isn't considering this big enough issue. Layer levels shouldn't affect, because the tool is comparing the pictures.

Test cases covered all the functions and some extra by giving different inputs. During the execution other test cases didn't raise awareness and therefore the planned test cases were executed without any case additions. Only change made for the test cases was the reduction of the test steps. This made the test run simpler. If the application would have more functions and deeper logic behind, it would have revealed some new issues, but the case was to study the tool, environment, and black box testing the application under test. With the simple application, all the logic-based errors can be limited out. Development of the mutations followed the planned steps and nothing wasn't added or excluded. The mutations were presenting the basic changes of the GUI and therefore can be considered that the mutations were giving the input needed for the tool. Some corner cases would have given more information of the tool. These cases could have been e.g. minor change in colour of the button, text, or the icon, or the change of the size of an object.

The JAutomate, Visual GUI Testing tool can be considered a great tool for black box testing an application. The minor changes of different objects were noticed and they didn't affect the test cases. When the object had disappeared or changed its appearance, the tool didn't succeed in this situation. The overall environment was functioning well. Only the opening of the Xamarin Android Player and the emulated device was taking a lot of time and can't be considered efficient way of executing the test cases. After all this wasn't the main part of the testing and therefore it isn't affecting the outcome of the study. Other point of view is that the testing was now executed only in emulated environment and the testing on a physical device hasn't been studied in this paper. The major amount of bugs is normally occurred on real devices and this should be the next step of testing this tool and environment. Also this version of JAutomate is not fully supported on Mac environment and can affect the testing. Windows environment should be considered in the next round of testing the environment in comparison.

6. RESEARCH FINDINGS

In this chapter the research findings are gathered and presented in two different sections following the research structure. First the findings of literature research are presented and case study findings are following afterwards.

6.1. Literature Research

Literature research was divided in three different sections. These were Android GUI Test Automation, Mobile GUI Test automation, and Other GUI Test Automation. After the search of the literature had been done, it was decided to divide the found literature under two separate titles: Mobile GUI Test Automation, and Other GUI Test Automation.

Literature research on Android and other mobile related GUI test automation yielded results, which have been categorized in five different titles. These titles are Capture/Replay, Test Case Generation, GUI Rippers and Crawlers, Optical Character recognition, and Automated Mobile Testing as a Service.

The literature related other GUI test automation was also studied for finding possible solutions that aren't yet used in the field of Android or Mobile GUI Test Automation. The found and studied literature was categorized in to two separate titles and they are Test Case and Test Script Maintenance, and Visual GUI Testing.

6.1.1. Mobile GUI Test Automation Findings

Mobile GUI Test Automation Findings are gathered under this chapter. The found literature is consisted of found studies of focusing the search on Android and other mobile studies on GUI test automation field.

Capture/Replay technique has been used for long time in test automation and Liu et al. (2014) have developed an approach using this technique and Robotium framework for converting user events into test scripts. This is executed using Android UI event handlers. Robotium is extending the JUnit-based testing framework AndroidTestCase.

For replaying the recorded script, the test script is compiled and executed on the target device. Afterwards the application will send the results.

Test case generation is focusing on maintaining the test cases and making script maintaining also more efficient. Anbunathan et al. (2014a) are proposing an approach in their work to use test scheduler and test cases as APKs. They have developed a framework where an Android APK is developed to run the test cases. All the testing is done in Android environment without any connection between computer and test device. The options are short duration test cases, long duration test cases, and cases to be run in background. With this proposed approach all the Android resources can be used and it enables the possibility to create complex test cases.

Mariani et al. (2011) have also studied test case generation and their approach proposes a black-box testing tool for automatic generation of test cases. The tool is named as AutoBlackTest and it is mainly focusing on crash and regression testing. AutoBlackTest is using machine learning to explore the available features and their combinations. The learning used in the tool is Q-learning that draws a model of the application's states, state transitions, and transition triggers.

GUI Rippers are also common technique in GUI test automation and Memon et al. (2013) are focusing on in their paper to the GUI ripping evolution and how the GUI ripping has changed the GUI test automation. An open source tool has been developed for GUI ripping, GUI Testing frAmewoRk (GUITAR), and this tool have been extended to several platforms including mobile and web. GUITAR consists of four components: GUI Ripper, Graph Converter, Test Case Generator, and Replayer.

GUI ripping has been used in many proposed approaches and Amalfitano et al. (2014) have proposed an approach using AndroidRipper, a GUI ripping technique for android GUI testing, to improve code coverage. Amalfitano et al. have suggested improving the code coverage with using information of pattern from different levels, and generating additional test cases automatically. In this approach AndroidRipper is in a major role of testing the GUI of an application under test. AndroidRipper needs access to the source code to execute testing properly and creating the model and base for test cases.

GUI Crawlers are similar to GUI rippers and Anbunathan et al. (2014b) have proposed a method for test automation to crawl the GUI menu path recursively for detecting

occurring crashes. In this proposal Anbunathan et al. (2014b) are discussing about non-instrumentation of the source code by using UI automator to learn the GUI objectives in the APK under test. The crawler is also an APK; build on the elements like a normal Android application.

Another study focusing on GUI crawlers is written by Wang et al. (2014). They have focused in their study on Android GUI traversal and they are presenting tool called DroidCrawler. This tool is run on a host PC and application can be tested on a device or emulator.

Optical Character Recognition is a technique for using image recognition in GUI testing. Wu et al. (2012) are focusing on in their research to create less work and constraints when generating model for GUI with Optical Character Recognition (OCR). OCR is using image processing to collect information of the characters. OCR approach is implementing the ripper method for traversing the GUI.

Cloud services have provided, with a great variety of mobile devices, a new test model *Test as a Service* (TaaS). Villanes et al. (2015) have proposed a framework for this type of testing. They have named it as Automated Mobile Testing as a Service (AM-TaaS). This framework follows AQuA's (App Quality Alliance) test criteria for providing mobile application test automation. TaaS is a web service with full cloud based test automation availability. It can execute testing on any layer of cloud computing (Infrastructure, Platform, or Software as a Service). Virtualization use, devices are emulated, and cloud infrastructure in use, are Automated Mobile Testing as a Service framework's main characters.

6.1.2. Other GUI Test Automation Related Findings

The other GUI test automation related studies are presented in this chapter. The focus has been on two topics: Test Case and Test Script Maintenance, and Visual GUI Testing.

Maintenance of test cases and test scripts are laborious and studies have been made on this field as well. Scott McMaster et al. (2009) have proposed an approach that would

use heuristics based framework for solving the GUI test case maintenance problem. This way the obsolete test cases could be updated to function with the updated GUI.

Yuan et al. (2010) have proposed a method for generating test cases fully automatically. This proposition is presented as a feedback-based technique. The feedback is gathered from the GUI in couple different manners. The first one creates the seed test suite from the smoke tests, which are the test cases of two-way GUI event interactions generated by the model-based algorithms. Another important feedback gathered is GUI run-time state that will be stored.

Daniel et al. (2011) are proposing a method of white-box approach while in previous studies of GUI test repair it has been executed using black-box approach. In the black-box approach the GUI test repair automation is trying to deduce the changes in two different GUI versions. Daniel et al. (2011) white-box approach is refactoring GUI to understand the changes and translating the changes into code. Main target is to give developers an easy way to automatically refactor the GUI and get the mapping information for the GUI test script update. This precise mapping information will specify the evolution of GUI and enables test scripts update automatically. This proposed approach also excludes the checking of all test scripts. GUI changes will direct to the test scripts that need to be repaired.

SITAR Test Case Repair by Gao et al. (2015) is also focusing on test case maintenance. ScrIpt repAiRer (SITAR) is a reverse engineering technique for repairing obsolete test scripts automatically. The focus is on low-level test scripts. SITAR is mainly automatic tool, but in some specific cases it might need human tester's input for repairing the script. After SITAR has repaired the test scripts, they can be executed automatically to run the test automation for the GUI. (Gao et al. 2015.)

Visual GUI Testing is studied more in recent years and the method is based on image processing and Capture/Replay techniques. Börjesson and Feldt (2012) are studying Visual GUI Testing in their paper. Their research focused on solving whether Visual GUI Testing is a better solution for acceptance and system testing in test automation or not. The study was executed comparing commercial tool and open-source tool Sikuli. Testing happened in industrial environment by testing non-animated security-critical software.

Alégroth et al. (2013) have presented a case study of the migration from manual testing to test automation using Visual GUI Testing. The company they were studying had used previously unit testing and Capture/Replay test automation tools and had problems with achieving in their tasks.

The previous papers of Visual GUI Testing have been using tool called Sikuli. The study written by Alégroth et al. (2013) will present another tool, JAutomate, and its comparison to a CommercialTool, which name is not revealed, and Sikuli.

Alégroth's et al. (2015) have executed one of the latest studies on the Visual GUI Testing comparing Visual GUI Testing with component-based GUI testing. The objective was to study the flexibility of GUI testing. The study executed the evaluation with two tools, GUITAR, from the point of view of component-based testing, and VGT GUITAR, from the Visual GUI Testing point of view. VGT GUITAR was a prototype of implementation of Visual GUI Testing to the GUITAR tool. Both studied techniques are differing from each other significantly as the component-based testing is fast, robust, and automatic with the limitation to the programming language of the tested application. In the case study Alégroth's et al. (2015) focus was in this study on system and acceptance testing, the results showed that Visual GUI Testing is more applicable for acceptance testing and the component-based technique for system testing.

6.2. Case Study Findings

The case study was executed by creating an Android application and ten mutations from the original one. Test cases were designed against the original version and recorded using Visual GUI Testing tool called JAutomate. The testing environment was set to be able to run the JAutomate on Android with Xamarin Android Player by emulating the device. All the mutations were tested against the test cases design for the original version of the application. Mutations included GUI changes.

The tool and environment did work as expected. Some minor improvements were made to the recording process and the steps recorded. In overall the tool is simplistic and easy to be used and no scripting is needed.

Testing mutation 004, test cases 003–005 failed. This was caused by change of the Send-button. The Send-button was placed horizontally to the right side and vertically to the centre of the GUI and activated keyboard overlaps with the button by hiding it. Also the same test cases failed in mutation 006. The text field object had been repositioned in this mutation and the pointer position in the original place contained also a part of the top bar inside it. After the reposition the same surroundings and top bar was missing and caused the test cases to fail. Mutation 010 was a version where the Send-button was overlapped by the Email button. The overlapping of these buttons should have failed all the test cases, but for unknown reason the Send-button worked normally.

The test cases outcome was following the expectation after removing the object from the GUI. Position changes didn't affect much the test case execution and the JAutomate succeeded searching the objects. When Email-button icon was changed, the tool didn't recognize the button. The text changes were a bit harder e.g. in Send-button case, the tool recognized the change immediately. But when studying the text input field, the tool failed the tests. Corner cases exist regarding to the pointer and the area in comparison.

The executing the tool and environment wasn't most time efficient. The test run execution took time in some cases and when recording the test cases, it had to be done step by step. The actual record action didn't record all the inserted steps.

7. CONCLUSIONS

The premise of this thesis was to find best practices for reducing the effort in test case and test script maintenance. The focused area was test automation when testing Android application via its GUI and focusing on the functional regression testing using black-box approach without instrumenting the source code. The objective was to find solutions to reduce the manual work when updating and maintaining the test cases, test scripts, and recorded test automation runs. The presented case was a simple situation where Record and Replay technique was used and the recorded test case clicks a button. When the button changes its position, the test case becomes obsolete and it needs to be updated manually every time there are changes in the GUI layout or made just to this button.

The findings that are producing some benefits and best practices, in field of black-box test automation of GUI without instrumenting the source code, are especially studies of Visual GUI Testing (Börjesson et al. (2012), Alégroth et al. (2013), and Alégroth et al. (2015)) and Optical Character Recognition (Wu et al. (2012)). The Visual GUI Testing is a great enhancement of the general technique of Record and Replay. The Visual GUI testing can make the test case maintenance more efficient. Based on the studies, Visual GUI Testing is most suitable on acceptance and system testing. Optical Character Recognition is a great effort in developing the system testing via GUI without instrumenting the source code. Only disadvantage is that the focus in Optical Character Recognition is on finding strings and counting on that the widgets always contains strings. There are a great amount of widgets with just an icon on them and this can become a problem in this approach. Other relevant studies were: the black-box testing tool for generating test cases automatically using Q-learning (Mariani et al. (2011)), and GUI crawler for black-box testing (Anbunathan et al. (2014b)). Q-learning is based on machine learning and it can be a very robust in testing system via GUI. The GUI crawler is using Android specific UI automator tool for learning the structure of the application GUI. The last study is focusing on Automated Mobile Testing as a Service (Villanes et al. (2015)) and this approach offers flexibility from the test environment point of view. The test case maintenance is not studied that much, but this gives a great effort in improving efficiency in overall.

The literature research yielded also results outside the field of black-box testing that were considered to be taken into account. These were Capture/Replay method with

instrumenting the source code for generating test scripts (Liu et al. (2014)), test case generation using running Android APK (Anbunathan et al. (2014a)), AndroidRipper for Android GUI testing and improving code coverage (Amalfitano et al. (2014)), GUI ripper called DroidCrawler for testing Android application (Wang et al. (2014)), heuristics based framework for solving the GUI test case maintenance problem (Scott McMaster et al. (2009)), feedback-based technique for generating test cases fully automatically (Yuan et al. (2010)), GUI test repair approach for refactoring GUI (Daniel et al. (2011)), and SITAR, reverse engineering technique for repairing obsolete test scripts automatically (Gao et al. (2015)).

The case study was executed by creating an Android application and ten mutations from the original one. Test cases were designed against the original version and recorded using Visual GUI Testing tool called JAutomate. The testing environment was set to be able to run the JAutomate on Android with Xamarin Android Player by emulating the device. All the mutations were tested against the test cases design for the original version of the application. Mutations included GUI changes. The result of the case study was that the Visual GUI Method is an improvement from record and replay technique and can reduce the maintenance effort in the presented research problem. The tool and environment were functioning well and only some minor issues were observed and they weren't affecting the case study results.

The case when wanting to test the system via GUI using only black-box method without instrumenting the source code, the best solution is to use Visual GUI Testing (VGT) method. This can be stated base on to the literature research and the case study. Using VGT the minor changes doesn't affect test cases to become obsolete. And cases when the GUI changes drastically only the images in the script need to be updated. Based on the case study the VGT is also competent in regression testing the system functionality. The other black-box methods are lacking some features compared to VGT and therefore cannot be considered equally great improvements.

The studies focusing on white-box methods are giving great approaches to improve GUI test automation. If the application can be instrumented or the source code is available, the Capture/Replay method is a giving the possibility to improve the test script maintenance. Also the GUI ripping technique is harvesting the GUI and generates the model of the GUI, and test cases and scripts according to this information.

The objectives have been succeeded in this thesis to make improvement from the Record and Replay method when the test case fails after objective has been repositioned. This improvement has been achieved with the Visual GUI Testing technique. Other benefits are the best practices found for other approaches in the field of GUI testing e.g. GUI Ripping, and the result of test case for using VGT for Android in emulated environment.

Future studies should focus on how the black-box testing of GUI could be improved to be more intelligent. How the pattern recognition and machine learning could help in GUI testing without having the access to the source code. Other aspect of research could be focusing on how the development process could give feedback of GUI changes for the testing tool to gain knowledge in changes and keeping the test runs updated according to these changes. Visual GUI Testing implemented with GUI Ripping have been introduced but it needs also more studies. It can become a very robust method for system testing via GUI.

REFERENCES

Alegroth, Emil, Feldt, Robert, & Olsson, Helena H. (2013). Transitioning manual system test suites to automated testing: An industrial case study. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)* [online] [25.2.2016], 56–65. 18–22 March 2013, Luxembourg. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=6569716 DOI: 10.1109/ICST.2013.14.

Alégroth, Emil, Gao, Zebao, Oliveira, Rafael, & Memon, Atif (2015). Conceptualization and evaluation of component-based testing unified with visual gui testing: An empirical study. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* [online] [27.2.2016], 1–10. 13–17 April 2015, Graz, Austria. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=7102584. DOI: 10.1109/ICST.2015.7102584.

Alegroth, Emil, Nass, Michel, & Olsson, Helena H. (2013). JAutomate: A tool for system-and acceptance-test automation. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)* [online] [26.2.2016], 439–446. 18–22 March 2013, Luxembourg. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=6569758. DOI: 10.1109/ICST.2013.61.

Alwardt, Anthony L., Mikeska, Nathan, Pandorf, Richard J., & Tarpley, Philip R. (2009). A lean approach to designing for software testability. *2009 IEEE Autotestcon* [online] [10.3.2016], 178–183. 14–17 Sept. 2009, Anaheim, CA, USA. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=5314039. DOI: 10.1109/AUTEST.2009.5314039.

Amalfitano, Domenico, Amatucci, Nicola, Fasolino, Anna R., Gentile, Ugo, Mele, Gianluca, Nardone, Roberto, Vittorini, Valeria, & Marrone, Stefano (2014). Improving code coverage in android apps testing by exploiting patterns and automatic test case generation. *Proceedings of the 2014 International Workshop on Long-Term Industrial Collaboration on Software Engineering* [online]

[15.2.2016], 29–34. 15–19 Sept. 2014, Västerås, Sweden. New York, NY, USA: ACM. Available: http://dl.acm.org.proxy.tritonia.fi/ft_gateway.cfm?id=2656426&ftid=1500282&dwn=1&CFID=602254891&CFTOKEN=24404840. DOI: 10.1145/2647648.2656426.

Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., & Memon, A. M. (2012). Using GUI ripping for automated testing of android applications. *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* [online] [28.1.2016], 258–261. 3–7 Sept. 2012, Essen, Germany. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=6494930. DOI: 10.1145/2351676.2351717.

Anbunathan, R., & Basu, Anirban (2014a). An event based test automation framework for android mobiles. *2014 International Conference on Contemporary Computing and Informatics (IC3I)* [online] [10.2.2016], 76–79. 27–29 Nov. 2014, Mysore, India. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=7019585. DOI: 10.1109/IC3I.2014.7019585.

Anbunathan, R., & Basu, Anirban (2014b). A recursive crawler algorithm to detect crash in android application. *2014 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)* [online] [11.2.2016], 1–4. 18–20 Dec. 2014, Coimbatore, India. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=7238518. DOI: 10.1109/ICCIC.2014.7238518.

Bae, Gigon, Rothermel, G., & Bae, D. (2014). Comparing model-based and dynamic event-extraction based GUI testing techniques: An empirical study. *Journal of Systems and Software* [online] 97 [28.10.2016], 15–46. Philadelphia, PA, USA: Elsevier. Available: http://www.sciencedirect.com.proxy.tritonia.fi/science/article/pii/S0164121214001472. DOI: 10.1016/J.JSS.2014.06.039.

Bai, Xiaoying, Tsai, W., Paul, Ray, Shen, Techeng, & Li, Bing (2001). Distributed end-to-end testing management. *2001. EDOC'01. Proceedings. Fifth IEEE International Enterprise Distributed Object Computing Conference* [online] [25.11.2015], 140–151. 4–7 Sept. 2001, Seattle, WA, USA. New Jersey, USA:

IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=950430. DOI: 10.1109/EDOC.2001.950430.

Beizer, Boris. (1995). *Black-box testing: Techniques for functional testing of software and systems*. New York, NY, USA: Wiley. 320 p. ISBN: 0-471-12094-4.

Bevan, Nigel (1999). Quality in use: Meeting user needs for quality. *Journal of Systems and Software* [online] 49:1 [11.3.2016], 89–96. 15 Dec. 1999. Philadelphia, PA, USA: Elsevier. Available: http://www.sciencedirect.com.proxy.tritonia.fi/science/article/pii/S0164121299000709. DOI: 10.1016/S0164-1212(99)00070-9.

Börjesson, Emil, & Feldt, Robert (2012). Automated system testing using visual GUI testing tools: A comparative study in industry. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)* [online] [25.2.2016], 350–359. 17–21 April 2012, Montral, QC, Canada. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=6200127. DOI: 10.1109/ICST.2012.115.

Cervantes, Alex. (2009). Exploring the use of a test automation framework. *2009 IEEE Aerospace Conference* [online] [13.1.2016], 1–9. 7–14 March 2009, Big Sky, MT. USA. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=4839695. DOI: 10.1109/AERO.2009.4839695.

Chang, Tsung-Hsiang, Yeh, Tom, & Miller, Robert C. (2010). GUI testing using computer vision. *Proceedings of the SIGCHI Conference on Human Factors in Computing System* [online] [28.1.2016], 1535–1544. 10–15 April 2010, Atlanta, GA, USA. New York, NY, USA: ACM. Available: http://dl.acm.org.proxy.tritonia.fi/ft_gateway.cfm?id=1753555&ftid=770032&dwn=1&CFID=602254891&CFTOKEN=24404840. DOI: 10.1145/1753326.1753555.

Daniel, Brett, Luo, Qingzhou, Mirzaaghaei, Mehdi, Dig, Danny, Marinov, Darko, & Pezzè, MMauro (2011). Automated GUI refactoring and test script repair. *Proceedings of the First International Workshop on End-to-End Test Script Engineering* [online] [17.2.2016], 38–41. 17 July 2011, Toronto, ON, Canada. New York, NY, USA: ACM. Available: http://dl.acm.org.proxy.tritonia.

fi/ft_gateway.cfm?id=2002937&ftid=995991&dwn=1&CFID=602254891&CFTOKEN=24404840. DOI: 10.1145/2002931.2002937.

Ericson Mobility Report (2016). [Online] [14.10.2015]. February 2016. Available: http://www.ericsson.com/res/docs/2016/mobility-report/ericsson-mobility-report-feb-2016-interim.pdf.

Gao, Zebao, Chen, Zhenyu, Zou, Yunxiao, & Memon, Atif M. (2015). Sitar: GUI test script repair. *IEEE Transactions on Software Engineering* [online] 42:2 [17.2.2016], 170–186. 1 Feb 2016. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=7214294. DOI: 10.1109/TSE.2015.2454510.

Hamill, Paul (2004). *Unit test frameworks: Tools for high-quality software development*. Sebastopol, CA, USA: O'Reilly Media, Inc. 216 p. ISBN: 0-596-00689-6.

Hassan, Mohammad M., Afzal, Wasif, Blom, Martin, Lindstrom, Birgitta, Andler, Sten F., & Eldh, Sigrid (2015). Testability and software robustness: A systematic literature review. *2015 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* [online] [10.3.2016], 341–348. 26–28 Aug. 2015, Funchal, Madeira, Portugal. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=7302472. DOI: 10.1109/SEAA.2015.47.

Holopainen, Juha (2004). *Regressiotestauksen tehostaminen.* [Online] [24.11.2015]. 2 Sept. 2004, University of Kuopio. Statement. Available: http://www.cs.uef.fi/tutkimus/Teho/RegressioselvitysJuha04.pdf.

International Software Testing Qualifications Board (2011). *Certified Tester: Foundation Level Syllabus.* [Online] [4.10.2015]. Available: http://www.istqb.org/downloads/send/2-foundation-level-documents/3-foundation-level-syllabus-2011.html.

Kuncheva, Ludmila I. (2014). *Combining pattern classifiers: Methods and algorithms.* 2nd edition. Somerset, NJ, USA: Wiley. 384 p. ISBN: 1118315235.

Last, Mark, Kandel, Abraham, & Bunke, Horst (2004). *Artificial intelligence methods in software testing.* Singapore: World Scientific. 208 p. ISBN: 9789812388544.

Liu, Chien-Hung, Lu, Chien-Yu, Cheng, Shan-Jen, Chang, Koan-Yuh, Hsiao, Yung-Chia, & Chu, Weng-Ming (2014). Capture-replay testing for android applications. *2014 International Symposium on Computer, Consumer and Control (IS3C)* [online] [9.2.2016], 1129–1132. 10–12 June 2014, Taichung, Taiwan. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=6846085. DOI: 10.1109/IS3C.2014.293.

Luostarinen, Riku, Manner, Jukka, Määttä, Juho, & Järvinen, Risto (2010). User-centered design of graphical user interfaces. *2010-Milcom 2010 Military Communications Conference* [online] [10.3.2016], 50–55. 31 Oct. 2010 – 3 Nov. 2010, San Jose, CA, USA. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=5680400. DOI: 10.1109/MILCOM.2010.5680400.

Mariani, Leonardo, Pezzè, Mauro, Riganelli, Oliviero, & Santoro, Mauro (2011). AutoBlackTest: A tool for automatic black-box testing. *2011 33rd International Conference on Software Engineering (ICSE)* [online] [10.2.2016], 1013–1015. 21–28 May 2011, Honolulu, Hawaii. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=6032578. DOI: 10.1145/1985793.1985979.

McMaster, Scott, & Memon, Atif M. (2009). An extensible heuristic-based framework for GUI test case maintenance. *2009. ICSTW '09. International Conference on Software Testing, Verification and Validation Workshops* [online] [26.1.2016], 251–254. 1–4 April 2009, Denver, CO, USA. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=4976393. DOI: 10.1109/ICSTW.2009.11.

Memon, Atif, Banerjee, Ishan, & Nagarajan, Adithya (2003). GUI ripping: Reverse engineering of graphical user interfaces for testing. *WCRE 2003. Proceedings of the 10th Working Conference on Reverse Engineering* [online] [29.1.2016], 260–269. 13–16 Nov. 2003. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.

org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=1287256. DOI: 10.1109/WCRE.2003.1287256.

Memon, Atif, Banerjee, Ishan, Nguyen, Bao N., & Robbins, Bryan (2013). The first decade of gui ripping: Extensions, applications, and broader impacts. *2013 20th Working Conference on Reverse Engineering (WCRE)* [online] [15.2.2016], 11–20. 14–17 Oct. 2013, Koblenz, Germany. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=6671275. DOI: 10.1109/WCRE.2013.6671275.

Memon, Atif. M. (2007). An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability* [online] 17:3 [21.1.2016], 137–157. 2 Jan. 2007. Hoboken, NJ, USA: John Wiley & Sons, Inc. Available: http://www.cs.umd.edu/~atif/pubs/MemonSTVR2007.pdf. DOI: 10.1002/STVR.364.

Molyneaux, Ian (2014). *The Art of Application Performance Testing: From Strategy to Tools*. 2nd edition. Sebastopol, CA: O'Reilly Media, Inc. 278 p. ISBN: 1491900547.

Mulder, Donovan L., & Whyte, Grafton (2013). A theoretical review of the impact of test automation on test effectiveness. *Proceedings of the 4th International Conference on Information Systems Management and Evaluation ICIME* 2013 [online] [13.1.2016], 168–179. 13 May 2013. Sonning Common, England: Academic Conferences and Publishing International Limited. Available: http://search.proquest.com/openview/5b5e6396d78060e553ff21e1abbf9308/1?pq-origsite=gschola r&cbl=396498. ISBN: 1909507180.

Myers, Glenford J., Corey Sandler & Tom Badgett (2011). *Art of Software Testing.* 3rd edition. Hoboken, NJ, USA: Wiley. 240 p. ISBN: 978-1-1180-3196-4.

Nagle, Carl (2000). Test automation frameworks. [Online] [26.1.2016]. Available: http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm.

Patton, Ron (2006). *Software Testing*. 2. edition. Indianapolis, IN: Sams. 389 p. ISBN: 0-672-32798-8.

Paul, Ray. (2001). End-to-end integration testing. *Proceedings of Second Asia-Pacific Conference on Quality Software* [online] [25.11.2015] 211–220. 10–11 Dec 2001, Honk Kong, China. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=990022. DOI: 10.1109/APAQS.2001.990022.

Potter, Bruce & Gary McGraw (2004). Software Security Testing. *IEEE Security & Privacy* [online] 2:5 [8.10.2015], 81–85. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=1341418. DOI: 10.1109/MSP.2004.84.

Rauf, Abdul M., & Alanazi, Mohammad N. (2014). Using artificial intelligence to automatically test GUI. *2014 9th International Conference on Computer Science & Education (ICCSE)* [online] [22.1.2016], 3–5. 22–24 Aug. 2014, Vancouver, BC, Canada. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=6926420. DOI: 10.1109/ICCSE.2014.6926420.

Rauf, Abdul E., & Reddy, Madhusudhana E. (2015). Software test automation: An algorithm for solving system management automation problems. *Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014* [online] 46 [25.11.2015], 949–956. 3–5 Dec. 2014, Kochi, India. Philadelphia, PA, USA: Elsevier. Available: http://www.sciencedirect.com.proxy.tritonia.fi/science/article/pii/S1877050915000058. DOI: 10.1016/J.PROCS.2015.01.004.

Raut, Pallavi, & Tomar, Satyaveer (2014). Android mobile automation framework. *International Journal of Multidisciplinary Approach and Studies, IJECS* [online] 1:6 [10.3.2016] 1–12. Nov–Dec 2014. Available: http://ijmas.com/upcomingissue/1.06.2014.pdf. ISSN: 2348 – 537X.

Ruiz, Alex & Yvonne W. Price (2008). GUI testing made easy. *Testing: Academic & Industrial Conference - Practice and Research Techniques, 2008. TAIC PART'08*

[online] [26.11.2015], 99–103. 29–31 Aug. 2008, Windsor, Canada. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=4670309. DOI: 10.1109/TAIC-PART.2008.11.

Szeliski, Richard (2010). *Computer vision: Algorithms and applications*. London: Springer Science & Business Media. 812 p. ISBN: 1848829353.

Shapiro, Linda G., & Stockman, George C. (2001). *Computer vision*. Upper Saddle River (N.J.): Prentice Hall. 580 p. ISBN: 0-13-030796-3.

Tamres, Louise (2002). *Introducing Software Testing.* London: Addison-Wesley. 281 p. ISBN: 978-0-201-71974-1.

Villanes, Isabel K., Bezerra Costa, Erick A., & Dias-Neto, Arilo C. (2015). Automated mobile testing as a service (AM-TaaS). *2015 IEEE World Congress on Services (SERVICES)* [online] [11.2.2016], 79–86. 27 June 2015 – 2 July 2015, New York City, NY, USA. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=7196507. DOI: 10.1109/SERVICES.2015.20.

Wang, Peng, Liang, Bin, You, Wei, Li, Jinghze, & Shi, Wenchang (2014). Automatic android GUI traversal with high coverage. *2014 Fourth International Conference on Communication Systems and Network Technologies (CSNT)* [online] [11.2.2016], 1161–1166. 7–9 April 2014, Bhopal, India. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=6821582. DOI: 10.1109/CSNT.2014.236.

Watkins, John (2001). *Testing IT: An Off-the-Shelf Software Testing Handbook.* Cambridge, UK: Cambridge University Press. 315 p. ISBN: 052179546X.

Wu, Yumei, & Liu, Zhifang. (2012). A model based testing approach for mobile device. *2012 International Conference on Industrial Control and Electronics Engineering (ICICEE)* [online] [9.2.2016], 1885–1888. 23–25 Aug. 2012, Xi'an, China. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=6322791. DOI: 10.1109/ICICEE.2012.500.

Yang, Xuebing (2011). *Graphic user interface modelling and testing automation.* [Online] [27.10.2015]. May 2011. Melbourne, Australia: Victoria University. PhD diss. Available: http://vuir.vu.edu.au/16066/1/Xuebing_Yang_PhD_thesis,_May_2011.pdf

Yuan, Xun, & Memon, Atif M. (2010). Generating event sequence-based test cases using GUI runtime state feedback. *IEEE Transactions on Software Engineering* [online] 36:1 [24.2.2016], 81–95. Jan.–Feb. 2010. New Jersey, USA: IEEE. Available: http://ieeexplore.ieee.org.proxy.tritonia.fi/stamp/stamp.jsp?tp=&arnumber=5306073. DOI: 10.1109/TSE.2009.68.