ANDROID GUI TESTING
A comparative study of open source Android GUI testing frameworks

Bachelor Degree Project in Informatics
G2E, 22.5 credits, ECTS
Spring term 2015

Linus Esbjörnsson

Supervisor: Gunnar Mathiason
Examiner: Joe Steinhauer
Abstract

Android is one of the most popular mobile operating systems on the market today with a vast majority of the market share when it comes to mobile devices. Graphical user interfaces (GUI) are often seen on the applications on these devices. Testing of these GUIs is important since they often make up for half of the source code of the application, and they are used to interact with the application. Automating these tests is very useful since it saves a lot of time, but can be difficult. The tools that are available for automating the tests are often not suitable for the developers' needs, because of the lack of functionality. Therefore, the need for a characterization of the frameworks is needed, so that developers more easily can fit a framework for their needs.

In this study, four open source frameworks for Android GUI testing have been selected for evaluation: Robotium, Selendroid, UI Automator and Espresso. Criteria used in the evaluation have been identified with the help of a literature analysis. The results show that two of the frameworks, Robotium and Espresso, lack the ability to fully test activities, which is the main component of Android application GUls. Furthermore, the study resulted in characterizations of the frameworks.

Keywords: Android, GUI testing, automated testing, evaluation of frameworks, Robotium, Selendroid, Espresso, UI Automator
Table of Contents

1 Introduction ........................................................................................................................................... 1

2 Background ....................................................................................................................................... 3

2.1 Android .......................................................................................................................................... 3

2.1.1 Android Software Development Kit (SDK) ................................................................................. 4

2.1.2 Android software stack ............................................................................................................... 5

2.1.3 Activities .................................................................................................................................... 6

2.2 Software testing ............................................................................................................................... 7

2.2.1 White-box testing ....................................................................................................................... 8

2.2.2 Black-box testing ....................................................................................................................... 9

2.2.3 Automated testing ..................................................................................................................... 9

2.2.4 Test oracles ................................................................................................................................ 10

2.2.5 Android GUI testing ................................................................................................................. 10

3 Problem ............................................................................................................................................. 12

3.1 Objectives .................................................................................................................................... 12

4 Method ............................................................................................................................................... 14

5 Related work ..................................................................................................................................... 16

6 Evaluation criteria .............................................................................................................................. 18

6.1 General criteria .............................................................................................................................. 18

6.2 Technical criteria ............................................................................................................................ 20

7 Frameworks ....................................................................................................................................... 22

7.1 Robotium ........................................................................................................................................ 22

7.2 Selendroid ....................................................................................................................................... 25

7.3 UI Automator ................................................................................................................................ 28

7.4 Espresso ........................................................................................................................................ 32

8 Result ................................................................................................................................................ 36

9 Discussion ......................................................................................................................................... 39

10 Conclusions .................................................................................................................................... 41

10.1 Future work ................................................................................................................................... 41
1 Introduction

Android is one of today’s largest operating system for smartphones. It is also the only well-known operating system to have gained market share during the last year according to IDC (2014), beating both Apple’s iOS and Microsoft’s Windows Phone. Applications on Android are easily downloaded and managed through a digital distribution platform called Google Play, which is a marketplace for applications like for example App Store for iOS. According to Statista (2014), Android is the leading platform for the amount of applications available on their digital distribution platform, with over 1.3 million applications.

Since the growth of mobile devices, new areas of testing have appeared with the addition of mobile operating systems, where user events are more advanced than on a PC. For example, a user may touch the screen of the device on two spots simultaneously. Software testing makes up for a big part of the software development process, and every component needs to be tested thoroughly to assure that it is working as intended. It is also important to test the application through interaction via the graphical interface of the application to be able to test the whole application the way the user sees it. Ammann and Offutt (2008) state that GUI (graphical user interface) oriented code makes up for half of the source code in software, and it’s very important to be able to test it since the GUI is the most common way for a user to interact with the software. If not tested properly, the user may be exposed to bugs that ruin the user experience in the form of example, application crashes.

According to Wang et al. (2014) there are several limitations to manual testing, two being human labor and that it’s a very time consuming activity. Therefore, there is a need for automation in GUI testing, and there are numerous approaches to solving this. Ammann and Offutt (2008) mention a number of approaches to GUI testing, two of them being capture/replay tools and scripts that stimulate the GUI through interaction in scripts. Capture/replay tools are tools where a series of events can be captured within the GUI, and then replayed in the same order to simulate user interaction with the GUI.

In this work, frameworks for Android GUI testing have been evaluated according to a number of evaluation criteria. There criteria have been carefully developed and their relevance have been analyzed to find the most important aspects of Android GUI testing framework, and what they should be able to handle. Characterizations of the frameworks have been set up, and then compared to each other according to the evaluation criteria to obtain an answer about how preferable each framework is in a specific area that the evaluation criterion covers.

In this study, the frameworks that are evaluated are, Robotium, Selendroid, UI Automator and Espresso. The results showed that it is not trivial when choosing a framework to work with, no framework proved itself to be superior when looking at the evaluation criteria. Therefore, the study resulted in a characterization of the frameworks and in what situations they are suitable in.

The report is divided into ten main chapters where chapter one, the introduction, introduces the reader to the problem area, and presents the method used to gather data, and what the results
were. Chapter two is the background chapter, where the reader is given information on Android and the problems that come with Android development, software testing, automated testing, and information on Android GUI testing.

The third chapter addresses the problem of this study, it explains for the reader why the problem is relevant, and defines a question that this study will answer with a number of well-defined objectives.

Chapter four is the method chapter; here the objectives defined in chapter three are translated into methods to solve these objectives. In this study, literature analysis is the main method to collect data.

The following chapter is the related work chapter. Here, the reader is more familiarized with related work within the area, and how they are related to this study.

Chapter six and seven address the evaluation criteria gathered from the literature, as well as the frameworks and how they perform in each criterion.

The next chapter summarizes the findings of the frameworks in two categorized tables, and explains the results.

Chapter nine is the discussion part, here the work is critically analyzed, and the reliability of the work is discussed and how the work was carried out.

Lastly, chapter ten summarizes the work and presents ideas for future work.
2 Background

2.1 Android
Android is one of the most popular operating systems for mobile devices as of 2014 (IDC, 2014). It is based on the Linux-kernel, and is available as open-source software where everyone can download the source code and compile their own version. This enables the community to contribute with solutions to bugs that developers otherwise might not find, and is an important feature with the operating system (Google Inc., 2015a).

Android Inc. was founded by Andy Rubin, Rich Miner, Nick Sears and Chris White in 2003, but was later acquired by Google Inc. The development was then headed towards mobile devices, and in 2007 the Open Handset Alliance was founded, with companies such as Google, Sony and device manufacturers such as HTC and announced open standard development for mobile devices. The Android operating system was then unveiled as their first product (Open Handset Alliance, 2007). In 2008 the first smartphone was released using Android as the running operating system (Moor, 2008).

There are a number of different versions of the Android operating systems, with each version having a codename that represents a piece of candy. Figure 1 shows the distribution of each version as of January 2015, where Jelly Bean is sitting as the most used Android version. By the time this paper was written, a new major Android version has been released called Lollipop (Beavis & McCann, 2015). This version is however, not considered in this report.
2.1.1 Android Software Development Kit (SDK)

To develop applications for Android, the developer needs access to the Android software development kit, which is basically a set of tools that are needed to develop Android applications. Android applications are packaged into *APK* files, which contain all the assets and compiled source code that is needed to install the application on a device (Google Inc., 2015b).

Tools for testing are included in the Android SDK, and provide the developer with support for writing automated tests using the *UI Automator* framework, and *Monkeyrunner* to generate pseudo-random events from outside the Android device, to emulate a user and to stress-test the application (Google Inc., 2015d).

Android applications are mainly written with the Java language, but also support languages like C and C++ that allow Android to run *native code* (code that is compiled directly to run on a specific processor). To be able to develop Android applications with native languages, the Native Development Kit is needed, and provides the developer with full native language support. Usually testing frameworks are integrated into the application, and scripts are therefore written in Java. There are frameworks, however, that let you test the application from outside the application using different scripting languages.
2.1.2 Android software stack

The software architecture behind Android is often called the Android software stack, and is composed by a number of layers (refer to Figure 2). CompileTimeError (2013) describes the following layers:

- **Linux Kernel.** At the bottom of the stack is the *Linux Kernel* responsible for hardware interaction such as accessing the graphics card, mouse and keyboard input and output.

- **Libraries.** For Android to be able to operate properly with its core features, the *Libraries* layer exists to support functionality such as 3D rendering using SGL (Scene Graph Library) and connecting to databases using SQLite. Android versions previous from Lollipop were equipped with the Dalvik Virtual Machine (DVM), which is used to interpret the bytecode that is generated from compiling Java-code. When Lollipop released, DVM was replaced with ART (Android Runtime) as runtime environment (Google Inc., 2015e).

- **Application Framework.** *API (Application programming interface)* calls made by applications are handled by the *Application Framework* layer. Some examples of API calls are accessing the phones contact list, switching activity or getting the current location of the device.

- **Applications.** The top layer is the *Applications* layer, which handles all applications that are installed on the device. These from applications made by the Android team to third party applications installed from example, Google Play, which is the main digital distribution platform of Android.
2.1.3 Activities

*Activities* in Android applications are components that are used to display something for the user to interact with. Android applications are usually composed by multiple loosely coupled activities that are bound to each other, and the activities can switch between each other to display different areas of the application, or perform different actions, like for example, database accesses (Google Inc., 2015f).

Activities can invoke other activities using *intents*, which are used as glue between the activities in the application. Intents are basically messages between the different components that are used to perform something (an intention). The most common use for intents are starting a new activity, and to be able for the user to send extra data to the newly started activity, a *bundle* can be used, which acts as an intermediate storage when transferring data between activities (Vogel, 2014).

There are a number of different *states* an activity can have, and the developer can override a corresponding method for each state in the source code. For example, if an activity just has launched, the `onCreate()` method is invoked (Google Inc. 2015f). This is called the activity lifecycle and can be one source of errors in Android applications. The reason for that is because
a service could be initialized in the `onCreate()` method, which then is finalized in the `onStop()` method. Since it is possible for the application to go from `onStop()` to `onRestart()`, and then reach the execution of the application once again, an error can occur. All types of states available for activities can be seen in figure 3.

![Activity Lifecycle Diagram](image)

Figure 3. The activity lifecycle. Licensed under the Creative Commons Attribution 2.5 Generic license (Creative Commons, 2015 and Google Inc. 2015f).

### 2.2 Software testing

Software testing is becoming a bigger part in software development, and the need for correctness and high quality products has increased. To ensure this, all software products must be tested thoroughly and the test engineer is becoming increasingly important. Ammann and Offutt (2008) states that a company needs to involve the tester early in the software development process, to ensure that the software maintains high quality.

It is important to automate testing as much as possible, since manual testing requires human labor and is very time consuming (Wang et al., 2014). Dustin et al. (2009) states in their book that the software tests should not only be used in the beginning of the software development,
but also during updating of the software components. This is because even a small change in a component can introduce new errors.

Lindström (2009) acknowledges the need for better testing tools, and that they need to support observability of the test execution, so that erroneous behavior can be detected when tests are run. Ammann and Offutt (2008) mention that testing is basically to “find a graph, and cover it”. Graph coverage is a fundamental concept when the tester has access to the source code (mentioned in chapter 2.2.1), but is different when it comes to graphical user interface (GUI) testing, since the tester does not know how to access certain parts of the code.

2.2.1 White-box testing
White-box testing are techniques used for testing software with knowledge of the internal structure (Mohan et al. 2010). Usually the tester uses this knowledge to construct test cases that will cover all the reachable code, this is called code coverage.

As mentioned in chapter 2.2, graph coverage is a useful concept for making sure that the coverage criteria are met. Since white-box testing allows the tester to see the internals, analyzing the code can result in testing requirements that need to be fulfilled by the test cases (Ammann & Offutt, 2008).

There are a number of different testing criteria with graph coverage, for example node-coverage and edge-coverage (Ammann & Offutt 2008). These testing criteria will not be discussed in detail in this thesis, but a simple example is illustrated with the code example below:

```java
if (age >= 18)
{
    adult = true;
}
else
{
    adult = false;
}
return;
```

The code can be translated into a graph, as illustrated in Figure 3, for which then a coverage criterion can be applied to. In this example, node coverage will be used to demonstrate how the graph can result in test requirements that the test cases must fulfill. Node coverage implies that every node in the graph needs to be covered at least once, and in the example scenario, node coverage would give two test requirements, \{(A, B, D), (A, C, D}\).

These requirements show the path that the code execution needs to take during execution, and for the code to traverse into the if-statement, the variable “age” needs to be set to 18 or above. To enter the else-statement, “age” needs to be set below 18.
2.2.2 Black-box testing
When the tester does not know anything about the internal architecture of the software, it is possible to conduct black-box testing. Mohan et al. (2010) describes black-box testing as the interaction with an interface by providing inputs without knowing how the internal system architecture looks like.

There are a number of different types of testing when it comes to black-box testing. Williams (2006) lists regression testing as one of six types of testing. Regression testing is conducted when you have a test suite that needs to be re-run once changes in the software happen, since a change in the code might change the output of the program. Only this type of testing is mentioned because regression tests are run throughout the whole testing cycle, and is important when testing any type of software.

In the case of Android GUI testing, black-box tests can be constructed with the help of frameworks built to emulate input to an Android device, and therefore test the application as it’s supposed to be used.

2.2.3 Automated testing
Ammann and Offutt (2008) emphasize that testing should be automated as much as possible. Though, there are some challenges when it comes to automating the testing process.

Rice (2003) lists ten of the major challenges in automated testing, some of them being problems with managing the test suite configurations, difficulty of using the tool, lack of tool support and in the end, investing in the wrong tool. Also, the need for automated testing frameworks is great, since testing plays a big role in software development (Khan & Khan, 2014).

Automated tests are usually run in scripts, below is an example of such a script made from the Robotium framework (refer to chapter 7.1):

![Figure 3. Example of a graph.](image-url)
solo.assertCurrentActivity("Activity error", MainActivity.class);
solo.clickOnButton(solo.getString(R.string.testButton));
solo.assertCurrentActivity("Wrong activity", MainActivity.class);
solo.clickOnButton("Example");
assertTrue(solo.waitForText("Hello World"));
solo.goBack();

The script uses assertion-functionality, which means that if a certain condition is not fulfilled, the execution will cancel and print the error that occurred during the statement. Examples of such assertions are found in the code example above, where for example in the first row, the script will check if the current activity is equal to the MainActivity class when started, which is usually the name of the entry point to applications.

2.2.4 Test oracles
Barr et al. (2014) mention the need for automated testing tools to be able to distinguish correct output from incorrect output; this is called the oracle problem. Ammann and Offutt (2008) describe the problem with automated testing to know whether a program has executed correctly given a certain input. The challenge is how to interpret the programs output (if there are any), or behavior, and analyze if this behavior is correct.

There are a number of approaches to how to solve this, and one of the most common solutions is direct verification. This approach is almost what it sounds like, directly verifying the output to the expected output, which is how it's described by Ammann and Offutt (2008). Direct verification is only a proper solution if the application comes with a specification to what output is correct, then it's easy to verify if the output is incorrect. The other approaches mentioned will not be covered any further because direct verification is the only approach interesting for GUI testing. This is because GUI testing often uses black-box testing, and therefore, direct verification is the only alternative.

2.2.5 Android GUI testing
According to Cuixiong and Neamtiu (2011) there are a number of issues regarding the activity and event-based structure of Android applications. Since the Android operating system runs on heterogeneous devices, where different manufacturers use different technologies, there are a number of bugs derived from the heterogeneous aspect (Amalfitano et al. 2011).

Google Inc. (2015g) states that Android applications can have multiple entry points since the activities act as independent modules that are connected. This means that testing is more difficult since the connection between every activity needs to be tested as well. It is also mentioned that Android applications can be customized for different types of devices by providing different layouts and functionality, depending on the device running the application.
Amalfitano et al. (2013) mention a couple of important areas with Android testing. The first one is to test the activity lifecycle. This includes testing the activities response to user, system and its own lifecycle events. Service testing is also mentioned as an important part to test. The service lifecycle event (as with activities) is important to include in the test cases. Content provider is also an important part of Android GUI testing, and is also mentioned, as well as broadcaster receiver testing. Content provider testing is testing of some sort of shared resource. An example of such a shared resource is a database. Broadcast receiver testing is testing on a component that is listening for a message from an intent.

When performing Android GUI testing, direct verification is easily performed since the application will most likely crash or show an unexpected screen if an error occurs (Memon et al. 1999). The script example in chapter 2.2.3 uses assertions to make sure the output from the performed action matches the expected output. For example, a button may make the application switch screens.

Memon et al. (1999) describes the amount of permutations of available GUI actions to be great, and that each action may change the state of the program, and all of the actions may need to be tested. They mention the downside of using human labor when creating test scripts, which is also mentioned in a study by Wang et al. (2014), since it is time consuming work. Kropp and Morales (2010) also mention that it is almost impossible to test all the states a graphical user interface can have. This increases the need for capture/replay tools since they help the testers with creating repeatable sequences of events that can be automatically run, and help test common sequences of events that users may perform.
3 Problem
The purpose of this study is to answer the following question: "Is there an open source Android GUI testing framework that is more preferable to other open source frameworks with regard to the chosen evaluation criteria?".

Open source software has been shown to be more innovative, and take full advantage of sharing knowledge (Von Hippel, 2001). Also, open source software is usually available without a purchase fee, which makes them preferable for developers and organizations since it saves resources. Therefore, only open source frameworks are considered in this study.

When choosing a framework to work with when testing Android applications, there are several different properties of the framework that developers and organizations need to consider to find the proper tool for their needs. As mentioned in chapter 2.2.3, one of the biggest challenges with software testing is investing in the right tool that fits the requirements of the testing. Persson and Yilmaztürk (2004) mention common pitfalls when it comes to automated testing tools, one being the lack of functionality for the developers’ needs when acquiring the automated testing tools. Therefore it is important to properly evaluate the frameworks before making the decision to integrate them into the development. Since there are Android-specific problems with the GUI (as mentioned with activities in chapter 2.2.5, and more in chapter 6.2), it is important to analyze how the frameworks support testing of activities.

The aim is to find out what frameworks are most suitable for a number of different evaluation criteria and to find out what criteria that are significant when comparing Android GUI testing frameworks (refer to chapter 6 for examples of such criteria). With a better understanding how the frameworks perform, developers for Android applications can more easily find a fitting framework for their type of needs by looking at the criteria that the frameworks were evaluated with.

Since the frameworks chosen for evaluation are open source, it is also possible to browse the source code because it’s typically available for open-source frameworks and then check how the frameworks solve a certain problem. The implementation of the frameworks can for example help researchers within the same area continue the development of Android GUI testing.

3.1 Objectives
To answer the question of this thesis, a number of objectives need to be fulfilled. These objectives are the following:

1. Identify and present evaluation criteria for Android GUI testing frameworks.
2. Identify and present valid open-source Android GUI testing frameworks.
   a. Select and motivate testing frameworks that are the most commonly used in projects/organizations and/or where a characterization is the most useful.
b. Evaluate selected frameworks, using the selected evaluation criteria.

3. Compare the frameworks to each other with regards to the evaluation criteria, such that the framework comparison shows the characteristics, advantages and disadvantages for different types of users.
4 Method
This chapter describes the chosen method to be able to complete the objectives that were presented in the previous chapter. There are also some reflections over alternative methods to solve each objective.

To identify the available frameworks to use in the study and to identify evaluation criteria (objectives 1 and 2), a literature analysis is the chosen method. Berndtsson et al. (2008) describe literature analysis as an analysis of published sources with a systematic approach. They also mention the risks and difficulties with using the method, being the usage of irrelevant sources and completeness. Completeness being the knowledge of whether enough information has been collected, and when to stop searching. For handling completeness, the literature analysis will proceed until the same information is being repeated in the literature. That is a sign for when to stop searching, since it can be concluded that enough information has been gathered when there are no new findings when looking in new literature.

To handle the risk with irrelevant sources, acknowledged sources are used as first hand option with the help of the scientific databases IEEE and ACM and the scientific search engine Google Scholar to find frameworks and important criteria for Android GUI testing that scientific studies mention. Sources that are not peer-reviewed or published in scientific journals need to be carefully considered, since documentation and general information about the frameworks often reside on general websites. To ensure that these sources provide the correct information, additional sources will be searched to ensure that the information is reliable. These sources will also be used to find frameworks to study, including Google as search engine support and other studies that are published in articles on different websites.

The frameworks that are considered in this study are those who only focus on the Android operating system, this is to scale down the study to a reasonable size and analyze if these frameworks have support for important criteria in Android GUI testing.

When evaluating the frameworks, the documentation of the framework will be used as a primary source of information when evaluating criteria. To make sure that the information gathered from the documentation is trustworthy, and with frameworks that lack good documentation, search engines (as mentioned previously) will be used to find similar information to what’s said with the documentation. If there is not enough information to evaluate a certain criteria, the framework will be analyzed by the author through investigation (with more general criteria), or by browsing the source code (with more technical criteria).

Berndtsson et al. (2008) state a way to handle the issues with performing a literature analysis by mentioning the importance of self-reflection during the process, and have a good understanding of the phenomena analyzed. This means that there is a risk of making different interpretation of the information found in the literature. For this study, this is handled because of the combination
of documentation and other sources to avoid making wrong interpretations of the information found.

The benefit of using this approach is the amount of data that can be collected through different kinds of sources with low resource cost and will allow for a wider perspective. The data will then be realistic, since the study is not relying on experiments, but real information from literature.

There is a similar study by Jones et al. (2006) that used an approach that is close to what this study is using. What they did was to gather enough information about relevant evaluation criteria for the frameworks, and then did a literature review of the frameworks to then compare them to each other with regards to the evaluation criteria.

Another approach to determining what frameworks and what criteria to use would be to perform a case study at a company that work with Android GUI testing where interviews would be conducted to obtain frameworks that are interesting to perform the study on, and criteria that are important for Android GUI testing. Börjesson and Feldt (2012) have used this type of approach to gathering frameworks, but is not chosen because of the lack of resources available and the same type of information is available through Internet browsing. In this case the frameworks are also motivated through the selection criteria that they should be open source, only focused on Android, and popular amongst developers, which scales down the frameworks to a reasonable number.

An alternative approach to evaluating the frameworks is by performing experiments with the frameworks. A prototype application would then be built so that scripts made from the API of the frameworks could be run and test the functionality of the frameworks. By using this approach, the support for a certain criterion can be validated, since it’s not theoretical. However, it is not certain that support exists outside of the experimental environment; real applications may have other difficulties that will not be found with only performing experiments.
5 Related work

Previous studies that are closely related to this study, are studies where comparison of automated testing frameworks are done. Other studies that are related include the area of Android GUI testing, automated testing in general and general GUI testing.

There are some previous studies that are related to this one. Kropp and Morales (2010) have written an article about automated GUI testing in Android. In their study, they've compared two framework approaches in Android GUI testing, the Android Instrumentation Framework, and the Positron Framework, and compared it with desktop testing techniques. In their comparison they implemented scripts that exemplify how the API works for the frameworks, and used a literature analysis to examine the features and internal structure of the frameworks.

The result of their study shows that the Android Instrumentation Framework and the Positron Framework are strong in handling UI resources, like the activity lifecycle; generating user events, and handling assertions to verify erroneous behavior. Other findings from the authors were that the frameworks have disadvantages, one being the need for knowledge of the source code (only white-box testing support) to identify the UI elements in the application, the frameworks are also limited to single activity testing, and cannot test spanning multiple activities and lack support for capture/replay and script generation capabilities.

When relating to this study, there are similarities since it focuses on Android GUI testing and the frameworks to support automated testing. It is also very similar due to the fact that the aim of the study is to characterize the frameworks with showing the advantages and disadvantages with using the frameworks.

The difference with this study is that Kropp and Morales (2010) do not have a specific set of criteria that they look at when characterizing the frameworks. They describe how the frameworks operate, and evaluate how the API works and demonstrate how to write scripts in both frameworks. It is more focused on the low-level technical details regarding the frameworks. This study is used as a guideline to find what areas of Android GUI testing they find interesting to look at when comparing Android GUI testing frameworks, it is also used to identify some of the criteria used in the evaluation in this study (refer to chapter 6).

Another work that is closely related to this study is one by Börjesson and Feldt (2012). Their article evaluates two tools for GUI testing of software used for image recognition. In their study they develop a number of criteria that are used to characterize and evaluate the tools, but the focus of their study differs from this one. The aim of their study is to evaluate tools used for GUI testing for visual recognition software in the industry, while this one aims at comparing advantages and disadvantages with Android GUI testing frameworks by characterizing them.

Their study resulted in a number of properties derived from the tools documentations, these properties were then used to compare the frameworks to each other, to find a favorable
framework in each of these properties. The comparison between the frameworks showed advantages and disadvantages with using them, and the authors proved this by conducting experiments with a number of scenarios where they implemented this with help from the tools evaluated.

What makes this study interesting is the fact that they construct criteria that are interesting for evaluating the frameworks, and then conduct experiments to evaluate certain features and characteristics, similar to what has been done in this study. The study is used as a template in how to conduct a proper comparison of GUI testing frameworks.
6 Evaluation criteria

Evaluating Android GUI testing frameworks is not trivial, this is because there are a number of aspects when it comes to testing where a framework can be preferable. For example, a framework that emphasises clean code and easy scripts might not be the better choice when it comes to testing the activity lifecycle. Therefore, a number of important criteria for Android GUI testing frameworks have been selected to bring a more overall view over the frameworks to characterize them.

The evaluation criteria have been divided into two categories, one being more general, where criteria that give an overview of the frameworks are concerned and one where important technical details regarding Android GUI testing frameworks are concerned.

6.1 General criteria

General criterion 1: API:
One factor when choosing a framework for a certain testing need is that the framework has an API that requires minimum amount of effort from the developer to write working scripts. Bajaj (2014) mentions some parameters when it comes to choosing the right framework, one of them being script creation time. Therefore, it is vital that the framework API is well designed so that the developer has to write minimum amount of code to reduce the script creation time. In this study, the complexity of the frameworks API will only be analyzed. Complexity, in this case, means the amount of operations required to perform basic functionality, such as clicking buttons for example. If the API is complex, developers have to write more code and it affects the script creation time.

General criterion 2: Logging support:
When running test cases with the help of a testing framework, there might be several steps that need to be taken for an error to appear. Therefore, it is important for the framework to be able to log each step in a manner that’s easy to analyze and interpret for the developers. With good logging functionality, developers can find the source of what’s causing the error and then recreate it with the knowledge of the steps that causes it.

General criterion 3: Capture/replay support:
As mentioned in chapter 2.2.5 by Memon et al. (1999), the amounts of permutations available are almost infinite. Capture/replay tools offer possibilities to create sequences of events that users often perform in the application to test the most common event patterns since all permutations are almost impossible to test. Kropp and Morales (2010) also state that tools for Android GUI testing had notable limitations, one being the absence of capture/replay support.

General criterion 4: Documentation:
In the paper by Bajaj (2014), learning time is a parameter that is evaluated in the framework comparison performed in the study. With the help of documentation of the framework, the
learning time can be reduced due to the possibility to follow instructions with the documentation, or get an easier understanding of the API.

**General criterion 5: Testing method:**
Testing method refers to the information needed by the application to be able to test it. When it comes to GUI-testing, black-box is a method that is often used; this is because only the GUI is needed to test since it doesn’t require access to the source code. However, there are frameworks that require access to the internal structure of the application to be able to test it, for example identifiers for UI elements. Therefore, this criterion is added for the evaluation.

**General criterion 6: Automatic event generation:**
Cuixiong and Neamtiu (2011) describe automatic event generation as a powerful technique for verifying GUI applications. Events that are not expected by the application can help detecting bugs that otherwise would not have been found. Therefore, automatic event generation is an effective way for the framework to help the developer find bugs.

**General criterion 7: Emulator/real device support:**
As mentioned by Amalfitano et al (2011), Android is used on many heterogeneous devices and use different technologies where different bugs may appear on each device. Therefore, it is important that the framework is able to support real devices when testing. On the other hand, while developing, regression testing is run often, and developers may not often have access to a real device during testing. The support for emulators is therefore very important as well, and that’s why this criterion was added.

**General criterion 8: Version support:**
Cuixiong and Neamtiu (2011) also mention API errors as a source of error in their study. API errors are caused through incompatibilities between the Android version that the application assumes the user to have, and the actual version that the user has installed. This can be for example the addition of new API methods or technologies that are assumed to be supported by previous versions of Android.

**General criterion 9: Version control support:**
Ammann and Offutt (2008) mention the importance of keeping the test artifacts (the versions of the test suites) managed because lack of management is likely to cause the testing to fail. For this, software support is needed. This criterion is added because it is important that the frameworks have support for test artifact management since it will ease the workload of the testers if they don’t have to do it manually or use a combination of software.

**General criterion 10: IDE support:**
When developers choose a framework for their testing, it’s important to be able to integrate the framework into the environment that’s used in their current work. This is because the time it will take to get used to the framework will lower when used in an IDE (Integrated development environment) that the developers are used to. Mostly Eclipse and Android Studio are used as development environment, therefore, these are only considered in this study.
Other relevant criteria for Android GUI testing frameworks can be found in a document written by AQuA (App Quality Alliance, 2013), which is an organization aimed at standardizing testing and quality of mobile applications. They are recognized by IEEE (Institute of Electrical and Electronics Engineers) and therefore the document is used as a checklist for ensuring high quality of mobile applications. Since the document is mainly a collection of use cases that help improving the quality of the application, it is of great interest to be able to automate them since executing them manually increases the risk of error.

Another set of criteria that are important to evaluate for the frameworks are the activity lifecycle support as mentioned in chapter 2.2.5. Figure 3 shows the activity lifecycle and all the methods associated with it. This figure can be viewed as a graph, where every method represents a node, and every edge is the event that triggers that method. To cover this graph, each edge must be reached at least once. The edges need to be translated into technical criteria that the frameworks must be able to perform to properly cover this graph.

Cuixiong and Neamtu (2011) found in their study that activity bugs are very common due to the inability of developers to properly implement them. Some of the activity lifecycle criteria are also directly tied to the use cases in the document by AQuA.

Technical criterion 1: Resume application:
To be able to test the onRestart() method in activity lifecycle, there has to be an option to be able to resume the application, since it's the first method that's called when a user navigates to the activity. Use case 9.2 in AQuA's list also defines resuming of the application as critical. Background tasks are often present in Android application due to the limitation of the Android UI thread, which acts as the main thread of all applications (Google Inc. 2015h).

Technical criterion 2: Suspend application:
For the onStop() method to be called from the application, the activity needs to no longer be visible, and this is done by suspending the activity. Use case 2.2 in AQuA’s list also mark suspending the application as a critical test case.

Technical criterion 3: Finish activity:
The onDestroy() method of the activity lifecycle is called when an activity is finished, this can be done by providing support in the API to instantly finish the activity, or to be able to emulate the “Back” button of an Android device.

Technical criterion 4: Change device settings:
According to the document by AQuA, there are a number of use cases that require changing of device settings (for example, 9.2 and 9.6). Changing of settings such as orientation will also help testing of the activity lifecycle. Therefore, this criterion is added as the framework may support changing of device settings during execution of test cases.
Technical criterion 5: Scrolling:
Some activities take advantage of the scrolling feature in Android when trying to present more data than the device screen can actually hold. Use case 13.1 in AQuA’s document also show that it’s critical to be able to test scrolling capabilities, and therefore it is interesting to analyze if the frameworks are able to automate it.

Technical criterion 6: Multi-touch/Simultaneous keypresses support:
Use case 13.4 mentions testing of simultaneous keypresses or multi-touch that might cause the application to enter an unusable state. If the framework can provide support to simulate multiple simultaneous key presses or touch presses, bugs that otherwise might not be found can be discovered.

Technical criterion 7: Re-launch application:
Support for launching of re-launching the applications provides both support for the onRestart() method as mentioned before, but can also test if the current state of the application will be reset. Use cases 2.2, 2.3 and 13.6 of AQuA’s document address this.

Technical criterion 8: Delay support:
Sometimes it can be useful to make the testing script sleep (delay) for a bit until further execution. This can be used to put delays in between button clicks, or wait until a certain string is displayed for example.

Technical criterion 9: Condition-based testing:
Condition-based testing in this case means different test execution paths depending on some factor. For example, there may be an if-statement that will click on a certain button if a certain piece of text is displayed, else it will click on another button. Frameworks need to support this, because it can also be applied to assertions (exit test execution with an error message if the assertion is false).

Technical criterion 10: Support for device keys:
Support for device keys can provide support for the whole activity lifecycle, as the home button, for example, suspends the current application, and the activity will then trigger its onStop() method. It’s also mentioned in AQuA’s document in test case 13.6, where all the keys need to be tested.
7 Frameworks

As mentioned before, the study focuses only on relevant frameworks, and in this case that means frameworks that are in question for developers and organizations when conducting Android GUI testing. The frameworks chosen for this study are Robotium, Selendroid, UI Automator and Espresso.

There are more frameworks that could have been chosen for evaluation, but the ones chosen in this study were more relevant for both developers and organizations since they are more powerful and well-used amongst developers. When searching for

As mentioned in chapter 3, the information of the frameworks are gathered from the documentation, with Google search engine as a way to ensure that the documentation is correct. The only framework that deviated from this approach due to its lack of good documentation was Selendroid. Selendroid was evaluated with the help of the documentation that was available, sources on the Internet (discussion forums) and by browsing the source code.
All the example scripts mentioned in the sections below are written by the author of this thesis, with the help of the frameworks API.

7.1 Robotium

Robotium is a widely discussed framework in both the scientific and developer world, and is considered one of the biggest Android GUI testing frameworks out there. It supports both hybrid and native applications. The motivation to why this is chosen is because of the wide user base, and the big community that contributes to making it as popular as it is. Version 5.3.1 is the one evaluated in this study.

API:
Robotium scripts are written in Java and the framework is installed through a JAR file that is added as an external dependency. The framework detects UI elements in the activities by providing an ID to the element if the developer has knowledge of the internals of the activity, or by providing a name that is displayed on the element. For example, buttons usually have a name associated with them.

The Robotium API is quite simple to use. It is based on the Android test framework and each class is derived from the ActivityInstrumentationTestCase2 class, and therefore has access to every method located in that class. Since scripts are written in Java, the developers do not need to learn a new scripting language to be able to start testing, which saves time in getting started with the testing.

Robotium’s main component is the Solo object, which is the main object of the Robotium API and contains methods for activities, menus, dialogs etc. Robotium is also able to test web elements within the applications with the help from the WebElement object.
A snippet of how the Robotium API works is demonstrated in the following script where the comments describe the intended action:

```java
// Try to click on the button named Button 1
solo.clickOnButton("Button 1");
// Wait until text appears
solo.waitForText("Button 1 clicked");
// Assert the text that should be found
assertTrue(solo.searchText("Button 1 clicked");
// Press the hardware back button
solo.goBack();
// Take screenshot of current screen
solo.takeScreenshot();
// Call finish() on all opened activities
solo.finishOpenedActivities();
// Finalize Robotium
solo.finalize();
```

**Logging support:**
Since Robotium is an external library that is connected to the test project, it can only take advantage of the logging features that the IDE offers. When it comes to generating a test report in HTML or XML format, Robotium tests are built on JUnit testing (JUnit is a unit testing framework), which allows the developer to export results in JUnit XML format if the IDE supports JUnit. To conclude, Robotium supports XML report generation in cooperation with the IDE running it, and normal logging is also handled by the IDE.

**Capture/replay support:**
When it comes to capture/replay support in Robotium it’s not supported by the framework. However, there is additional software that can be installed as a plugin for Eclipse and Android Studio that adds capture/replay support for Robotium called Robotium Recorder. This software is commercial and is not open-source, so it is not taken into consideration in this evaluation.

**Documentation:**
There is detailed description of every method available online on Google Code (see [http://robotium.googlecode.com/svn/doc/index.html](http://robotium.googlecode.com/svn/doc/index.html)). On the website of Robotium, they also have a wiki-page, where tutorials, guides and examples of scripts are located to help the developers easily get started (see [https://code.google.com/p/robotium/w/list](https://code.google.com/p/robotium/w/list)). Robotium is also a very widely discussed framework, which makes it very easy for developers to use various search engines to search for information regarding Robotium.

**Testing method:**
Robotium provides partial support for black-box testing. The reason for this is because for the developer to be able to test an application using only the APK file, knowledge of the package name and the name of the launching activity is needed. Furthermore, the APK needs to be signed with the same signature as your test project to be able to test it. To conclude, Robotium supports gray-box testing (a combination of white-box and black-box testing).
**Automatic event generation:**
There is currently no support for automatic event generation in Robotium.

**Emulator/real device support:**
Robotium provides support for both emulators and real hardware devices. This is mainly because it is executed as a normal application, and therefore it is possible to run it through the IDE as an emulator, or connect a real device and use it for debugging.

**Version support:**
Robotium has great support for both older versions of the Android API, going down to API level 8 (Froyo) and also providing support for the newer versions.

**Version control support:**
Robotium does not provide any support for version control, external software will need to be needed and manually add the scripts.

**IDE support:**
Robotium has support for both Eclipse and Android Studio since it’s easy to integrate Robotium to the development environment, as mentioned before.

**Resume application:**
There is no support in Robotium to resume a suspended application.

**Suspend application:**
Robotium does not have specific API support to suspend the current application being tested. However, there is support to simulate pressing of the “home” button of the Android device, which enables Robotium to suspend an application that way.

**Finish activity:**
With the API (as explained in the API criteria section), the Solo object provides support for activities. The method finishOpenedActivities() works to finish all activities that have been navigated to by the framework but are still running in the background.

**Change device settings:**
There is no support to access the settings menu in Android devices. There is, however, support for certain settings that are accessed often, for example screen orientation, volume, wi-fi and mobile data through the API.

**Scrolling:**
The Robotium API has great support for scrolling in applications. It supports both horizontal scrolling if activities have a swipe functionality, and vertical scrolling.

**Simultaneous keypresses:**
No information within the Robotium API has been found about simultaneous keypressing. However, it might be possible to achieve this with the help of synchronized threads. This approach will require work from the developer since it's complex and therefore it is not considered.

**Re-launch application:**
Robotium test script classes are based of the class ActivityInstrumentationTestCase2 from the Android testing framework, which includes the launchActivity() method that can be used to launch the application currently being tested.

**Delay support:**
It is possible to add a certain delay in Robotium, both with the Solo.sleep() method, that allows the developer to put Robotium to sleep for a certain amount of seconds. There is also support to wait until a certain condition has been fulfilled, for example to wait until a certain string has appeared with the Solo.waitForText() or wait until a certain activity has been opened with the Solo.waitForActivity().

**Condition based testing:**
In Robotium, there is great support for assertions to check certain conditions. For example, it is possible to make use of the Assert.asserttrue method from the JUnit framework to cancel the testing operation if a string is not found using Solo.searchText(). It is also possible to assert the current activity running using Solo.assertCurrentActivity.

**Support for device keys:**
Robotium has support for all the common hardware device keys that Android devices have. For example, Solo.goBack() simulates the "back" button on Android hardware devices. There is also support to send key codes from the Android API to the Solo.sendKey() method to simulate other hardware keys like the volume up/down buttons.

### 7.2 Selendroid

Selendroid is based on Selenium to be able to give full support to both hybrid and native Android applications. The reason Selendroid was selected for comparison is to try to identify any critical differences/advantages/disadvantages that separate it from the other frameworks. Version 0.15.0 was used in this study.

**API:**
The API for Selendroid is moderately difficult to work with; this is mainly because of the lack of documentation of how the framework is structured, and class names colliding with each other in different parts of the framework. However, since Selendroid is based off Selenium, it is possible to write Selendroid scripts in every language where there is a Selenium client binding, for example, Python, Ruby and Java.

The API is built up with several components that are responsible for different parts of the testing. SelendroidCapabilities is used to set the configuration of the tests, for example, require
the use of an emulator and setting the locale of the system. When performing the testing actions, such as clicking a button, the Selenium WebDriver API is used, where every view in the activity is translated into a WebElement.

Below follows a snippet of a Selendroid test script:

```java
// Find the button named "button1"
WebElement button1 = driver.findElement(By.name("button1"));
// Find and send the string “Text” to a field named “Enter text”
driver.findElement(By.name("Enter text")).sendKeys("Test");
// Click the button
button1.click();
// Get element under id “helloWorld”
WebElement helloWorld = driver.findElement(By.id("helloWorld"));

// Check that the text element has the text “Hello world”
Assert.assertEquals(helloWorld.getText(), "Hello world");
```

Logging support:
Logging in Selendroid is handled by sending log messages of the actions to LogCat. If the IDE does not support LogCat, log messages have to be written manually and are then displayed in the console. This makes logging in Selendroid very restrictive, since if support for other IDE’s that do not have access to LogCat are added, the logging has to be done manually.

Capture/replay support:
Selendroid does support capture/replay of click events with the help of a piece of software called Selendroid Inspector. This allows the developer to record click actions while exploring the UI of the application, and then create tests according to these click actions.

Documentation:
The documentation of the Selendroid API is quite difficult to grasp and not well structured. There are tutorials on how to get started on setting up the framework and information about some of the available features (see http://selendroid.io/). However, these tutorials do not cover all of the features, and the lack of a fully updated javadocs makes it difficult for developers to familiarize themselves with the framework. The author of this study, for example, had to analyze the source code of the framework to find information about the technical criteria mentioned later.

Testing method:
When searching for elements in Selendroid, the static class By is used. This class has methods to retrieve elements through ID, name, class name etc. Therefore it is possible to work with both black-box testing and grey-box testing in Selendroid.

Automatic event generation:
Selendroid does not support automatic event generation.

Emulator/real device support:
It is possible to run Selendroid-made scripts on both emulator devices, and on real hardware devices.

**Version support:**
Selendroid has support for almost all of the considered versions in this study, going down to API level 10 (Gingerbread) and all the way up to API level 19 (KitKat).

**Version control support:**
Selendroid does not support version control.

**IDE support:**
From the information gathered during this study, only Eclipse has been mentioned as fitting IDE for Selendroid. No other information has been found, and therefore no other conclusions can be made.

**Resume application:**
The SelendroidDriver makes it possible for the developer to use the resumeApp() method, which makes the current running application resume itself from the background state.

**Suspend application:**
With the SelendroidDriver class, there is also a method that allows for suspension of the currently running application with backgroundApp().

**Finish activity:**
It is possible to finish a currently displayed activity with the use of the back button in Selendroid. This can be done using the sendKeyEvent() mentioned later.

**Change device settings:**
Selendroid has support for certain settings, for example changing the current locale of the system, turning on airplane mode and changing the screen orientation. It is not possible to access all of the settings however, which makes the support limited.

**Scrolling:**
With the TouchAction class, there is support for scrolling with the method scroll(). TouchActions can also be used to perform swipes that can be used to scroll a certain view.

**Simultaneous keypresses:**
Selendroid has support for gestures that can be executed in parallel. To do this, the developer has to create two TouchAction objects with the help of the class TouchActionBuilder, that is used to create sequences of gestures that will be executed. After, a MultiTouchAction object has to be created by providing the two newly created TouchAction objects and then calling the perform() method.

An example of how this can look is demonstrated below:
Re-launch application:
Selendroid does not provide support to re-launch the application being tested.

Delay support:
Delay support is available in Selendroid with the help of the wrapper class AndroidWait, which allows the developer to avoid making errors when using the built in Android sleep methods. It's not always the case that the developer knows how the testing framework is threaded, and the Android waits can easily collide with the framework. The until() method in AndroidWait can be used to make the test case sleep until a certain condition is fulfilled.

Condition based testing:
With the Selendroid API, the objects that represent views in the activity, like buttons for example, are well designed with methods that retrieve the current state of that view. This enables the developer to write scripts that make use of conditions, since all objects are very well designed.

Support for device keys:
Selendroid has support to interact with the device by sending Android key events through the method sendKeyEvent() available through the Selendroid driver.

7.3 UI Automator
UI Automator comes with the Android SDK and is supported by Google Inc. It only supports native applications, and not hybrid applications. It was added into the comparison for the reason that it’s part of the standard Android SDK and it’s features and functionality is important to compare with other available frameworks since every Android developer is able to use it without installation. Version 2.0 available in API level 19 was used in this study.

API:
The API for UI Automator is easy to work with because of its structure and support for many areas of Android GUI testing. Scripts are written in Java, and the framework is now based on the Android instrumentation framework as of version 2.0, which allows the developer to invoke the methods derived from the Android instrumentation class. UI elements can be detected through the element description (i.e. text on the button etc.) or by the resource id (if the developer has access to internals).

UI Automator is built up by a number of objects. The main objects in the API are UiObject, that represents a UI element that is visible for the user, UiScrollable, that is used to search for items
in a scrollable UI container, UiSelector, that is used to query for one or more UI elements, and finally UiObject2, that is similar to UiObject, but cannot be reused in other views than the one it was referenced in.

In some cases, the ID of certain elements on the UI is needed to perform a certain action. For this, a tool named UI Automator Viewer comes with UI Automator that looks at snapshots of the screen and shows the ID that is needed of every element. Down below follows a snippet made with UI Automator, where comments describe the intended action:

```java
// Get the current device
UiDevice device = UiDevice.getInstance(getInstrumentation());
// Press the home button on the device
device.pressHome();
// Wait for a window transition with a timeout of 4 seconds
device.waitForWindowUpdate("", 4000);

// Find the applications button and click it
UiObject2 appsBtn = device.findObject(new UiSelector().description("Apps"));
appsBtn.click();
device.waitForWindowUpdate("", 4000);

// Get the applications view
UiScrollable apps = newUiScrollable(new UiSelector().scrollable(true));
// Change orientation to horizontal
apps.setAsHorizontalList();

// Get the settings button from the list
UiObject settings = appViews.getChildByText(new UiSelector().className(TextView.class.getName()), "Settings");
// Click and wait for window transition
settings.clickAndWaitForNewWindow();

// Check if application was found
assertTrue("Application was not detected", settings != null);
```

**Logging support:**
UI Automator does not offer any specific logging capabilities other than the ones supported by the IDE running the framework. The current state of the instrumentation is logged, but for more detailed log messages, it has to be done manually. It does not allow any log report generation other than exporting the IDE log to a plain text file. To conclude, UI Automator has no direct logging capabilities, and has to be done manually.

**Capture/replay support:**
UI Automator does not provide any support for capture/replay.
**Documentation:**
The API is well-documented through their Android developer page (see https://developer.android.com/reference/android/support/test/uiautomator/package-summary.html) that explains all the methods connected to each class. For further support on the framework, Google Inc. has a group in Google Groups (see https://groups.google.com/forum/?fromgroups=#!forum/android-platform) focused on Android where questions regarding UI Automator can be answered both through the community, and by the developers. Though, it is not fully committed to only UiAutomator.

**Testing method:**
With UI Automator, developers can choose to work with only black-box testing by using the content of the UI elements to identify them. The developer can, however, also take advantage of knowledge of the internals, and instead of using content of the UI element (like text on buttons for example), the ID can be used. This makes the framework support both black-box testing and gray-box testing.

**Automatic event generation:**
There is no support for automatic event generation in UiAutomator.

**Emulator/real device support:**
UI Automator scripts can be run on both emulators and real devices. The projects made with UI Automator can be deployed to any device and run from there.

**Version support:**
UI Automator has limited support for the Android versions, it was added in API version 18 (Jelly Bean) and therefore does not support versions earlier than that. With the versions mentioned in figure 1, only Jelly Bean and KitKat are supported by UI Automator.

**Version control support:**
As with Robotium, UI Automator consists of only a jar file and the UI Automator Viewer. Therefore, it does not support version control.

**IDE support:**
UI Automator is compatible with both Eclipse and Android Studio. Eclipse is the supported IDE originally, but since it’s part of the Android SDK, it’s possible to integrate into Android Studio as well.

**Resume application:**
The API provides access to methods that allow the developer to open the recent applications tab available on Android devices. Therefore it is possible to navigate through this tab, and click on the desired application that should be resumed.

**Suspend application:**
UI Automator does not have specific API support to suspend a running application. There is, however, support to simulate the “home”-button on the device which makes UI Automator able to suspend the application.

**Finish activity:**
There is support to finish activities with UI Automator. However, the API does not directly support it, but it can be closed with the help of the UiDevice method pressRecentApps(). UiAutomator can also simulate the “back”-button to close the application and therefore invoke the onDestroy() method on the activities opened.

**Change device settings:**
UI Automator allows the developer to navigate through all the available screens in the device that is used for testing. Therefore, the settings application is accessible, and is able to change any setting.

**Scrolling:**
There is support for scrolling capabilities with the UiScrollable object in the UI Automator API. This object provides support for fling gestures, as well as normal scrolling and horizontal scrolling.

**Simultaneous keypresses:**
It is possible to perform multi-touch actions with the help of the UiObject class. The UiObject.performMultiPointerGesture() method allows for multiple touch gestures defined by the developer.

**Re-launch application:**
Since UI Automator has support to navigate through all the screens in the device, it is also possible to navigate to the “applications” tab. Within this tab, it is also possible to click on the desired application to make it run. Therefore, re-launching of applications is supported by UI Automator.

**Delay support:**
The UiObject class in the UI Automator API provides methods to add delay into the testing. For example, the method UiObject.clickAndWaitForNewWindow() allows the developer to add a waiting time (with support for timeouts) until a windows transition happens. With the methods UiObject.waitForGone() and UiObject.waitForExists(), the developer may also add a delay until a certain view is undetectable or visible.

**Condition based testing:**
UI Automator provides support for condition based testing with the UiSelector class. This class contains methods that are used to search for a specific UI element, or a certain view. For example, the method UiSelector.text() allows the developer to search for an element that contains the string specified.

Below follows an example of how condition based testing can look with UI Automator:
UiObject object = new UiObject(new UiSelector().text("UI Element"))
if (!object.exists())
    // Do something
else
    // Do something else

Support for device keys:
There is API support for many of the device keys, where they exist as methods in the UiDevice class. Home button simulation is for example done by calling the UiDevice.pressHome() method, and functionality such as accessing the recent apps menu by calling UiDevice.pressRecentApps() is also available. Support for Android key codes is also available.

7.4 Espresso
Espresso is another framework that is available through the Android SDK. It was created by Google Inc. to revolutionize Android GUI testing with a simple API and with minimal lines of code. It was added into the comparison because it’s the most recently released framework compared to the other. Therefore, it is interesting to analyze how it handles the identified problem areas from this study. Version 0.2/2.1 was used in this study.

API:
The API for Espresso is very simple and well structured. Scripts are written in Java and the framework is based on the Android instrumentation framework. It makes use of three types of actions, ViewActions (for example, clicks, scrolling, type text), ViewMatchers (used to check if a view matches a specific criterion, for example, has a specific text, is clickable) and ViewAssertions (used together with ViewMatchers to stop testing if a certain property is not fulfilled).
Mostly identification of UI elements happen with the help of resource IDs, but there is also capabilities to find elements with strings connected to the element.

These actions can be used with the onView (used to sent interactions to a view that is part of the hierarchy) and onData (used to interact with views that are, for example spinners, lists). Below is a snippet of how Espresso works:

// Find the button with id “button1”
ViewInteraction button1 = onView(withId(R.id.button1));
// Find the text “Hello world”
ViewInteraction helloWorld = onView(withText("Hello world"));

// Click the button
button1.perform(click());
// Check if the text was displayed
helloWorld.check(ViewAssertions.matches(isDisplayed()));

Logging support:
Since Espresso test cases are based off JUnit (just as Robotium), it is therefore possible to generate test reports in JUnit XML format with the help of the IDE running Espresso. Espresso also has great support for logging messages if the IDE supports LogCat, since all views are logged and sent to LogCat. It also provides a hierarchy view over the views where the failure occurred which allows the developer to see exactly the steps taken to make the failure appear.

**Capture/replay support:**
Espresso does not support capture/replay.

**Documentation:**
There are numerous tutorials available, written by the authors on the wiki section, available on Google Code (see [https://code.google.com/p/android-test-kit/wiki/Espresso](https://code.google.com/p/android-test-kit/wiki/Espresso)). For support, the community for Espresso is growing quickly, and there is a support group on Google Groups that is visited actively by the authors and experienced developers that can answers questions or discuss anything regarding the framework (see [https://groups.google.com/forum/#!forum/android-test-kit-discuss](https://groups.google.com/forum/#!forum/android-test-kit-discuss)).

For the API, there is a well-documented javadocs page available on the Android developer page (see [https://developer.android.com/reference/android/support/test/espresso/package-summary.html](https://developer.android.com/reference/android/support/test/espresso/package-summary.html)) where all objects and methods are well described and easy to find.

**Testing method:**
Espresso supports only gray-box testing, this is because the developer needs to identify the class name of the startup activity in the application that’s being tested. When writing test scripts it is possible to make use of string names of the elements to identify them, which allows the developer to write regular black-box test cases. However, the requirement that the developer needs knowledge of which class is the startup activity makes it only possible to conduct gray-box testing in Espresso.

**Automatic event generation:**
Espresso does not support automatic event generation.

**Emulator/real device support:**
It is possible to run tests made with Espresso both in emulator environments, and on real devices.

**Version support:**
Espresso was designed to support most versions of Android. It supports Froyo (API level 8) up to the newer versions. To conclude, Espresso supports all versions of Android considered in this study.

**Version control support:**
Since Espresso only provides an API, there is no built-in version control support.
IDE support:
Espresso has support for both Eclipse and Android Studio, since it’s part of the Android SDK.

Resume application:
It is not possible to resume a suspended application with Espresso since it’s only possible to perform actions within the application that was stated in the configuration.

Suspend application:
As mentioned later when device keys support is evaluated, Espresso has support for all the available Android key events, which allows for the “home”-button to be simulated, and therefore suspend the application.

Finish activity:
Espresso has support for the “back”-button of Android hardware devices, and through that, the activity that a user is currently browsing can be finished through this approach.

Change device settings:
It is not possible to change any settings, other than the ones that can be changed through keycodes, like for example, the volume.

Scrolling:
Within the available ViewActions, there are methods that can be used for scrolling. For example, scrollTo(), swipeUp() and swipeDown(). Since swiping is the only method a user has for scrolling in a view, Espresso has excellent support for simulating this.

Simultaneous keypresses:
No information regarding simultaneous keypresses in Espresso has been found. However, the API does not specifically point out support for this, and therefore the conclusion is that there is no support.

Re-launch application:
As with Robotium, Espresso test case classes have access to methods inherited from the Android Instrumentation framework. Therefore, the launchActivity() method can be used to re-launch the application, but the API does not specifically support it.

Delay support:
In Espresso, delays are handled automatically through the framework, and therefore it is not needed to add delays to compensate for slow emulators etc.

Condition based testing:
The ViewMatchers that are available with the Espresso API make it possible for developers to check for certain properties, for example, if a view is visible or clickable. These can be used in combination with a ViewAssertion to stop execution when a property is not fulfilled.
Support for device keys:
With the available ViewActions, there is a method called pressKey() that can be used to send Android keycodes to the view that is being interacted with.
8 Result

The result from the evaluation of the general criteria (see table 1) shows clearly how all frameworks lack support for version control and automatic event generation. The main reason for this might be that the frameworks chosen for this study are mainly provided without third-party software that adds support for these criteria. What’s also interesting to note is that all of the frameworks have support for both emulators and real hardware devices when testing.

<table>
<thead>
<tr>
<th>General criteria</th>
<th>Robotium</th>
<th>Selendroid</th>
<th>UI Automator</th>
<th>Espresso</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Very simple</td>
<td>Moderate</td>
<td>Simple</td>
<td>Very simple</td>
</tr>
<tr>
<td>Logging support</td>
<td>Average</td>
<td>Average</td>
<td>Bad</td>
<td>Good</td>
</tr>
<tr>
<td>Capture/replay support</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Documentation</td>
<td>Good</td>
<td>Bad</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>Automatic event generation</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Emulator/real device support</td>
<td>Both</td>
<td>Both</td>
<td>Both</td>
<td>Both</td>
</tr>
<tr>
<td>Version support</td>
<td>API 8+</td>
<td>API 10+</td>
<td>API 18+</td>
<td>API 8+</td>
</tr>
<tr>
<td>Version control support</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>IDE support</td>
<td>Eclipse, Android Studio</td>
<td>Eclipse</td>
<td>Eclipse, Android Studio</td>
<td>Eclipse, Android Studio</td>
</tr>
</tbody>
</table>

Table 1. Comparison of the general criteria between the frameworks.

From the result, it is clear that there are only minor differences between the frameworks when looking at the general criteria. A big drawback with one of the frameworks is, however, the fact that UI Automator only has support for newer API levels (Jelly Bean and up), which can be a big factor when choosing a framework to work with because applications often need to support many different versions since there are users on many different types of devices.

Further findings include that Robotium and Espresso are the frameworks that provide the best backward compatibility support going back as far as to Froyo.
While these frameworks excel in that aspect, UI Automator is the framework that shows to have the worst backward compatibility support, with only Jelly Bean being the minimum version that can be tested. For this, Selendroid is a great complement as it is the closest framework to what UI Automator offers (seen in table 2).

It’s hard to draw any conclusions regarding the relevance of the general criteria. Automatic event generation and version control support are two criteria that none of the frameworks had support for, and it is hard to analyze whether it is something that the frameworks should strive to support, or if it’s irrelevant in this case. What can be said about the other criteria is that all frameworks seem to support both emulator and real devices, and they all support at least one of the two major IDE’s (Eclipse).

### Table 2. Comparison of the technical criteria between the frameworks.

<table>
<thead>
<tr>
<th>Technical criteria</th>
<th>Robotium</th>
<th>Selendroid</th>
<th>UI Automator</th>
<th>Espresso</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resume application</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Suspend application</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Finish activity</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Change device settings</td>
<td>Limited</td>
<td>Limited</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Scrolling</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Simultaneous keypresses</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Re-launch application</td>
<td>Limited</td>
<td>No</td>
<td>Yes</td>
<td>Limited</td>
</tr>
<tr>
<td>Delay support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Automatic</td>
</tr>
<tr>
<td>Condition based testing</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for device keys</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Looking at the table for the technical criteria (see table 2), it can be stated that Robotium and Espresso are not able to access all the states of the activity lifecycle since they do not support resuming of the application. UI Automator is also the only framework that allows for full activity lifecycle testing and automating of the criteria extracted from AQuA’s document.

The relevance of the technical criteria is a great subject in this study. They were identified from activity lifecycle testing and a number of use cases in AQuA’s high quality testing document. From what the result shows, the frameworks seem to handle these criteria well, since all the
frameworks support device keys, condition based testing, delays, scrolling, finishing of activity and suspending the application. However, since there are criteria that are not fulfilled by every framework, indications point towards that there are possibly more areas that need to be explored with Android GUI testing frameworks.
9 Discussion

The literature analysis used in this thesis to extract evaluation criteria to perform the framework comparison was able to detect multiple important areas with Android GUI testing. This made sure that the evaluation criteria were relevant and important from the Android GUI testing standpoint. However, it is not possible to state that criteria are relevant for comparing Android GUI testing frameworks, which is the biggest drawback by using this type of method. With using another approach like interviews, the relevance of the criteria can be assured by consulting developers that work with these type of frameworks on a daily basis.

Performance is one example of a criterion identified during the study that could potentially be of great value when comparing Android GUI testing frameworks, but had to be excluded due to the lack of time to perform experiments on all frameworks.

When performing the evaluation of the frameworks, API documentation and information gathered from different search engines was used. This allowed for a broader view over the frameworks because of the low resource cost for the data since no experiments were conducted. However, that affects the reliability of the result, since there might be potential issues with the framework when trying a certain feature that is not discovered by not conducting experiments which impacts the characterization that this study resulted in. By performing experiments, it would give more in-depth characteristics of the frameworks that would otherwise not be possible with just performing a literature review. However, the method chosen in this study allows for a broader overview, even though it’s also the limitation of using that method since the frameworks have not been tested with experiments. An overview of the frameworks only allows the result to be estimated, since there might be technical workarounds that will not be taken into consideration in this study.

Since the result shows that only two out of four studied frameworks actually support functionality to stimulate the activity lifecycle, the main component of Android GUI’s, and only a single framework provides full support for functionality to automate high quality tests. This clearly indicates that the area is interesting to study, and this study provides by no mean a definitive result to whether a framework is more suitable in a certain position. However, this study helps with understanding the problems that come with Android GUI testing, and the importance of evaluating common frameworks for developers.

Regarding the criteria API, logging support and documentation, the result may have been affected since they are using scales “simple”, “good” and “bad” to estimate how the frameworks perform in the certain criterion. However, due to them being rather insignificant when looking at the results, they have not affected the outcome of the study in a drastic way, but rather provide an estimation on how they perform in each criterion.

During the course of the study, the ethical aspects have been taken into consideration. The study has been performed in a scientific and methodical manner. The steps taken to perform the study have been mentioned (refer to chapter 4), so that others are able to find similar results.
when performing the same steps. All the related work that was used in the study has been properly referenced and the authors of the studies have been mentioned.

When putting the result from the study in a wider perspective, the study has resulted in evaluation criteria that may help researchers or developers compare other frameworks for Android GUI testing. This study can also help developers or organizations to find the right tool for their type of testing needs. Even though the evaluation criteria were developed through a literature analysis instead of consulting organizations or developers that work with Android GUI testing, the criteria are able to give an overview of how the frameworks perform and how they handle the most critical parts of Android GUI testing.
10 Conclusions

The question of this study was "Is there an open source Android GUI testing framework that is more preferable to other open source frameworks in terms of a chosen number of evaluation criteria?". To answer the question, evaluation criteria was set up with the help of literature analysis and relevant frameworks were picked for the comparison. As the evaluation was conducted, results showed that Robotium and Espresso are unable to fully exercise the activity lifecycle. However, they both provide great backward compatibility, with simple API’s that helps when developing at a faster pace, these two frameworks have few criteria that sets them apart, and therefore it is hard to say which one is more preferable.

UI Automator is the only framework to provide full support for both the activity lifecycle and the high quality tests from AQuA. The bad backwards compatibility support, however, is a big drawback when developing for many types of users. Selendroid is however, the only framework to provide good backwards compatibility support and also be able to fully exercise the activity lifecycle. It also has capture-replay support to compensate for the more complex and badly documented API.

The study showed that two out of four frameworks evaluated were not able to fully exercise the activity lifecycle, one of the main components in the Android GUI. When looking at the result from the previous chapter, there are similar results to what Kropp and Morales (2010) found in their study. The frameworks show lack of capture/replay support, with the exception of Selendroid, and no framework is able to provide automatic generation of events. Furthermore, the study showed that the area contains many problems that are open for further research. To conclude, it is not possible to say that any of the frameworks is more preferable than the other seen to the evaluation criteria, since it varies from what the developers’ needs are.

10.1 Future work

When performing the study, there were several other areas that appeared to be interesting to investigate, but were not addressed in this study. For example, it would be interesting to study the potential problems with iOS (iPhone operating system) GUI testing, and potential issues and risks that come with it, and how frameworks for automated iOS GUI testing handle these issues and risks.

It would also be interesting to do more research to see if there are more criteria that are important with Android GUI testing. Hybrid applications are a topic that was only mentioned briefly in the study. Doing more research in how frameworks handle criteria when it comes to testing of these types of applications would be interesting, since they are becoming more common.

Another interesting topic that appeared during the study is crawling-based Android GUI testing. It would be interesting to make a comparative study on the frameworks that are available for crawling-based Android GUI testing, and find out what the important criteria to look at are.
References


Creative Commons. 2015. Creative Commons Attribution 2.5 Generic. Retrieved 13 February, 2015, from Creative Commons: http://creativecommons.org/licenses/by/2.5/


Memon, Atif, Pollack, Martha and Soffa, Mary Lou. 1999. Using a Goal-Driven Approach to Generate Test Cases for GUTS. *Proc. of the 21St Int. Conf. on Software Eng.*, Los Angeles, CA, pp 257-266.


