Comparison of GUI testing tools for Android applications
Abstract

Test automation is an intriguing area of software engineering, especially in Android development. This is since Android applications must be able to run in many different permutations of operating system versions and hardware choices.

Comparison of different tools for automated UI testing of Android applications is done in this thesis. In a literature review several different tools available and their popularity is researched and the structure of the most popular tools is looked at. The two tools identified to be the most popular are Appium and Espresso.

In an empirical study the two tools along with Robotium, UiAutomator and Tau are compared against each other in test execution speed, maintainability of the test code, reliability of the test tools and in general issues. An empirical study was carried out by selecting three Android applications for which an identical suite of tests was developed with each tool. The test suites were then run and the execution speed and reliability was analysed based on these results. The test code written is also analysed for maintainability by calculating the lines of code and the number of method calls needed to handle asynchrony related to UI updates. The issues faced by the test developer with the different tools are also analysed.

This thesis aims to help industry users of these kinds of applications in two ways. First, it could be used as a source on what tools are overall available for UI testing of Android applications. Second, it helps potential users of these kinds of tools to choose a tool that would suit best their needs, depending on if they are most interested on test execution speed or in the easiness of developing a reliable test suite.

The results of this study are the first benchmarking of these Android UI testing tools. From this thesis one can see how a few of the tools available fare up against each other in a small-scale study. Tau and Espresso share the first place in maintainability, since Tau requires the least amount of lines of code, but Espresso requires no waiting method calls to ensure synchronization of the UI. While Tau faced least general issues during the test suite development, Espresso is clearly the fastest of the tools and it attains highest level of reliability, so it seems to be the best choice of the tools.

Keywords
Software Testing, UI testing, Android, Test Automation, Appium, Espresso, Robotium, Tau, UiAutomator

Supervisor
D.Sc. Professor Mika Mäntylä
Foreword

I want to thank my thesis supervisor Mika Mäntylä for his great patience, his guidance during my work for this thesis and that he at a times pushed me to continue this work so that it is finally finished at this point. I would also like to thank him greatly for the opportunity to work on this thesis, since it was he who had the general idea in mind when he inquired if I wished to work on it. Without him I would have never worked on this subject.

I’d also like to thank the University of Oulu and my department for providing me a chance to study in there and learn much more of the world. A word of thank you should also go to Miikka Kuutila and Päivi Raulamo-Jurvanen for their help in making the long summer during which I worked on this thesis go much faster and for their hints and help while I worked with this.

Thanks to my family without whom I would not be where I am now, and finally a big thank you to Ilona Myyri, without whom my life would be a lot bleaker.
## Contents

Abstract ................................................................................................................................. 2  
Foreword ................................................................................................................................. 3  
Contents ................................................................................................................................. 4  
1. Introduction ......................................................................................................................... 6  
2. Prior Research .................................................................................................................... 9  
   2.1 Android applications ..................................................................................................... 9  
   2.2 GUI testing ................................................................................................................ 9  
       2.2.1 GUI testing on Android ....................................................................................... 10  
   2.3 Android GUI testing tools and libraries (RQ 1.1) .................................................... 11  
       2.3.1 Android testing tools in articles and blogs .................................................... 11  
       2.3.2 Closer look at the tools available for Android GUI testing .......................... 13  
   2.4 How do the tools rank in popularity? (RQ 1.2) ....................................................... 16  
       2.4.1 Tool popularity in Google trends ...................................................................... 16  
       2.4.2 Tool popularity in Stack overflow .................................................................... 18  
       2.4.3 Which are the most popular tools ..................................................................... 20  
   2.5 Architecture of the popular tools (RQ 1.3) ................................................................ 20  
   2.6 Automatic test generation for Android (RQ 1.4) ................................................... 26  
3. Methodology ...................................................................................................................... 30  
   3.1 Research methods ....................................................................................................... 30  
   3.2 Research questions ....................................................................................................... 31  
       3.2.1 Research Question 1 ......................................................................................... 32  
       3.2.2 Research Question 2 ......................................................................................... 33  
   3.3 Empirical research ....................................................................................................... 35  
       3.3.1 Applications chosen for evaluating the testing tools/libraries ....................... 36  
       3.3.2 Devising the test set ......................................................................................... 38  
       3.3.3 Running the test set ......................................................................................... 43  
       3.3.4 Test run environment ....................................................................................... 43  
4. Findings ............................................................................................................................. 45  
   4.1 General findings and sources of difficulty for the tools (RQ 2.1) ............................. 45  
       4.1.1 General points of problem .............................................................................. 45  
       4.1.2 Issue severity per tool ...................................................................................... 47  
   4.2 Execution times (RQ 2.2) ......................................................................................... 51  
   4.3 Maintainability (RQ 2.3) ........................................................................................... 61  
       4.3.1 Lines of Code ..................................................................................................... 61  
       4.3.2 Number of waits ............................................................................................... 63  
   4.4 Reliability (RQ 2.4) .................................................................................................... 64  
       4.4.1 Failures in the Notes test suite ........................................................................... 66  
       4.4.2 Failures in Amaze test suite ............................................................................. 66  
       4.4.3 Failures in Wikipedia test suites ........................................................................ 67  
   4.5 Ranking of the tools ..................................................................................................... 69  
5. Discussion and Implications .............................................................................................. 73  
   5.1 Encountered issues while developing the test sets (RQ 2.1) ..................................... 73  
   5.2 Execution times (RQ 2.2) .......................................................................................... 74  
   5.3 Maintainability (RQ 2.3) ............................................................................................ 76  
   5.4 Reliability (RQ 2.4) .................................................................................................... 78  
       5.4.1 A look at the tests that fail ................................................................................ 79  
       5.4.2 Costs related to failures .................................................................................. 81  
   5.5 Tool rankings .............................................................................................................. 81  
6. Conclusions ....................................................................................................................... 83
1. Introduction

Continuous integration is a practice where an individual developer rapidly combines his or her own work to the main development build that is being developed by a group of developers. This can be beneficial since if the integration efforts are left to the end of a software project, it can bring much uncertainty to when the piece of software is finally fully functional. (Fowler & Foemmel, 2006)

Test automation is a key concept in continuous integration according to Fowler & Foemmel (2006). Automated software tests are, in contrast to manual testing, tests that can be set to run without a person interacting with the piece of software when the test suite is started to run (Hoffman, 1999). Hoffman (1999) continues that to the tests to be automated, the test environment should be also set up automatically, the tests should automatically compare outputs of the software with expected values, and the test should report if the test passes or fails and should output report based on this.

Test automation is said to be a way to decrease the costs of testing in software development, which can be important since testing efforts are said to take up half of the software project’s budget in some cases (Ramler & Wolfmaier, 2006). Fowler & Foemmel (2006) continue that in a continuous integration a set of tests should be run before a change is committed from an individual developer’s local repository to a central repository. They also say that the suite of tests makes sure that the changes the individual developer has done to the code base of a software does not break the current build that he or she is developing, at least on portions that have a test built for them.

Android as an operating system is an interesting choice to research since it now holds the largest market share of mobile operating systems ("Smartphone OS market share, 2016 Q3" n.d.). This makes it an interesting research object since there seems to be a general interest towards it, both in the software industry and in the user space of mobile devices.

Testing seems to be important for applications developed for the Android operating system, as applications developed for this OS, as all software, can have bugs. (C. Hu & Neamtiu, 2011). Emulators that mimic Android OS have been developed to make it easier to develop and test applications being developed. However, just testing the application manually with an emulator is not enough since it is possible that there are bugs that only manifest themselves on an actual device (Gao, Bai, Tsai, & Uehara, 2014; Ridene, Belloir, Barbier, & Couture, 2010). Ridene et al. (2014) continue that one source of such bugs is that some devices don’t properly support HTTPS communication protocol that an app might use. The authors also mention that emulators for all versions of Android and different devices are not available.

Fragmentation of OS versions and devices is known to Android developers. An example of this fragmentation is that in the statistics that Google offers on the Android developer info pages, which shows that as of August of 2016 only 15.2% of the Android devices connected to Google’s Play store (digital distribution market for Android applications) are running the newest released version of Android, codenamed Marshmellow (API level 23), while 29.2% of the devices are still running KitKat, Android API level 4.4 which was published almost 3 years prior in October of 2013. ("Dashboards - platform versions" n.d.). Another example of this fragmentation is that in 2012 Netflix was accessed by 1000 different kinds of devices (Kher, 2012).
This fragmentation is a problem that needs to be tackled with when developing for Android (Ridene et al., 2010) and many different devices and versions of Android should be used to test the software (Choudhary, Gorla, & Orso, 2015). Another source also mention this, and it is said that much testing should be done on the application because of the high amount of fragmentation (Starov, Vilkomir, Gorbenko, & Kharchenko, 2015).

Considering all this it seems that adequately testing an Android application can be a big task, especially for smaller developer groups, as has also been mentioned by (Starov et al., 2015). Test automation would then seem very appealing since there would be a lot of work if the software under development would be tested by hand on many different devices and operating system versions. It would seem reasonable to find if some tools are faster than others at executing a suite of tests, since running the tests costs money through the infrastructure costs of the testing environment (Herzig, Greiler, Czerwonka, & Murphy, 2015). It also seems interesting to see if some testing tools are more susceptible to failing more often than others. Inspecting tests that fail for a reason not related to a code defect can be a waste of resources and money, especially if the number of failures climbs high (Herzig et al., 2015).

In their systematic mapping study Zein, Salleh & Grundy (2016) discussed about papers that are related to Android testing. They found out that there is an abundance of research related to methodologies, papers that evaluate or explain a new methodology that could be used for testing, such as papers discussing automatic testing tools mentioned previously. They identified that there is a lack of research related to helping engineers choose tools for the work they do related to testing on Android OS. They also recommended studies that would do such a comparison between test automation tools among other testing tools. The authors also criticize that many of the papers are set in a laboratory environment, and further research closer to industry setting would be recommended. (Zein, Salleh, & Grundy, 2016)

Automatic test generation has received interest in the scientific community. Tools that do this analyze an application and automatically generate inputs to it to see if it crashes. Papers presenting these tools can be categorized as “explaining a new methodology” as discussed in the previous paragraph, and they are numerous. Some developed tools are Dynodroid (Machiry, Tahliani, & Naik, 2013), AndroidRipper (Amalfitano, Fasolino, Tramontana, De Carmine, & Imparato, 2012) and SwiftHand (Choi, Necula and Sen, 2013).

A few papers were found that made a comparison between the different tools, such as a thesis where Appium, Robotium and UiAutomator were compared through many criteria, but no test tool performance analysis was done (Hökelekli, 2016). Another piece of research that was found is a bachelor’s thesis, where Robotium, Selendroid, UiAutomator and Espresso were compared only through the documentation that is provided for them (Esbjörnsson, 2015).

It would then seem that there could be a lack of knowledge about the performance aspects of the testing tools. This kind of knowledge could help engineers in an industry setting and individual developers wishing to start UI testing of their Android applications to choose the tools that would be good for their needs.

It was decided that a study comparing UI testing tools on Android OS would be carried out. The focus was put on user interface tests since these would be reasonably easy to develop as an external developer to a software project. There also seems to be several different tools developed for this purpose and they were different enough that it seemed
interesting to do a comparison between them. The scope was further narrowed to a situation where the developer would manually write the test scripts.

The research questions devised to research this subject are following:

RQ 1: What tools are available for GUI test automation of Android applications?
RQ 1.1 What tools are available for manual script generation for Android applications?
RQ 1.2 How do these tools order in popularity?
RQ 1.3 How do these popular tools differ in architecture?
RQ 1.4 What tools are available for automatically testing Android applications?

RQ 2: How do the most popular tools differ in performance?
RQ 2.1 What points of weakness do these tools have?
RQ 2.2 What are the differences in execution time?
RQ 2.3 How is the maintainability of the test scripts different between the tools?
RQ 2.4 What are the differences between reliability of the tools?

As can be seen in the research questions, the thesis is clearly split to two parts, a literature review considering the research question 1 and an empirical research considering research question 2.

The main contribution of this thesis is the benchmarking of five different Android UI testing tool with several criteria: general issues found, execution time, maintainability of test script code and the reliability of the test scripts constructed with the tools. This information can be used by people interested in these tools to figure out which of the tools might suit best their test automation needs. The results can also be used as a basis for future comparisons to see if the tools improve or decay with new versions.

The structure of the thesis is as follows: Chapter 2 contains the literature review, and discussion of different tools available as well as their popularity and structure of a few of them is discussed here. Along this, automated Android testing tools are addressed here. In chapter 3 the methodology used here as well as a closer look at the research questions and how the test setting is set up is discussed. In chapter 4 the results of the test runs and test development are discussed. In chapter 5 a discussion and some implications about the results are discussed, and finally in chapter 6 conclusions, limitations and further research is addressed.
2. Prior Research

At the start of this chapter a general look at Android applications and GUI testing is made. The research question 1 (What tools are available for GUI test automation of Android applications?) is answered in this chapter, with chapters 2.4, 2.5 and 2.6 answering research questions 1.1 (What tools are available for manual script generation for Android applications?), 1.2 (How do these tools order in popularity?), 1.3 (How do these popular tools differ in architecture?) and 1.4 (What tools are available for automatically testing Android applications?) respectively.

2.1 Android applications

Android applications (apps) can be put into three separate categories: native, hybrid or web-based. Native applications are ones that are built to run on the device’s OS, and if they were ported to another platform they would be re-made almost completely. Hybrid apps are ones that have a native wrap around them, but they are mostly a web app hidden inside that native wrapping, and web apps are applications that run on the browser of the device. (Joorabchi, Mesbah, & Kruchten, 2013)

Per Google’s documentation, hybrid apps on Android are constructed so that one would use an UI widget called WebView (“Building web apps in WebView” n.d). This widget is used to handle HTML and JavaScript code to display a web page to a user inside the application. An example of this is Wikipedia web page loaded from their servers that is then displayed to the user inside an Android application.

The apps are distributed as .apk files. It is a compressed file that includes, among other things, one or more .dex files and AndroidManifest.xml file. Android apps are often written in Java, and the Java code is compiled into .class files as is usually done with Java. These .class files are then changed into .dex files. (Zhauniarovich, Philippov, Gadyatskaya, Crispo, & Massacci, 2015). The new .dex files can then be utilized by the Android’s virtual machine, either Dalvik on older OS versions or ART on newer versions (ART and dalvik.n.d.).

When developing and Android application a minimum API level for it is chosen. This API level corresponds to a version of Android that is installed on an Android device, such as version 7.1 of Android (codenamed Nougat) corresponding to API level 25. This minimum API level indicates on how old version of Android the application can run on, the higher the API level is the fewer the number of devices the application can be installed on, but the higher the API level is the newer version of Android the application can rely on.

2.2 GUI testing

The issue of graphical user interface (GUI) testing is not new, since already in early 2000’s Memon wrote of testing GUIs, stating that they are “ad-hoc and largely manual” (Memon, 2002). Automating GUI testing means that a test script is created that drives the application under test (AUT), and that this script enters inputs to the AUT automatically and finally verifies that the application responds correctly (Kropp & Morales, 2010).
GUI testing in general can be difficult: the execution of the application can go to any direction depending on what the user happens to input to the system. It can become impossible to test all possible execution paths a user can take when using the application. Manually testing a GUI can be much work, flaws found can be difficult to reproduce and the practice is hardly scalable. (Kropp & Morales, 2010)

Mobile application testing is activity where native and web applications are tested by using “well-defined software test methods and tools” so we can try and make sure those applications function correctly and well enough for the user (Gao et al., 2014). Further, these activities can be done through different infrastructures, such as by crowd-sourcing the testing or by doing the testing activities on a physical device, on an emulator or in a cloud testing service (Villanes, Costa, & Dias-Neto, 2015).

The verification portion of GUI testing can be divided into three categories: bitmap comparison, using an API or by optical character recognition. Bitmap verification can be difficult to maintain because making small changes in the GUI of an application can cause the verification to fail. Verification through an API can also have its’ problems; wrong conclusions can be deducted because of the disparity of what an API states is the situation and what might be visible on the screen. (Takala, Katara, & Harty, 2011)

In 2012, automation of GUI testing of mobile applications was a problem that needed a solution that was not very readily available at the time (Muccini, Di Francesc0, & Esposito, 2012). In a systematic literature review of Mobile application testing, a lack of literature regarding choosing tools was pointed by Zein et al. (2016) and the authors also point out a lack of research in real development environments.

2.2.1 GUI testing on Android

It seems that Android GUI testing is in the same situation as general GUI testing was at least in the early 2000’s, since according to Joorabchi, et al. (2013), manual testing of Android applications was still very prevalent. Manual testing in this situation means that the tester is running the application in an emulator or a physical device and manually navigating through the application looking for errors or crashes (Joorabchi et al., 2013).

Takala, Katara & Harty (Takala et al., 2011) mention that testing used to be limited to testing a single application at a time. They continue that this was because the tools were reliant on the Instrumentation framework, tool which launches a test application (that has access to the AUT’s context) alongside with the AUT. This is usually not possible because of Android’s security model, applications do not usually have access to other application’s contexts. According to the authors this is the reason the tests were unable to navigate outside the application and that it needed access to the source code of the AUT. After this, tools have been developed that do not rely on the Instrumentation framework, such as UiAutomator. (Testing UI for multiple apps.n.d.)

When doing GUI testing on Android one must consider some issues. First, Android applications can have multiple entry-points into the applications. Also, the OS environment can have an unexpected impact on the execution of the program through events and Intents (Android OS level “messages” that applications can respond to – for example an application can start an intent to request a camera to snap a picture and provide it back to the application) provided by the surrounding system can have an impact on how the application works. User-driven nature of Android applications can also be a source of problems. (Zhauniarovich et al., 2015)
This is also mentioned in elsewhere: it is said that the omnipresence of the Android framework and the nature of code execution (snippets of code being called here and there) pose difficulties when compared to more “monolithic” and independent desktop applications. On the other hand, with knowledge of the framework along with metadata about the applications some testing activities can be easier to automate. (Mirzaei, Bagheri, Mahmood, & Malek, 2015)

When creating UI tests for an Android applications there are a few ways to do it. First one can write the test cases by hand (such as with Appium or Robotium), one can use a tool to record the steps that are performed by a test and a test script is automatically generated from this (such as is possible with Espresso and Tau). There are also some tools that can automatically generate inputs to the AUT, such as Dynodroid (Machiry et al., 2013)

2.3 Android GUI testing tools and libraries (RQ 1.1)

There are several different tools available for driving the UI of Android devices. The tools were identified through blog posts and the author’s own knowledge about different tools available. Most of these tools do not have scientific articles behind their development or function, so non-scientific sources had to be used instead. The information about them was gathered, if possible, from the source that seemed official for the tool. This means that the tool’s documentation was searched through its’ homepage, or alternatively GitHub page was searched. These tools are mostly responsible for communication towards the application under test (AUT), they are responsible for sending inputs to the AUT and of asserting that something has happened on the application (Kulkarni, 2015).

2.3.1 Android testing tools in articles and blogs

In a survey on Android automation testing tools several Android UI testing tools were identified: Robotium, Ranorex, Appium, MonkeyTalk and UiAutomator (Gunasekaran & Bargavi, 2015). In another study, Android Instrumentation Framework and Positron Framework are compared (Kropp & Morales, 2010). However, after a search on the internet it seems that Positron is no longer available for use and the Android Instrumentation framework is not mentioned as a tool to be used for UI testing of Android apps ("Automating user interface tests" n.d.). In a master’s thesis Robotium, Appium and UiAutomator were compared against each other (Hökelekli, 2016). In that thesis Calabash, MonkeyTalk, Selendroid, Espresso and MonkeyRunner were also mentioned.

To complement the tools identified in these articles and thesis, a look at different blogs and articles from non-scientific sources was used to expand the list of tools available. If a tool is mentioned in multiple blogs, it seems to have at least some popularity and could be studied further. Many blogs where mobile test automation was discussed were found, and they are presented in Table 1. The blogs were searched through a Google search with relevant search terms, such as “android ui testing tools” and “comparison android ui testing tool”. The relevant blogs and articles were inspected and tools mentioned in them were recorded.
As is seen in the Table 1, the blogs discussing the subject are numerous. As can also be seen in the table there are some tools that pop up in many of the blogs discussing about these tools. Robotium seems to be present in most of the blogs along with Appium, Calabash and UiAutomator. Selendroid, Espresso and MonkeyTalk were mentioned in a few blogs as well as Scirocco, Ranorex, Testdroid, MonkeyRunner, Monkey and Test Studio Mobile.

Some of these tools are not meant for writing scripts that automate UI testing of Android applications, some are cloud services that might run the test scripts on real devices, with some of these tools one just records a test that can then be re-ran and some of these tools are completely automatic. Short description of these tools is presented below. The tools that are used for writing GUI testing scripts for Android are presented in the following chapter.

Monkey is a tool that is not actually used for writing test scripts and running these, but a tool that automatically sends inputs to the device so random buttons are pressed and it is tested if the application will crash at some point. The tool is available for all Android devices. ("UI/application exerciser monkey" n.d.)
An official source for the MonkeyTalk was not found, it seems that the tool was at development at some point and was purchased by Oracle. The tool seemed to be a cross-platform tool for both Android and iOS, as discussed in a blog post on how to use the tool in question (Vasanth, n.d.).

Scirocco cloud is a commercial cloud-based tool where one can control a real Android device through a web browser. They also offer possibility to write test classes and have them run when necessary. ("What is scirocco cloud?" n.d.)

Testdroid is another cloud-based testing environment, though they offer devices on which the tests can be automatically run. They do not offer a scripting language of their own, but they support several testing tools such as Appium, Calabash and Espresso. ("Testdroid cloud" n.d.)

Finally, Test Studio Mobile is a commercial tool with which one can record a test and later replay it on a device. It seems one must configure the AUT for the tool to be able to record the test. Test Studio Mobile is also a cross-platform tool, being able to handle both native Android and iOS applications as well as hybrid Android applications. ("Getting started with native/hybrid iOS and android applications in test studio mobile" n.d.)

2.3.2 Closer look at the tools available for Android GUI testing

A look at the different UI testing tools that are available is done in this chapter. Key features for the tools are reported from the tools’ wiki pages, product descriptions or other documents that were found.

Table 2 below shows the tools that were identified to be usable in manual test script writing. The tools were identified through the blogs presented in Table 1, except for Tau which was presented at the start of the study by a company as a tool to be used. White/black box testing column means if the tester needs to have the source code of the application available, though as can be seen almost all the tools can work without needing access to it. The table also lists what API level the testing tool requires on the device where the tests are run – this ties directly to the Android version of the device on which the application under test (AUT) is run on. This can be interesting since if the tool requires a newer version of Android than the minimum level that is set for the application, it can be that the tool can’t be used to test the applications in every situation. Finally, the table also presents if the tool is free for use or if it requires some manner of licence.
Table 2. Discovered Android UI testing tools

<table>
<thead>
<tr>
<th>Name of the tool</th>
<th>White/black testing</th>
<th>box</th>
<th>API level</th>
<th>Year level released</th>
<th>API level released</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appium</td>
<td>Black</td>
<td>18+</td>
<td>UiAutomator or 10+ Selendroid</td>
<td>2012/2010</td>
<td>Open source</td>
<td></td>
</tr>
<tr>
<td>Calabash</td>
<td>Black</td>
<td>N/A</td>
<td></td>
<td>N/A</td>
<td>Open source</td>
<td></td>
</tr>
<tr>
<td>Espresso</td>
<td>White</td>
<td>8+</td>
<td></td>
<td>2010</td>
<td>Open source</td>
<td></td>
</tr>
<tr>
<td>Monkeyrunner</td>
<td>Black</td>
<td>Any</td>
<td></td>
<td>2008</td>
<td>Open source</td>
<td></td>
</tr>
<tr>
<td>Ranorex</td>
<td>Black</td>
<td>N/A</td>
<td></td>
<td>N/A</td>
<td>Non-free</td>
<td></td>
</tr>
<tr>
<td>Robotium</td>
<td>Black/white</td>
<td>8+</td>
<td></td>
<td>2010</td>
<td>Open source</td>
<td></td>
</tr>
<tr>
<td>Selendroid</td>
<td>Black</td>
<td>10-19</td>
<td></td>
<td>2010/2013</td>
<td>Open source</td>
<td></td>
</tr>
<tr>
<td>Tau</td>
<td>Black</td>
<td>N/A</td>
<td></td>
<td>N/A</td>
<td>Non-free</td>
<td></td>
</tr>
<tr>
<td>UiAutomator</td>
<td>Black</td>
<td>18+</td>
<td></td>
<td>2012</td>
<td>Open source</td>
<td></td>
</tr>
</tbody>
</table>

A common theme with the tools presented here is that they are mostly not concerned with how the app handles the context of the device – physical location, brightness, sound environment, or the data connectivity.

Google’s testing support library has two tools, Espresso and UiAutomator. The tools are offered for free as part of the Android Open Source Project. ("Testing support library" n.d.). These tools use Java as their test-development language.

Espresso is meant for writing functional UI tests where the focus is on navigation inside the application under test (AUT). In fact, since Espresso is dependent on the instrumentation framework of Android, Espresso cannot navigate outside the AUT ("Testing UI for a single app" n.d.). The tests seem to be reasonably simple to write, since Espresso takes care that the application is at a stable state before continuing with the test script. Stable state means that the application is not waiting for an example for an animation or network call to finish (Nolan, 2015). One only needs to state the intent that should happen and Espresso takes care of timings.

UiAutomator on the other hand does not use the Instrumentation framework and can send inputs outside the application under test, for example visiting system settings with UiAutomator is possible where Espresso cannot do that. Unlike with Espresso, the test developer needs to take care that the application is in a stable state before test execution is continued. (Testing UI for multiple apps.n.d.)

Per the Google’s documentation, Espresso seems to be the recommended way of writing tests inside the single application. It is mentioned that Espresso is suitable for white or gray-box testing while UiAutomator is more suitable for black-box testing ("Testing support library" n.d.). Tests written for Espresso and UiAutomator are somewhat like JUnit tests, inputs are sent to the UI widgets, state of the screen changes somehow (new elements appear, a new activity is launched etc.) and one does an assertion on the state of
the UI widgets, for an example that they have a correct text or that some widget is not visible.

Robotium is a tool like Espresso, same kinds of UI level tests can be written where an UI widget is searched for, it is sent an input and assertions can be done afterwards. Like Espresso, Robotium is only limited to navigating inside the AUT as it is dependent on the instrumentation framework. The tool seems to be in active development. In contrast to Espresso and UiAutomator, Robotium is a 3rd party tool. When using the tool the library is simply imported to the Android application project. The test development language is Java. ("Robotium" 2016)

Monkeyrunner is another tool that could be used for UI testing. It is offered with the Android SDK and developed by Google. The tool is used for sending inputs, such as key strokes or touch events, to the application under test and for taking screen shots of the application. It also can apply test suites to multiple emulated devices and boot up said emulators. This tool is much lower level way of testing than for example Espresso is, but it could possibly be useful in automating the running of the test suite. Monkeyrunner should work on any Android version, since it functions on a level lower than the Android Framework. In contrast to the other tools presented here, it seems to lack high-level methods for searching and asserting on screen items. The language used for Monkeyrunner is Jython, an implementation of Python that uses Java. ("Monkeyrunner" n.d.)

Selendroid promises to be “Selenium for Android”, a reference to the web testing tool Selenium. Selendroid has a client-server architecture where the steps in the test script are commands that are sent by the Selendroid library to the Selendroid server. The server then commands the device (either emulator or a real device) to do what the test script told it to do. It supports native and hybrid Android applications along with web applications. For native and hybrid applications, the test scripts can be written in either Ruby or Java, and for web applications Python can be used as well. It is a black-box testing tool and promises to offer different means of locating UI elements and different gestures that can be performed on those. It targets API levels from 10 to 19 on Android. ("Selendroid's architecture" n.d.)

Appium is a cross-platform black-box testing tool, which in the case of Android uses UiAutomator or Instrumentation framework (accessed through Selendroid module) for running the tests you write. For this reason, Appium can interact multiple applications if the Android API level on the test device is higher than what is required by UiAutomator, 18. It is very similar Selendroid as the tool sets up a web server that communicates with the device while the test script communicates with the web server through a REST API. The way the tests are written is by using different bindings, which are available in Ruby, Python, Java, JavaScript, PHP, and C# as code libraries. The method calls for these libraries then send the test commands to an Appium server that handles communication with the device where the AUT is. Since only the web server communicates with the device where the test is being run, the same tests could in theory be used to drive both iOS and Android applications. Appium supports native, hybrid and web applications. ("About appium" n.d.)

Tau is a commercial tool that is more than just a library with API calls that drive the UI of an application. The application also provides many other tools that can be used to gather more information about the device or the test runs, such as battery statistics, execution logs or the Live view through which one can interact with the device through
the Tau client. Tau can handle interaction with applications besides the AUT. Tau’s test implementation language is Python. ("Deliver faster results and improve quality" n.d.)

Ranorex is another cross-platform testing tool and supports either writing the test suite by hand or by recording the steps to generate the test suite. It promises to work on normal Android or iOS devices without need for root access or jailbreaking, and is capable of recognizing objects on the screen and performing verifications on them. The recorded test suites can be replayed and it seems that this recording capability is the preferred method to generating the tests. The test suite can be modified with C# or VB.NET ("Mobile automation testing" n.d.)

Calabash is also another cross-platform testing tool, enabling the automation of both native and hybrid applications. Calabash tests can be written with Ruby, but their focus seems to be in Cucumber framework, where one does not write test scripts but tries to express through real-life intentions what he or she wishes to do. Calabash supports different gestures and assertions. Calabash uses Android’s Instrumentation for driving the UI. The documentation on neither the landing page of the project nor the GitHub page mention API level with which the tool is compatible with. ("Calabash" n.d.)

2.4 How do the tools rank in popularity? (RQ 1.2)

A look at the popularity of the different tools was made to figure out which tools should be looked at more closely. Since no scientific studies of the popularity of the tools in industry were found, more informal metrics had to be utilized. The two metrics for popularity were the relative Google searches done for the tools, inspected through Google Trends service¹, and through a forum that is popular with questions regarding computer science and programming, Stack overflow². There are of course other places where discussion takes place besides Stack overflow, but since those can’t be compared that was used instead to measure popularity. When the results of both these sources was combined, it seemed that Espresso, Appium and Robotium are three popular Android UI testing tools. After this the results get hazier, but UiAutomator and Monkeyrunner seem to be next followed by Selendroid, Calabash and Ranorex.

2.4.1 Tool popularity in Google trends

The tools were put into order by popularity using Google’s Trends. In Figure 1 are the more popular tools and in Figure 2 the less popular ones. Because of the limitations of Google Trends, the figures could not be combined. Based on the results the ranking of the tools seems to be in May 2016 as follows: Appium and Espresso on the top spots while UiAutomator, Robotium, Selendroid, Monkeyrunner, Calabash seem to be in the same figures and Ranorex is not searched as Android specific tool at all.

The keyword used for the tools was the tool’s name supplemented with word android. This is since if one would not add it to the end of the name, the results would be skewed towards Appium and Ranorex, both tools that are used for cross-platform testing where the other tools are Android specific.

¹https://www.google.com/trends/
²http://stackoverflow.com/
In the Figure 1 above we can see that Appium as an Android testing tool is a bit more popular than Espresso is, and those two tools seem to be the clearly trending ones. From the rest of the tools no clear winner can be seen. Decisive conclusions can’t be drawn from the data since adding android suffix to the end of Selendroid or monkeyrunner drops them completely off the chart, as happens with Ranorex. However, when comparing only term “Selendroid” and “monkeyrunner” to terms “Robotium Android” and “UiAutomator Android” the terms are in very similar popularity ratings, as can be seen in Figure 2. With a quick search, it seems that monkeyrunner and Selendroid as terms are only used for the Android specific testing tools, so a comparison like this should be somewhat legitimate.

Searching for “Tau Android” seems to not give reliable results either, at least regarding the testing tool Tau, since there seems to be many Android-related subjects that are named Tau, such as different applications in the Play store and a telecommunications company offering Android devices.
When looking at Figure 2, the results seem jumbled and no clear winner can be seen. Selendroid is often above the other tools, but in 2016 this seems to change. UiAutomator seems to have risen in popularity gradually over time. However, if only terms “Robotium” and “UiAutomator” are used without Android suffix, they are above “Selendroid”, “monkeyrunner” and “calabash android”. Robotium and UiAutomator too seem to be Android-specific terms, so it would seem reasonable to use those. Considering that terms “Robotium” and “UiAutomator” are above in popularity of “Selendroid”, “monkeyrunner” and “calabash android”, it seems the terms at least are more popular.

2.4.2 Tool popularity in Stack overflow

The tools were also compared by searching for the terms in Stack overflow, a community for programming-related questions. The tags were identified by the author in Stack overflow by searching for the tool’s name in the tag search and then reading the description provided for the tag to verify that the tag is related to the correct subject, the tool in question. All the tools except for Appium and Ranorex had a corresponding tag, and UiAutomator seemed to have two tags that were used. The data was gathered from stackExchange data explorer site¹. To make sure that the questions are related to android, the selection statement included a clause that the question must also include the tag “android”. Tau did not seem to have a tag of its’ own on the service.

¹ [https://data.stackexchange.com/](https://data.stackexchange.com/)
Figure 3 shows the number of questions each tag has received on Stack Overflow through the years. The figure shows somewhat similar results to the Google Trends figures shown earlier. Robotium used to be the most popular tool, but over time Espresso and Appium took over and are clearly the two most popular tools. In 2016 Calabash seems to take the third spot, UiAutomator the fourth spot and Robotium the fifth. However, when looking at the previous years, Robotium and UiAutomator have been on higher positions. As a note, Robotium’s first questions appeared already in 2010, but this was left out of the figure for the sake of clarity.

Table 3. Tags and number of questions on Stack Overflow overall in December 2016

<table>
<thead>
<tr>
<th>Tag name</th>
<th>Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>[android-espresso]</td>
<td>1199</td>
</tr>
<tr>
<td>[appium]</td>
<td>1147</td>
</tr>
<tr>
<td>[robotium]</td>
<td>881</td>
</tr>
<tr>
<td>[uiautomator]</td>
<td>360</td>
</tr>
<tr>
<td>[monkeyrunner]</td>
<td>354</td>
</tr>
<tr>
<td>[calabash]</td>
<td>217</td>
</tr>
<tr>
<td>[selendroid]</td>
<td>111</td>
</tr>
<tr>
<td>[ranorex]</td>
<td>5</td>
</tr>
</tbody>
</table>

The total number of questions per each tag is shown above in Table 3. The data for this table shows situation as it was in December 2016. As can be seen in the table, it seems
that Espresso, Appium, Robotium, UiAutomator and monkeyrunner are the top 5 in popularity of amount of questions that are asked from the community.

2.4.3 Which are the most popular tools

When combining the results from Stack overflow and Google trends, the most popular tools are Espresso and Appium. There could be a case made for the third tool being Robotium, monkeyrunner, UiAutomator or Calabash or Selendroid. When considering also the data from Stack Overflow, Appium and Espresso are the two most popular tools. Calabash, UiAutomator and Robotium are in the third, fourth and fifth spots, with no clear winner for now, though Robotium’s trend is clearly downwards. Overall, Espresso and Appium seem to be clear winners and ranking is difficult to establish between the other tools without more reliable data and further analysis.

For the purposes of this study Appium and Espresso warrant a closer look. From the three next tools Robotium and UiAutomator are chosen for a closer look. It would seem to be interesting to see how the old Robotium fares up against the newer tools, and it seems that when also looking at the historical data UiAutomator is slightly more popular than Calabash, although with this data it is not too clear which one would be more popular. Selendroid does not seem to have gathered large popularity, at least when looking at the questions on Stack Overflow. It also does not support versions of Android above API level 19, which is already several years old, so it is not chosen for this comparison.

2.5 Architecture of the popular tools (RQ 1.3)

The tools chosen for closer study are the four most popular tools identified in previous chapter: Appium, Espresso, Robotium and UiAutomator. In addition to these, Tau was chosen as an alternative tool since it’s developed by a local company and was offered freely for the comparison.

As mentioned by Takala et al. (2011), UI verification can be done through visual recognition or through an API. All the tools chosen for this study use the API verification, they search an element in the UI by certain search criterion, usually at least by “resourceId” (unique name provided by a programmer for an UI element) or text that the element holds. After the element is found, different assertions can be done on the element.

It has been presented that when doing GUI testing, the actual output shown is compared with the expected output shown when the input value is considered. This in GUI testing should be done by analysing a screenshot of the windows and other elements visible. (Memon, 2002)

None of the tools chosen here offer the possibility to take a screenshot and compare it with another screenshot automatically, and the verification through an API is done instead.

Some, but not all, of these tools offer possibility to interact with other applications besides the AUT. It could be argued that it is a useful feature to be able to do this, however it was mentioned in Android Developer Backstage podcast by Google’s Android engineers that this might be a bad idea, since if one writes a test where navigation is done in an application developed by someone else, your test suite might break when that other party
does an update to the UI of their application, causing you to lose test coverage (Norbye, Haase, Knych, & Zakharov, 2015).

The relationships between the tools can be seen below in Figure 4. As can be seen in the figure, Espresso and Robotium make use of the Instrumentation framework that comes with the Android software development kit. The other tools use some other ways of both identifying the elements on the screen and interacting with them, it is not explained in the documentation of either of the tools what mechanisms they use to interact with the UI of the device.

Instrumentation-based tests means that the tests are packaged into an separate .APK file (install package for Android applications), and that test application is run in the same process as the AUT is. Espresso, Robotium and UiAutomator are based on this concept. This is done so the test application can access the UI elements of the AUT ("Getting started with testing" n.d.). The documentation also states that this makes it necessary to run these tests with an emulator or a real Android device.

![Figure 4. Relationships between testing tools, modified from (Helppi, n.d.)](image)

### Table 4 Comparison of some key features of the tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Can access source code?</th>
<th>Require access to source code?</th>
<th>Integrates with Android Studio?</th>
<th>Access to ADB / shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Espresso</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Api lvl 21+</td>
</tr>
<tr>
<td>Tau</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Robotium</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Api lvl 21+</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Api lvl 21+</td>
</tr>
<tr>
<td>Appium</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Local machine only</td>
</tr>
</tbody>
</table>

In Table 4, IDE (integrated development environment) integration means if the tools can be integrated with Android Studio. Integration in this case means if the testing libraries can be added straight to the app’s project and if the test suite can be run from the IDE. Only Android Studio was considered since it is the officially recommended IDE for Android development by the Android development documentation ("Creating an android project" n.d.).
Ability to access source code means that the testing tool can access the source code of the AUT, or in other words the test tool runs tests as instrumented tests. This can be an advantage in some situations and is required to do certain things with for example Espresso. The advantage can be tied to the point about wiping application data, if the tool does not provide means to wipe an application’s data by some other means, one can create helper methods in the source code of the application itself to handle this, remove database data and remove shared preferences for an example. Doing these of course requires the tester to have knowledge about the codebase of AUT.

Requires access to source code means that the testing library requires that the test scripts are written in the same Android project as the source code of the application. The opposite of this is that the testing tool could test with the installable package of the application (the .APK file). This can be preferred, since the tester only needs the installable application to do testing on it.

Finally, in the Table 4 access to ADB (Android debug bridge) means that the testing tools offers a way to interact with the ADB inside the tests code. This could be useful in some situations, such as pulling screenshots taken during the test execution off the device, to give certain permissions to an application before running the tests or to set up the testing environment.

**Espresso and UiAutomator**

Espresso and UiAutomator are libraries that are imported to an existing Android application project. They come with the Testing Support Library that is a part of the Android Open Source Project.

Both libraries use AndroidJUnitRunner, a test runner that enables one to develop test classes similar as one would develop JUnit tests. The test runner takes care of loading the test package and the AUT to a device and running the test package there. Supported JUnit versions are 3 and 4. ("Testing support library" n.d.)

Test code that is written for these tools is included in the actual project where the application resides in, and so they can access the publicly available methods and classes in the AUT. However, since UiAutomator can access applications besides AUT, one does not have to put the test code in the same project where the application itself is. The tests developed with both tools are based on the instrumentation framework, so the test applications run in the same process as the AUT does on a real device or an emulator.

The main differences between these two tools are that UiAutomator can interact with other applications besides AUT, and that Espresso promises that one does not have to handle synchronization when tests are written. UiAutomator can navigate outside of the AUT since it does not rely on the Instrumentation framework unlike Espresso does, but uses some other mechanism for interacting with the UI of the device.

The point about Espresso is its’ unique attribute, other tools do not make any promises regarding this. It should remove any need to include any sleep or wait methods in the test code. Espresso accomplishes this by occasionally checking if the main thread of the application is idle. Since in an Android application only the main thread can do any updates to UI widgets, this is a very secure way to make sure that the application is in an idle state before the test execution continues. ("Testing UI for a single app" n.d.)

1https://github.com/google/android-testing-support-library
When developing the tests for Espresso, what needs to be done is that one sets an object annotated with Java annotation @Rule to tell the tool which activity should be started before the test is ran, and in the test script itself one would make for an example a following method call for static methods of the Espresso library:

```java
onView(withId(R.id.buttonDefaultNegative)).perform(click());
```

and with UiAutomator one first initializes an object called UiDevice by passing it an instance of the instrumentation framework, and then interacts with the device through methods provide by UiDevice, such as following:

```java
device.findObject(By.res("com.amaze.filemanager:id/buttonDefaultNegative")).click();
```

Espresso and UiAutomator tests are mostly recommended to be developed in Android Studio, the officially supported IDE for Android development. It is possible to use IntelliJ with the Android plug-in as well. These libraries can be used together so the tools could cover each other’s weaknesses if needed.

AndroidJUnitRunner is used to run the tests if they are run through Android Debug Bridge commands directly.

![Diagram](https://example.com/diagram.png)

**Figure 5** Instrumented tests on Android, modified from ("Getting started with testing" n.d.)

When the tests are run, following sequence of actions is taken:

1) Build from sources (app & test code), create .apk file
2) Deploy .apk file with test code to phone
3) Start executing test cases
   a. Start application
   b. Start interacting with the UI

In this case, the test code that a tester writes will start to execute at step 3, where she first starts up the AUT and starts to interact with the UI. The steps before this can be done by Android Studio or directly by the Gradle build system used in Android projects.
Robotium

The instrumentation framework was a very low-level UI testing tool where it was possible to write tests that test a single activity in an application (Kropp & Morales, 2010). The authors mention that tool was however a very low level tool that gave much power to the test developer, but made development of those tests cumbersome. Robotium was built at the time to abstract away this hard-to-use instrumentation framework. (Takala et al., 2011)

It was developed as an open-source project, original author being Renas Reda. As is with Espresso and UiAutomator, Robotium can access classes and methods that are publicly available in the AUT since the test code is included in the same project as the AUT’s source code is ("Robotium" 2016).

Robotium functions similarly as Espresso and UiAutomator do, the tests are instrumented tests and run in the same process as the app itself. However, Robotium tests extend an outdated ActivityInstrumentationTestCase2 and use JUnit3 instead of JUnit4 in the test cases. Since Robotium relies on the instrumentation framework, it can only handle interactions within the one application for which the tests have been written for ("Robotium" 2016).

When developing the tests one uses Solo class to interact with the test device. One first initializes this class by passing it an instance of the instrumentation framework, and then uses Solo class’ methods to interact with the device, for example with a following code call:

```
solo.clickOnView(solo.getView(R.id.buttonDefaultNegative));
```

The Robotium tests are developed with the same IDE with which the Android projects are developed – either Eclipse or Android Studio. It is possible to use IntelliJ with the Android plugin as well. As is with Espresso and UiAutomator, AndroidJUnitRunner is used if the tests are run through Android Debug Bridge.

In the case of Robotium, the situation where one would start to run the test suite is the same as it is with Espresso and UiAutomator. One would first build the code and create .apk file, deploy the application on the device, launch it and start executing the test suite. These tasks do not need to be done manually by the developer, since either Gradle or Android Studio can handle these, one only needs to start the testing task through command line or the Android Studio IDE.

Appium

Appium is the result of an open-source project that attempts to automate native applications on iOS, Android, and Windows mobile devices. It also offers automation of hybrid applications and mobile web applications (applications that run in the browser on the device) on iOS and Android platforms. Appium wraps the native testing tools that are provided for these different platforms into one API that conforms to the WebDriver API. ("Introduction to appium" n.d.)
Appium requires only that the app is either installed on the device or that a path to the .APK is supplied when the Appium server is started. It is an interesting tool, since in the case of Android, for finding elements and interacting with them it uses either UiAutomator or Selendroid in the background. It also chooses different tools depending on how new Android version on the device where the AUT is running, as it uses UiAutomator for Android version 4.2+/API level 17 or Selendroid (backed by Android’s Instrumentation) for Android versions 2.3+/API level 9+. If the device’s Android API level is 17 or higher, it can touch apps besides the AUT because it uses UiAutomator ("Native android automation” n.d.).

Appium’s general architecture is a client/server architecture. Appium will start a web-server that listens to a certain port for client connections. A client can connect to the web-server, after which it can send commands to the server. When a proper command is received, the Appium server will issue a command to the Android, iOS or Windows device that it detects. When the command is run, an HTTP response that tells what happened is sent back to the connected client. ("Introduction to appium” n.d.)

Driving the UI happens inside a session that has been initiated by a client. While initiating the session the client sends a list of so-called “desired capabilities” to the server, which indicate different settings under which the session is started, such as indicating which OS the test automation is aimed at or what is the path to the .apk file in the case of Android automation. ("Introduction to appium” n.d.)

The Appium server itself is written in Node.js. The clients that connect to the Appium server are written usually using one of the ready libraries that are built for different programming languages. These libraries provide abstraction of the WebDriver protocol that the server expects. The libraries are available for many popular programming languages, such as Java, Ruby, Python, PHP, JavaScript, and C#. (Shah, Shah, & Muchhala, 2014)

When the tests are developed with the Java bindings of Appium, one first initializes an instance of AndroidDriver that is then used to communicate with the device. This object can be used to get objects from the device with different parameters and to interact with them, for example with a method call such as this:

driver.findElementById("com.example:id/buttonDefaultNegative").click()

Appium tests can be developed with any IDE that supports the language used to develop the tests, since they are not paired to Android directly.

When the test suite is going to be run, the following steps need to be taken:

1) Build source (test code only)
2) Start Appium server
3) Start executing test cases
   a. Install application on the phone
   b. Launch application
   c. Start interacting with the UI

In the case of Appium the test code execution starts at step 3 where she provides Appium with the ready .apk that will be installed on the device. Alternatively, if the application is already installed on the device, installation step can be skipped, but per the usage instructions this does not seem to be the preferred solution.
The steps 1 and 2 are done by the developer or a test automation environment, and Appium test script takes over and starts the testing by sending the Appium server a command to install the application on the device. One can use practically any IDE that supports the chosen language for developing the Appium tests, since one only needs to import the relevant Appium libraries to the project.

**Tau**

Tau is a commercial tool developed by a company Profilence. The tool functions a bit differently from the other tools. It has its’ own client that is installed on the test development machine that needs to be used to run and write the tests. According to the tool’s documentation, the client starts a background service on the device or emulator and those two communicate through ADB. ("Deliver faster results and improve quality" n.d.)

It can handle multiple devices simultaneously and does not require any access to the source code of the AUT. The tests can be written either completely by hand, or can be partially copied from a command history that is generated when a tester clicks the device’s UI through a Live view tool. ("Deliver faster results and improve quality" n.d.)

Language used to develop the tests with Tau is Python. The tool requires that the AUT is already on the device, but it can handle interactions with the phone everywhere, not only for the application for which the tests have been written for. The documentation for the tool does not go very much into depth of how it finds objects on the device or how it interacts with them, though one could imagine that the background service it starts is used for this. When developing the tests one needs to call functions such as following:

```python
tap.resourceId("com.example:id/buttonDefaultNegative")
```

Tau tests are developed with the IDE offered with the product. The test developer does not need to worry about importing libraries unlike with the other tools, nor does he need to initialize an object through which the functions would be called, unlike with Appium, Robotium and UiAutomator.

With Tau the steps taken when the test suite is run are the following:

1) Install application on test device  
2) Launch application  
3) Start interacting with the UI

In the case of Tau test execution starts from the second step, since Tau expects the AUT to already be installed on the device, and at least on the version of Tau inspected there was no mention of ability to install the application on the device with Tau. One starts the application inside the test script and then the test script will start to interact with the application. Since Tau’s language is Python, the test code does not have to be compiled prior to running the tests.

2.6 Automatic test generation for Android (RQ 1.4)

Finally, a look at automatic test generation for Android applications was done. This was originally meant to be the focus of the thesis, but after several these tools were looked at it seemed that many of them were either inactive or not available for use on the internet, so a focus was switched to manual UI test generation instead.
Automating the test generation of functional Android tests has gained some interest in the research community during the past few years. There are many papers showing the development and usage of these tools, first ones starting to appear at early 2010’s. There seem to be three main approaches to automated test generation: random testing, model-based testing and testing with learning capabilities (Choi et al., 2013; Machiry et al., 2013; Moran, Linares-Vásquez, Bernal-Cárdenas, Vendome, & Poshyvanyk, 2016).

Random testing (or fuzz testing) is generation of random UI inputs for the AUT, such as with the Monkey tool in Android world. This approach is a purely black-box technique, since it does not require any knowledge of the AUT (Machiry et al., 2013). Per MacHiry et al. (2013) model-based testing is a methodology where the testing framework is given a model of the application’s GUI - how navigation works in there and from which screen with what inputs will we get to another screen. Automating the generation of the model is also a common approach (Moran et al., 2016), (Yang, Prasad, & Xie, 2013). This last approach is used for example in MobiGUITAR, a testing framework that automatically “crawls” through the GUI of the AUT and generates test cases for it. Another approach to building this model is where the model of the application is extracted through analysis of data that is gathered when the application is manually executed. This manner of model-generation would be even usable in a crowd-sourced way, which would of course be very useful in lessening the testing burdens (Linares-Vásquez, White, Bernal-Cárdenas, Moran, & Poshyvanyk, 2015).

Thanks to the relative simplicity of the GUIs of mobile applications, when compared to desktop variants, it could be possible to automatically generate more complete models for purposes of model-based testing (Yang et al., 2013). The authors continue that mobile application frameworks are easy to analyse in regards of possible user inputs and gestures. However, the authors continue that it is very important to figure out proper inputs since only tapping on widgets leaves out many inputs a user could give to the application. On the other hand, static code analysis methodologies can have a hard time understanding the event-driven and often asynchronous nature of Android applications (G. Hu, Yuan, Tang, & Yang, 2014). It then seems that there are pros and cons to automated testing on Android.

MacHiry et al. 2013, mention yet one more technique, systematic testing. This is a technique where each sequence of inputs leads to a unique path in the AUT, and the paths are automatically explored. They also state that a big problem with this approach per the authors is that the scalability of this approach is poor, the number of execution paths can become very high. These approaches also include a heuristic by which the AUT is explored, either a breadth or depth-first approach (Moran et al., 2016).

Another definition on how to generate the event sequences required for automated testing is by four following ways: static analysis of source code, dynamic analysis through executing the app and somehow giving inputs (GUI ripping), manual programmer definition of the event sequence or completely random inputs (Linares-Vásquez et al., 2015).
<table>
<thead>
<tr>
<th>Tool name</th>
<th>Status</th>
<th>Paper or other source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynodroid</td>
<td>Probably dead, Last commit: 29.3.2015, developer said they are not updating it (<a href="https://groups.google.com/forum/#!topic/dynodroid/hRIOe-9FZVE">https://groups.google.com/forum/#!topic/dynodroid/hRIOe-9FZVE</a>)</td>
<td>Dynodroid: An Input Generation System for Android Apps (<a href="http://www.cc.gatech.edu/~naik/dynodroid/">http://www.cc.gatech.edu/~naik/dynodroid/</a>)</td>
</tr>
<tr>
<td>Puma</td>
<td>Probably dead, Last commit in GitHub 2014</td>
<td>Puma: Programmable UI-automation for large-scale dynamic analysis of mobile apps (<a href="https://github.com/USC-NSL/PUMA">https://github.com/USC-NSL/PUMA</a>)</td>
</tr>
<tr>
<td>Android GUI Ripper</td>
<td>Probably dead, last post on wiki page in 2013</td>
<td>A Toolset for GUI Testing of Android Applications (<a href="http://wpage.unina.it/ptramont/GUI">http://wpage.unina.it/ptramont/GUI</a> RipperWiki.html)</td>
</tr>
<tr>
<td>MobiGUITAR</td>
<td>Not found available anywhere, otherwise would seem promising</td>
<td>MobiGUITAR – A Tool for Automated Model-Based Testing of Mobile Apps</td>
</tr>
<tr>
<td>Quantum</td>
<td>Not found available</td>
<td>Automated generation of oracles for testing user-interaction features of mobile apps</td>
</tr>
<tr>
<td>MonkeyLab</td>
<td>Not available for use</td>
<td>Mining Android App Usages for Generating Actionable GUI-based Execution Scenarios</td>
</tr>
<tr>
<td>AppDoctor</td>
<td>Available, but instructions on usage are very limited. Few contributions and stars, last commit 9 months ago.</td>
<td>Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor (<a href="https://github.com/columbia/appdoctor">https://github.com/columbia/appdoctor</a>)</td>
</tr>
<tr>
<td>SIG-Droid</td>
<td>Probably not available, not found for use with quick google search</td>
<td>SIG-Droid: Automated System Input Generation for Android Applications</td>
</tr>
<tr>
<td>TrimDroid</td>
<td>Tool available, did not get it to work</td>
<td>Reducing Combinatorics in GUI Testing of Android Applications (<a href="http://www.sdalab.com/projects/trimdroid">http://www.sdalab.com/projects/trimdroid</a>)</td>
</tr>
<tr>
<td>Robo test</td>
<td>Available, commercial</td>
<td>(<a href="https://firebase.google.com/docs/test-lab/robo-ux-test">https://firebase.google.com/docs/test-lab/robo-ux-test</a>)</td>
</tr>
</tbody>
</table>
It seems that in many automated test generation methods first goal is to generate a model out of the application under test, and then send different input paths to the application and see if the application crashes. It also seems that not a single technique is the only way to approach automated testing. Much research has been put into this subject, and many tools have been developed. Mostly these tools seem to be focusing on finding execution paths in the application that lead into the application crashing; validation of proper inputs is not something that in general is focused upon.

In Table 5 the identified tools are presented. Even though in many papers the researchers state that they have successfully utilized the automated test generation tool to find bugs in several applications, the tools either require a specific version of Android framework or of the android development tools making the use of the tool difficult or make it unreliable. The Table 5 includes the article’s name in which the tool is presented, and a possible source that was identified for the tool. The tools seem mostly be abandoned at this point or are not found through a Google search. This makes the tools uninviting and it is dubious if they could be used in an industry setting. For this reason, it was decided that the tools would be not considered further for study, and focus was put on the testing tools where one writes the UI test scripts manually.
3. Methodology

The research was carried out as a literature review followed by an empirical study. Results of this are presented in chapter 4 and discussion of these results in chapter 5. This is accompanied by a literature review and trend research relying on non-scientific sources, which is presented in the chapter 2.

3.1 Research methods

The study is split into two portions (research questions), in the first portion is the literature review where Android and UI testing tools available for it is described. The second portion consists of an empirical study where a test suite was built with the four popular tools identified in RQ 1.2, along with Tau that was chosen for the closer inspection as well. Three separate test suites were built for three different applications with the five chosen tools to study differences between these tools. The three applications that were chosen are open source applications that are freely available on Google’s Play Store for download. The empirical research is further expanded in chapter 3.3.

To identify the different tools that are available for UI testing on Android platform the literature review was carried out. The literature review consists of both scientific and non-scientific sources. This is since there is not much scientific literature regarding the UI testing tools for Android – especially literature that outlines what tools are overall available or descriptions of the tools. A look at non-scientific blogs was also given to search for different tools that could be evaluated in the study. To try to lessen the impact of poor sources of information multiple sources were searched to look for the information.

When ranking the popularity of the tools two main sources were utilized, Google Trends service and the number of Stack overflow questions. There is a possibility that this does not accurately present how popular the different tools are in an industry use, but since no study of the actual use of these tools was found nor was it in the scope of this study to do a large-scale study such as this, it was felt that these two sources of information would be adequate. Further, since the two sources seemed to indicate very similar results, it was felt that this would be sufficient proof that these tools are relevant in real-world scenarios.

Architecture of the more popular tools is looked at by looking at the documentation and the behaviour of the tools in RQ 1.3. In RQ 1.4 a short literature review on completely automated testing tools for Android are looked at. Initially this was another possible direction for the empirical research, but since it seemed that many of the tools were not usable or were not available for download anywhere it was not continued.

To evaluate the tools identified in previous research question, four different categories were identified by the author with the help of the study by Nanz & Furia (2015) and by looking at the ISO 9126-1 standard for software quality characteristic. Especially the study by Nanz & Furia seemed to be similar enough to work as a basis for this comparison. There are a few Android UI testing tool specific comparisons that were identified, but they did not provide good guidelines for this. First is an old study where Android Instrumentation framework and Positron Framework are compared (Kropp & Morales, 2010). However, in this study, no rigorous comparison seems to have been done, it merely discusses the differences between the tools’ API, how they handle getting an UI element and the difference of how high-level tools the tools are. In another study, a master’s thesis, the tools were compared by criterion such as “can we use delays in test code” and “is
interaction with soft keyboard available” (Hökelekli, 2016). The categories however did not fit the benchmarking angle that was done in this study.

It seemed that carrying out an empirical study seemed to be the best way to approach this, so one was carried out by developing a suite of tests for several applications. This is discussed further in chapter 3.3. Results of these measurements are presented in chapter 4 and these are discussed in chapter 5.

3.2 Research questions

The research questions answered in this thesis are as follows:

RQ 1: What tools are available for GUI test automation of Android applications?
RQ 1.1 What tools are available for manual script generation for Android applications?
RQ 1.2 How do these tools order in popularity?
RQ 1.3 How do these popular tools differ in architecture?
RQ 1.4 What tools are available for automatically testing Android applications?

RQ 2: How do the most popular tools differ in performance?

RQ 2.1 What points of weakness do these tools have?
RQ 2.2 What are the differences in execution time?
RQ 2.3 How is the maintainability of the test scripts different between the tools? RQ 2.4 What are the differences between reliability of the tools?
In Figure 6 the flow of information through the research questions in this thesis can be seen. Starting from the top, the tools that are available are identified through blogs and scientific literature. A popularity filtering is done for the tools to limit the number of tools that will be inspected closer. At this point, Tau is added to the tools to be considered. A short review on the tools’ architecture and functionality is done at this point, and the analysis is continued with an empirical review of the tools.

The drive point behind this research is to help practitioners in the industry to better choose a tool suitable for their Android UI testing needs. Scientifically the point of this study is to a) look at what tools are available for Android UI testing, b) produce research that is more near to “real-world” situations rather than research being done in labs, as it was one of the critiques aimed at mobile application testing papers by Zein et al. (2016).

3.2.1 Research Question 1

Research question 1 was discussed in chapter 2. Goal of this research question was to answer what tools are overall available for use in UI testing for Android applications, and how the tools differ from each other. A small look at the popularity of the tools is also conducted, and based on this the most popular tools’ structure and behaviour is looked at more closely.

Research question 1.1 deals with what tools are overall available for GUI testing of Android applications. Few scientific studies were found for answering this question, mostly offhand mentions in different scientific articles discussed what tools are available but no study mapping out them was found. The question was then answered primarily by
searching for blogs related to Android UI testing and listing out what tools are mentioned there. The tools were then looked at to weed out tools that are not related strictly to GUI testing.

When this research was initially begun, it was not clear what tools are overall available. There seemed to much research focused on tools that automatically build a suite of tests for an application by analysing the source code and the application while it’s running. However, when these tools were looked at, many of them did not work with current versions of Android or they were not available. The research regarding this is under RQ 1.4 and is presented shortly in chapter 2.6. Focus was then shifted to looking at the tools where a tester writes a test script that is executed against the AUT, and this is discussed under questions 1.1-1.3.

RQ 1.2 was then answered by ranking the tools identified in RQ 1.1. Again, since no scientific sources for industry popularity of these tools could be found, more informal means were utilized. The popularity was looked at with two tools, Google’s Trends service and a forum focused on programming-related questions, Stack overflow. By using two different sources the results should be more reliable, especially if they give similar results.

RQ 1.3 was conducted to dig deeper to the tools that were identified to be popular in RQ 1.2. The official documentation for the tools was looked at to try to figure out how the tools work and if they differ from each other.

As mentioned earlier, RQ 1.4 was also explored initially. A look at scientific literature listing different automated testing tools for Android applications are explored under this research question, and their apparent situation is discussed shortly. This could have been another avenue of empirical research in this study, but because of the difficulties faced with most of the tools identified in here it was not continued.

### 3.2.2 Research Question 2

Regarding RQ 2.1, different weaknesses that were discovered by the author during developing the test suites for the three applications are discussed under this research question. This question is researched qualitatively since no absolute metrics can be established for the issues researched under this question. The issue is discussed through the author’s findings and experiences while developing the test suites.

Results are discussed in chapters 4.1, and a condensed Table 13 at that chapter is presented to show the weaknesses discovered and severity of the problem with each testing tool, as experienced by the author.

Rest of the research questions had quantitative metrics used. In RQ 2.2, execution times for the testing tools were measured by running the test suites multiple times in succession. This was done in two ways: first for the test suites that could be run by Gradle, Android’s Build tool, a PowerShell script was built that would run the tests and save the run results. Second for Tau some slight measurement code was added to the test code to gather run times of the tests, the test suites themselves could be run multiple times through the Tau software.

The reason execution time is included to be studied is that it is a well-established metric in software engineering world. In a study by Nanz & Furia (2015) programming
languages were compared, and execution speed was one of the categories. It is also mentioned as one of the six characteristics on quality of software in ISO 9126-1 ("ISO 9126 software quality characteristics" n.d.), so it was decided to include as well. Analysis of these run times was then carried out and is presented in chapter 4.5, where boxplots of the execution times, medians and other figures are presented for each application and tool.

Regarding RQ 2.3, differences in maintainability of the test suites between the tools was inspected. This was done through lines of code and the number of waits required by the testing tool. When measuring maintainability of a piece of software, lines of code seems to still be a good indicator (Sjøberg, Yamashita, Anda, Mockus, & Dybå, 2013). Also, according to McConnell (2004), programming languages do not have an effect on the defect density, as reported by Nanz & Furia (2015). It would seem reasonable that this would hold true to test script code as well as application logic.

Number of waits was identified while the test suites were being developed as being a source of problems for the author and it seemed to vary between the tools. Including waiting method has been identified to be one of the main ways of reducing test flakiness (Luo, Hariri, Eloussi, & Marinov, 2014). On the other hand, one of the main points mentioned for the tool Espresso is that it does not require these wait method calls in the test code ("Testing UI for a single app" n.d.). It then seemed to be interesting to investigate how many of these method calls each tool require making the suite of tests as non-flaky as possible.

Another approach to this could be to compare the number of classes and the test suites require, but this would not make sense in this setting since the test suites were constructed in a very similar way. The number of classes in the test suites is almost the same. Alternatively, the number of method calls the test code requires could be another measurement, but it is not clear if this would provide any more information than the lines of code does. This research question is further expanded in chapter 4.3.

Finally, regarding RQ 2.4 the reliability of the testing tools in this study was studied. It was measured through how many of the test runs with each tool fail. The same test runs that were used to measure the speed of the tools (RQ 2.1) were used to measure the number of failures that each tool faced. The percentages of failed tests and other figures are presented in chapter 4.4. In theory, none of the test runs should fail if the test suite is developed correctly, but in practice there are some tests that fail only some of the time.

This issue can be related to the ISO 9126 standard regarding reliability. Reliability is a desired aspect of a software product, and regarding a testing tool inspecting how often the tool gives false results could be a sign of the tool’s reliability. If there are many false negatives when the tests are run (tests that fail for some other reason than because there is a fault in the AUT) it can become cumbersome to see if there is something wrong with the source code of the application or if there is a problem with the testing tool. Luo et al. (2014) report that in a certain software product at Google 4.6% of the test failures are accounted for by a flaky test. It then seems that reliability of the testing tools is something that large software companies struggle with as well. Finally, false negatives can be expensive if engineers must use much of time to investigate these failures (Herzig & Nagappan, 2014).
3.3 Empirical research

The empirical research was carried out in a few steps. First several suitable applications for which to build a suite of UI tests were identified. Second, a suite of tests with each of the tools was developed for each of the applications. Finally, the test suites and several runs of these test suites against the applications were analysed. Choosing the applications is discussed further in chapter 3.3.1, developing the set of tests is discussed in chapter 3.3.2 and running the test sets is discussed in chapter 3.3.3.

The comparison part of this study is carried out in a somewhat similar manner as a programming language study was carried out by Nanz & Furia (2015). However, they researched differences between programming languages and in this study UI testing tools are benchmarked. In the study by Nanz & Furia they selected several tasks from a repository but in this study the tasks (or rather the test scripts) are developed by the author with the different tools.

First the author first used the application to see its’ main functions and then developed a suite of tests to exercise a portion of the app’s UI. Some guidance on how to develop the tests was acquired from App Quality Alliance testing guide ("App Quality Alliance", 2014). After this the same suite of tests was developed with every tool, and much care was taken that the tools exercise the UI in a very similar fashion: the same buttons are pressed and if possible searched for with the exact same parameters. The focus was on testing a single application in contrast to testing how multiple applications would work together, as this is a recommended approach when doing GUI testing (Zakharov, 2016). This approach would only cover one of the four categories (Components within your app only inside integration tests) that the Android Developer guidelines outline, shown in Figure 7.

![Figure 7](image-url) Figure 7. Four categories of tests for Android applications (Getting started with testing.n.d.)
Second, similarly as in the study by Nanz & Furia (2015), the tasks were run in a controlled environment. In this study, the test suites were run multiple times to lessen the impact of random changes that can happen on the device running the tests or on the Android device. This was done by creating a PowerShell script that would automatically run the same command-line command as many times as required. The script would then be left to run the command by itself for a time. Finally, the data of the test executions was gathered from the test run reports that were saved in the previous step, and those were analysed with R and Excel.

3.3.1 Applications chosen for evaluating the testing tools/libraries

It was required that the application is an open source one, for example Espresso and Robotium require that the tests are bundled with the application that is installed. Thus, the applications were searched from the F-droid service. F-droid offers several open source Android applications, and has been used as a source for applications in other research as well (Machiry et al., 2013).

The three applications chosen were Notes app by NoNonsenseApps¹, Amaze FileManager by Team Amaze² and Wikipedia Android app by Wikimedia Foundation³. All the applications were published as open source applications with licences granting the use of the work in this manner.

The applications were chosen based on if they seemed to have recent updates displayed in the Play store (updated in the last 6 months when the applications were chosen) and if they had a decent number of downloads (100 thousand or more) and a decent amount of ratings.

These criteria were put in place so we could be certain that the applications were at least somewhat established ones and which had a decent user base making them reasonable applications for which to develop a suite of tests. The requirement for the applications to have been updated recently was so that we could be sure the projects were not abandoned, this way it would be possible to have a conversation with the developers if need arose during developing the tests and so we could be sure there was some usage of the applications in the world. This all was done so that the setting where the tests would be developed would be closer to a real-world industry.

The number of users and votes was also considered to make sure the applications were in use in the real world and hopefully complex enough to be of some use to the users. This way we could be sure that the tests developed were closer to how people in real-world development situations would develop the tests.

All the applications chosen were classified as “simple” applications per AQUA criterion. This means that the applications can do networking activities, access location data can access keyboard, camera and such, can write their own data, and can read files. They however do not send SMS messages, do not add information to external services and do not write data to calendar, contacts or such. ("App Quality Alliance", 2014)

¹https://github.com/spacecowboy/NotePad
²https://github.com/arpitkh96/AmazeFileManager
³https://github.com/wikimedia/apps-android-wikipedia
Notes

Notes is “A notepad and to do app with both phone and tablet UI. Syncs with your existing Google account to keep your notes and tasks backed up in the cloud”. The application’s UI is straightforward with an ordinary Android application design including a Floating Action button, navigation drawer and text fields. The application stores its’ data in an SQLite database and uses shared preferences as many other applications do. It can do network communication through synchronizing the notes. The application also has the possibility to add reminders and due dates for tasks and notes, and allows grouping of notes and tasks.

The application is not bloated with features, and only is focused around creating notes, adding reminders or due dates to them and grouping them by task lists. Simple nature of this application makes it a good target for building simple UI tests. Synchronization of notes could bring added complexity to testing the application. Application’s source code is available in GitHub and it is developed mostly by one person. The application had some unit tests in place but no GUI tests so far.

Amaze

Amaze file manager is a native application that would replace the default file manager that comes with most versions of Android. It is used to navigate the file structure that the user has in his or her device’s internal and external storage space. It also automatically offers some groups of file types. The application naturally also provides the ability to create new files as one would expect to do when browsing his or her file system. The application also offers a possibility to connect to a networked drive. The application offers an ordinary Android application’s GUI, like Notes with a navigation drawer, number of lists and floating action buttons.

Amaze is also a straightforward application to test, since it features an ordinary UI in relation to author’s experience in Android apps. The application does have possibility for network communication through FTP servers, which could bring increased complexity to the testing activities, as well as the application handling file system of the application. This can prove to be a point of problems when the tests are run, since it is not enough to simply delete and reinstall the application to remove any data it might have created during the test, but it could be required to remove also files that have been created in the file system of the device. However, the UI of the application seems straightforward to navigate, so this application is a decent target for UI testing with the tools selected. The application’s source code is available in GitHub and seems to have two active developers. The app had no tests in place when developing of these GUI tests was started.

Wikipedia

The Wikipedia android application is a hybrid application, combining native Android GUI code with HTML and JavaScript that is fetched from a server. The application is meant to show the same articles as the Wikipedia website does, and offers capabilities to open articles in new tabs and to follow articles through links in the article’s body text. The application also offers a map to show articles of things that are nearby the user currently, using the devices’ GPS data to pinpoint the location of the device being used.
The application’s UI is mostly a native Android application’s UI, with navigation drawers and floating action buttons as well as lists, but it does also make heavy use larger cards and large images at the top of individual articles. Many native elements are used on the side with the article body, which should make this application a good target for the selected testing tools. The application however does offer quite a few different features so thoroughly testing these could be a big task.

Application also naturally requires data connectivity since the articles themselves are fetched from a server. This makes the application interesting in the sense that many Android applications fetch some data over a network and it is important for the testing tool to be able to handle this kind of situation.

The element that makes the application a so-called hybrid application is the WebView GUI widget. This, as discussed earlier, is an important part of the application since it shows the article itself that is of interest to a user. This can prove a point of problem for the testing tools that only target testing native Android apps, since the code run inside the WebView is HTML and JavaScript.

Application is developed by Wikimedia foundation as open-source project, and at least 13 people are listed on the development team for the application (Wikimedia apps/team.2017). The app seemed to have some unit tests and some instrumented tests in place when development of these test suites was started (it was not overly clear if they were UI tests or just unit tests meant to run on a real device), but not a comprehensible UI test suite was found. It seems the team had started to move towards implementing tests since the Espresso libraries were already imported in the build.gradle file of the project, but at that point they were not implemented yet.

### 3.3.2 Devising the test set

The tests were devised mostly depending on the knowledge of the researcher. Mostly the tests were devised to be functional tests. There was not found comprehensible guide on what things one should test when testing Android applications, so it was felt that figuring how the applications work and what they should do would be used as basis for developing the suite of tests for the application, much like a novice tester might start doing when starting out testing activities on his or her app.

A guide by App Quality Alliance was found that was used to give some guidance on what things should be tested (App Quality Alliance, 2014). This list of things to be tested was not a perfect match and needed to be modified heavily to suit the needs of doing purely automated GUI testing. As the list from AQUA is meant for testing in more general form and not as a list of what to test in GUI tests, it includes parts that are not relevant in this kind of testing or in other parts it did not seem to be feasible to test in a setting where the point of the testing is to see how tools can be used to write GUI tests. Test cases that were left out include cases such as “Long launch time”, “Memory card operation”, “application behaviour after forced close” or “Multiple screen sizes”.

Testing for incoming phone calls and incoming text messages was also decided to be left out, since it would require depending on external software or a testing harness, which is also not in the scope of this tool comparison. The testing libraries and tools do not themselves support these kinds of actions. Some of the test cases devised were inspired from by the guide, but were not explicitly mentioned in it, such as making sure onPause and onResume lifecycle activities in the AUT are called and making sure that some
elements that should be shown are indeed visible in the user interface. The Table 6 presents the categories chosen from AQUA to be used when developing the tests in this research.

**Table 6 AQuA tests chosen**

<table>
<thead>
<tr>
<th>Name of the test case</th>
<th>Description of the test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Screen repainting</td>
<td>Rotate the phone twice, see that the elements are still visible</td>
</tr>
<tr>
<td>Suspend / resume from main menu</td>
<td>Rotate the phone, make sure fields that were visible are still visible and hold correct information</td>
</tr>
<tr>
<td>Functionality sanity check</td>
<td>Tests about functional aspects of the application, make sure the application functions as it should (when adding a new note or file it is displayed in the UI). This is simplified from AQUA’s steps in the way that the offline-online testing is left out.</td>
</tr>
<tr>
<td>Scrolling in menus</td>
<td>Scroll up and down in navigation drawer and possibly other visible scrollable lists</td>
</tr>
<tr>
<td>Application stability</td>
<td>Implicitly verified with other tests (especially with functionality tests), if the application crashes at some point the test case will fail</td>
</tr>
</tbody>
</table>

Some of the cases also were modified to make more sense considering the AUT – for an example the test case “Suspend / resume from main menu” was simplified to a test case where the application is to be rotated to landscape mode and back to portrait mode (to simulate calling onPause of the application and onResume). In most of the tests resourceld was used to select the element instead of other selection methods, this is the recommended way in the UI testing guide by Google ("Testing UI for multiple apps" n.d.). The constructed test suites are available publicly on GitHub, see Table 7.

**Table 7 Location for test suites**

<table>
<thead>
<tr>
<th>Notes</th>
<th>Amaze</th>
<th>Wikipedia native</th>
<th>Wikipedia WebView</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robotium</td>
<td><a href="https://goo.gl/tPFqSP">https://goo.gl/tPFqSP</a></td>
<td><a href="https://goo.gl/nrZIfh">https://goo.gl/nrZIfh</a></td>
<td>-</td>
</tr>
<tr>
<td>Tau</td>
<td><a href="https://goo.gl/CHSGso">https://goo.gl/CHSGso</a></td>
<td><a href="https://goo.gl/IAJTEjr">https://goo.gl/IAJTEjr</a></td>
<td><a href="https://goo.gl/ALM-Im">https://goo.gl/ALM-Im</a></td>
</tr>
</tbody>
</table>

Test suites are outlined in the subchapters below. The tables only include a short description of the tests, not all the steps necessary steps for completing the test. Interested
parties are encouraged to investigate the test suites themselves, as they are available for inspection.

**Notes**

Since Notes is a straightforward app with barely any possibility for asynchronous events and not that many different features the test suite is simple as well. The tests that were designed are presented in Table 8 below.

**Table 8 Devised tests for Notes and their relation to AQuA**

<table>
<thead>
<tr>
<th>Name of the test</th>
<th>Description of the test</th>
<th>Corresponding AQuA test</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Create new task list</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T2</td>
<td>Create new note</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T3</td>
<td>Add note to a task list</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T4</td>
<td>Create note and add reminder</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T5</td>
<td>Create note, add due date, make sure it is visible</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T6</td>
<td>Create and delete a note</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T7</td>
<td>Create notes and order by date</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T8</td>
<td>Create task list and delete it</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T9</td>
<td>Clear done tasks</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T10</td>
<td>Add several notes, scroll to one outside view and open it</td>
<td>Scrolling in menus</td>
</tr>
<tr>
<td>T11</td>
<td>Add task list, rotate screen, check if task list visible</td>
<td>Screen repainting &amp; suspend /resume from main menu</td>
</tr>
<tr>
<td>T12</td>
<td>Add a note, rotate the screen and check if note is visible</td>
<td>Screen repainting &amp; suspend /resume from main menu</td>
</tr>
<tr>
<td>T13</td>
<td>Add task lists, scroll navigation drawer down</td>
<td>Scrolling in menus</td>
</tr>
</tbody>
</table>

From this test set 11 out of 13 tests were run. The T7 was left out of the final set because after a reasonable effort it seemed to be impossible to implement the tests with all the testing tools. Test T10 was left out because of the bug in the application, it proved to be impossible to scroll the visible list of notes since after a certain note the list would stop working correctly and would just start from the beginning in the middle of the list. More description about the T7 is in the findings chapter.

**Amaze File Manager**

Tests devised for the Amaze File Manager are in Table 9 below.
Table 9. Devised tests for Amaze and their relation to AQuA

<table>
<thead>
<tr>
<th>Name of the test</th>
<th>Description of the test</th>
<th>Corresponding AQuA test</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Add a new folder and delete it</td>
<td>Functional sanity check &amp; Scrolling in menus</td>
</tr>
<tr>
<td>T2</td>
<td>Add a new text file and delete it</td>
<td>Functional sanity check &amp; Scrolling in menus</td>
</tr>
<tr>
<td>T3</td>
<td>Copy a file to another folder</td>
<td>Functional sanity check &amp; Scrolling in menus</td>
</tr>
<tr>
<td>T4</td>
<td>Open two folders and swipe between them</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T5</td>
<td>Open a file multiple times and see if it is added to Quick Access</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T6</td>
<td>Test if “Set As Home” works</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T7</td>
<td>Test if grid view / list view works</td>
<td>Functional sanity check &amp; Scrolling in menus</td>
</tr>
<tr>
<td>T8</td>
<td>Add file to bookmarks</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T9</td>
<td>Edit a bookmark's name</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>T10</td>
<td>Rotate screen in main fragment, see if items are still visible</td>
<td>Screen repainting &amp; suspend /resume from main menu</td>
</tr>
<tr>
<td>T11</td>
<td>Rotate screen in drawer, see if elements are still visible</td>
<td>Screen repainting &amp; suspend /resume from main menu</td>
</tr>
</tbody>
</table>

From this test suite, all but one test was included to be run in the final test set, T5 had to be left out of the test suite that is ran because Robotium is not able to handle with the situation where the application would navigate outside of the AUT. More about this is in chapter 4.1.3.

Wikipedia

The Wikipedia application hosts a WebView component inside it, making it a so-called hybrid application, and this had to be considered when doing the testing activities with these tools. Initially it was planned that there would be one test suite built for this application as well, but it was soon realized that all the tools are not able to automate the WebView portion of the application reliably. After this was noticed the test suite was split into parts that only handle the native Android GUI and to tests that also manipulate the contents inside the WebView. User of course might not notice this difference herself, but it must be considered when doing automated GUI testing. The tests developed for the Android native portion of Wikipedia are presented in Table 10.
Table 10. Devised tests for Wikipedia native and their relation to AQuA

<table>
<thead>
<tr>
<th>Name of the test</th>
<th>Description of the test</th>
<th>Corresponding AQuA test</th>
</tr>
</thead>
<tbody>
<tr>
<td>NT1</td>
<td>Search for an article, make sure its title is shown on top of the screen</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>NT2</td>
<td>See that recent searches shows an article</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>NT3</td>
<td>Test changing language when searching</td>
<td>Functional sanity check &amp; Scrolling in menus</td>
</tr>
<tr>
<td>NT4</td>
<td>Test scrolling table of contents down and clicking a subheading, assert title is no longer visible</td>
<td>Functional sanity check &amp; Scrolling in menus</td>
</tr>
<tr>
<td>NT5</td>
<td>Test changing language in an article</td>
<td>Functional sanity check &amp; Scrolling in menus</td>
</tr>
<tr>
<td>NT6</td>
<td>Make multiple tabs, see that you can switch between them</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>NT7</td>
<td>Add an article to a reading list, see it is available and can be clicked (open article, click bookmark button near top picture)</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>NT8</td>
<td>Test history (open a few articles, see they are visible in history</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>NT9</td>
<td>Test opening table of contents and rotating phone</td>
<td>Screen repainting &amp; suspend/resume from main menu</td>
</tr>
</tbody>
</table>

Below in Table 11 are the tests devised for the WebView portion of the Wikipedia application.

Table 11. Devised tests for WebView part of Wikipedia

<table>
<thead>
<tr>
<th>Name of the test</th>
<th>Description of the test</th>
<th>Corresponding AQuA test</th>
</tr>
</thead>
<tbody>
<tr>
<td>WT1</td>
<td>Test that table of contents shows correct subtitles</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>WT2</td>
<td>Test following a link by full text</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>WT3</td>
<td>Test following a link by partial text</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>WT4</td>
<td>Test expanding and closing references</td>
<td>Functional sanity check &amp; Scrolling in menus</td>
</tr>
<tr>
<td>WT5</td>
<td>Open an article in a new tab (click link - click overflow button - open in new tab)</td>
<td>Functional sanity check</td>
</tr>
<tr>
<td>WT6</td>
<td>Test opening tabs and rotating phone</td>
<td>Screen repainting &amp; suspend/resume from main menu</td>
</tr>
</tbody>
</table>
All the tests from both test suites were ran, but Robotium had to be left out of the comparison because of great problems with running the tests with this tool, more about this is in findings chapter.

3.3.3 Running the test set

The test suites for Appium, Espresso, Robotium and UiAutomator were run through Gradle, a build tool that is used by any Android project made with Android Studio. The choice was made because Gradle could be used from command line so the test suites could be run many times in succession, and because it could run both instrumentation-based tests as well as Appium tests and Gradle also offers a good html-based report and xml output of the test run making it easy to gather the results of the test runs.

The command line command to run the instrumented tests was “gradle connectedXYZDebugAndroidTest”. The command to run Appium tests through Gradle that was used is “gradle testXYZDebugUnitTest”. With both cases XYZ is the Gradle flavour of the application where the test suite is written in, for example “gradle connectedPlayDebugAndroidTest” in the case of Amaze File Manager tests.

The test suites were run multiple times by creating a PowerShell script that would take in as parameter the name of the test suite and the number of how many times the suite should be ran. Since Gradle generates automatically the html-based reports about the test run, it was possible to gather statistics such as run tests, failed tests, names of the failed tests and test execution times. Before every execution of the test script both the device on which the tests were run and the host device on which the script is run were rebooted to reduce the number of background services and applications running on either of the devices. On the host device, there was no other applications started while the test scripts were running, except for Appium where the Appium server was run on the same device where the test script execution was running. On a normal PC and phone that were used it is very difficult to make sure that there are no background programs, such as virus scans or OS updates, on the device, but hopefully those will not have a large effect on the results.

In the case of Wikipedia application both the test sets were run with multiple network situations. First was a situation where a residential high-speed cable connection was used, to which the test device was connected by Wi-Fi. Second situation was where a test device had a commercial pre-paid card and the device’s preferred network type was set to 4g.

The tests that were built but in the end, were not run were still included in the test suites. These are the T7 and T10 in Notes and T5 in Amaze File Manager, as mentioned earlier. In the case of Espresso, UiAutomator and Appium the tests were denoted with @Ignored annotation that Junit 4 supports, meaning that they should not be ran at all. In the case of Robotium the tests’ names were changed so that JUnit3 will not run them since annotations are not supported with it. Finally, in the case of Tau only the tests that were supposed to be run were selected on the tool.

3.3.4 Test run environment

The test run environment is as following, the tests were run on a single host machine and on two separate testing devices, both with the same specifications and model. When the tests were run, the host device on which the tests were run from and the android phone where the AUT was running were rebooted after every test run. The tests were run in
batches of 10-40 tests at a time through a PowerShell script that would run the same command-line command continuously many times to reduce the manual work required to run the test sets. The Wikipedia’s test suites were run in two different network situations, first was a commercial residential cable network (theoretical speed of 100/10 Mbit) used over a Wi-Fi connection and second was a commercial mobile 4G data connection with a promised speed of 5Mbit to 40Mbit. This was done to gather data for possible further analysis between network conditions, but which ultimately was not addressed in this thesis outside presenting the results in chapter 4.

The phone used for running the applications is a Nexus 5x, with Android 6.1.

PC specs:

Lenovo Thinkpad E460

- Windows 10 Pro (x64)
- Intel Core i7-6500U CPU @ 2.5ghz, 2 cores, 4 logical processors
- 8gb of DDR3 RAM
- 192 GB SSD (Samsung MZ7LF192HCGS-000L1)

Gradle version 2.1.0 was used to run the tests, and the following versions of the testing tools were used:

- Espresso version 2.2.2
- Appium version 1.5.3
- UiAutomator version 2.1.2
- Robotium version 5.6.1
- Tau version 1.2.1

The versions used were the newest versions available when the tests were started to be developed. Some of the tools had to be updated during the development to get the test suites finished, but otherwise the versions were not updated.
4. Findings

In this chapter, the findings of the study are discussed. At first a general glance at the tools is made. After this notes about difficulties and other points of worth mentioning regarding experiences at developing tests for the different apps is discussed. In chapter 4.2 the results research question 2.2 about execution times are presented, in chapter 4.3 RQ 2.3 and maintainability is addressed, in chapter 4.4 RQ 2.4 and reliability’s results are presented and finally in chapter 4.5 the tools are ranked with scores gathered from the previous chapters.

4.1 General findings and sources of difficulty for the tools (RQ 2.1)

General notes and points of difficulty are discussed in this chapter. These issues are ones that rose for the developer when he was developing the test suites. The points of problem are presented in Table 12, and the severity of these issues is discussed in chapter 4.1.3 and with Table 13. As can be seen in the table, Appium, Tau and UiAutomator performed the best in this category, with Tau taking the top spot. All of them had very similar scores regarding the general sources of difficulty. Espresso fared slightly worse and Robotium had clearly the most issues.

4.1.1 General points of problem

There are several sources of difficulty for the different tools, some tools can handle certain interactions with the AUT with ease where as some tools make a tester jump through different hoops to interact with a certain element on the screen. These notes are presented in the Table 12 below, and they are later referenced when inspecting how effective different tools were manipulating the UI of the different applications. Points are as follows:

D1: WebView was shortly explained in chapter 2.1. This issue was difficult since documentation for UiAutomator or Tau do not really tell if they can handle WebView elements. Robotium on the other hand made promises that it could manipulate the WebView UI element, but that seemed to not hold true with the Wikipedia application.

D2: Navigation drawers are UI elements used for displaying the top navigation options inside the application. All the applications in this study included them. Opening one can be done in many ways in all the applications, all have a dedicated button for opening it and user could also swipe from the side of the screen towards middle. Only Espresso and Robotium have a dedicated method call for opening the drawer, and in Robotium it seemed to not function. For the other tools one must emulate either a finger swipe action or one must push an UI widget to open a drawer, there is no dedicated method for opening it.

D3: Interacting with the elements inside the drawer can be tricky since on all tools one can’t designate that element should be searched inside another element, or that the tool might not understand if an UI element is visible on the screen to the user or if it is behind the drawer. In the case of Tau, it could at a times click in the wrong part of screen when it tried to click on an item behind the drawer leading to test failure.
## Table 12. Points of difficulties for the tools

<table>
<thead>
<tr>
<th>Points of difficulties for the tools</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1: WebView</td>
<td>Elements inside WebView are not handled in same way as “native” Android elements, all tools can’t handle them</td>
</tr>
<tr>
<td>D2: Opening navigation drawer</td>
<td>Navigation drawer must be opened before interacting with elements inside it – this is surprisingly difficult with some tools</td>
</tr>
<tr>
<td>D3: Interacting with navigation drawer items</td>
<td>Interacting with only items inside navigation drawer can be difficult, but it is necessary to do it at times</td>
</tr>
<tr>
<td>D4: Asynchronous UI updates</td>
<td>Some tasks user does can require a moment for the app to update the UI, such as copying a file or doing a slow network call</td>
</tr>
<tr>
<td>D5: Removing app preferences and database</td>
<td>Many tests require the app to be reset to a blank status – such as removing notes or making sure we start at home screen</td>
</tr>
<tr>
<td>D6: Permissions</td>
<td>In Android API level 23 asking for permission causes a popup window not part of AUT, which all testing tools can’t click on</td>
</tr>
<tr>
<td>D7: Searching in list</td>
<td>Handling lists differs by tools; some tools take a list and search from it while others require one to give very specific parameters.</td>
</tr>
<tr>
<td>D8: AUT opens another application</td>
<td>Because of Android’s security model, some of the tools can’t click on UI expect for the application for which the tests are built for</td>
</tr>
<tr>
<td>D9: Assert that element is not on screen</td>
<td>Asserting element is gone is needed at a times, some tools do not offer simple methods to assert this</td>
</tr>
</tbody>
</table>

**D4**: Actions like copying a file or waiting for a web page to load can take a while to finish. All tools except Espresso have methods for easily waiting for an element to appear on the screen, but since Espresso wants developer not to care of timing problems, it can be trickier to make it wait for an unexpected UI update.

**D5**: Clearing application’s status after a test seems like a normal task, but it is not well supported by most tools. Different ways to do this needed to be implemented by the test developer. It is possible that this would be a task that the environment driving the tests should handle this instead of the testing tools, but at least in this case it would have been useful if the tools could take care of it themselves.

**D6**: A surprising problem that can cause Robotium and Espresso to be unable to test certain parts of the application. Permissions are an important part of Android that decide if an application can for example access the microphone, send text messages, or get the location of the device (Requesting permissions.n.d.)

**D7**: Differences between the tools occurred, and particularly with Appium searching inside a list is difficult, since one must pass in a line of UiAutomator code as a string to
do this properly. Some tools did automatically start to scroll a list if they found it, while some other tools needed to be told that search inside a list for an element.

**D8:** Related to D6, some of the tools are not able to touch UI of an application besides AUT. Some applications might need something from an external application, such as sending an email or taking a picture, so these kinds of tests with these tools can be impossible to make.

**D9:** Making sure something is not visible on the screen seems like a normal thing to do, but for some reason not all tools have a straightforward way to do this. With some tools, all elements need to be iterated through and polled if they are the element that is to be asserted to be not visible on the screen.

Last on a more general note, Appium was perhaps the most cumbersome tool with which to start writing the tests, since the first way to select elements presented in the Appium’s documentation is by Xpath, which takes in a string with quite a few special characters that makes it somewhat difficult to type out repetitively. (About appium.n.d.) For a novice tester, it can be a rough start to UI testing.

Finding an UI element with Appium by Xpath using a variable “taskListName” and clicking it:

```java
driver.findElement(By.xpath("//[@text='taskListName']")).click();
```

Doing the same action with UiAutomator with the necessary wait method included:

```java
device.findObject(new UiSelector().text(taskListName)).clickAndWaitForNewWindow(LAUNCH_TIMEOUT);
```

This however is remedied by using other methods for locating the elements on the screen, such as searching for an element by ID or name or by constructing helper methods to avoid typing this. The UiAutomator’s method is easy to type if code autocomplete is used, functionality that many modern IDEs offer. Appium’s method however needs to be typed mostly by hand.

### 4.1.2 Issue severity per tool

The Table 12 presented in previous chapter is expanded with issue ratings. A rating from 0 to 4 is given to each issue for each tool, somewhat similarly as JIRA classifies severity levels for security issues, 4 being comparable to a critical issue and 1 being a low issue (Severity levels for security issues.n.d.). Rating of 0 is included to signify that the issue is not applicable for the tool. The scores are visible in Table 13 and justification for these scores are visible after the table. The general notes related to all the tools and all the applications is in Appendix 2, where a longer examination of these is done. The ratings are given based on how severe of an issue this was felt to be, along with if the problem could be circumvented somehow.
Table 13. Severity of the tool issues per each tool

<table>
<thead>
<tr>
<th></th>
<th>Appium</th>
<th>Espresso</th>
<th>Robotium</th>
<th>Tau</th>
<th>UiAutomator</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1: WebView</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>D2: Opening navigation drawer</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D3: Interacting with navigation drawer items</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>D4: Asynchronous UI updates</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>D5: Removing app preferences and database</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D6: Permissions</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D7: Searching in list</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D8: AUT opens another application</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D9: Assert that element is not on screen</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>8</strong></td>
<td><strong>12</strong></td>
<td><strong>19</strong></td>
<td><strong>7</strong></td>
<td><strong>8</strong></td>
</tr>
</tbody>
</table>

**D1** is how well the tool could handle the WebView component of Android. Both Appium and Espresso did well in this regard, while Tau, UiAutomator and Robotium had problems with it. Tau and UiAutomator could interact with some of the web URLs inside the WebView element, while Robotium could interact with it somehow but being quite slow in searching the text for elements. Robotium’s API showed methods that should be able to interact with the WebView but author could not get any interaction with those working, having to resort to simply searching an UI element with the text and clicking this. Robotium, UiAutomator and Tau were unable to interact with the JavaScript portion of the WebView.

For these reasons, Robotium was given the worst score, issue being critical with Robotium. Tau and UiAutomator were given a score of 3 since this was a big issue with them, but not as bad as with Robotium.

**D2** seems like a trivial thing but it proved to be a surprisingly normal problem when the tests were built. Espresso was in every situation very reliable and easy in opening the navigation drawers, while Robotium had the most problems. Espresso has a dedicated method where one only selects a navigation drawer and sets it open. Appium had some flakiness problems when opening the navigation drawer by emulating a swipe, but when a dedicated button was clicked it functioned well. Tau and UiAutomator exhibited similar flakiness problems when swiping, but less than Appium.

In Notes application with Robotium a custom swipe action needed to be done to open the navigation drawer since at the time it was thought it could not access the dedicated button
made to open it (the “hamburger icon”). It was however later discovered that a following method call could be used to access the button, despite its’ name (which most likely is related to “home” button being in upper left corner of the screen in old UI guidelines for Android):

```
solo.clickOnActionBarHomeButton()
```

With Amaze’s test suite, a swipe action was used to open the drawer instead of the dedicated button. For Robotium the swipe action proved to be too flaky to be used so the dedicated hamburger icon was used instead with the method call shown above.

All these things considered, Espresso, being problem free, was given a rating of 0, UiAutomator and Tau score of 1 with minor issues, Appium a score of 2 with some issues and Robotium a score of 3 because of the problems with both Notes and Amaze and a lack of usable easy way to open the drawer (there is a method in the API that should do this but in these cases, it did nothing).

**D3** Interacting with navigation drawer items had problems in Amaze and Wikipedia applications, with Notes there weren’t any problems. Appium and UiAutomator did not have problems with interacting on elements inside the navigation drawers. With Espresso, there was difficulties in clicking items the “recommended” way by specifying the list and polling it for a child element’s position. Instead it was necessary to just click an element containing a text inside the navigation drawer, this worked with these applications but it could be that it would be a problem in other situations, for example on test devices with small screen sizes.

Robotium had issues with searching for items inside a navigation drawer and it had to be done with much manual work, but after an issue was opened on GitHub a patch was issued and this problem fixed. However, with Robotium and Tau the main problem was that clicking elements inside the navigation drawer was flaky, especially in the case of the Amaze application where the tools could not differentiate between items inside the navigation drawer and items behind it.

For these reasons, Appium and UiAutomator were given a score of 0, Espresso a score of 1 because of annoying but problems which could be worked around and score of 2 for Robotium and Tau for medium-sized issue which required a lot of attention.

**D4** is about the UI updating at some time after an action has been done. These actions include loading the results of a search query in the Wikipedia application and copying a file in Amaze File Manager. Since all tools except for Espresso and Tau require the use of waiting methods in the test code all the time, these similar approaches could be used with the asynchronous UI updates as well. Tau provides an additional parameter that can be used while searching for an element so one can tell it to search for the element for x number of milliseconds, supplying this made the test suite reliable.

UiAutomator however was not always reliable with these waits, it could arbitrarily just not wait for the period that it was supposed to wait, leading to many different failures at both Amaze and Wikipedia apps. Espresso on the other hand has different problems. Since Espresso automatically waits for the UI of the app to stabilize before continuing, it does not provide these kinds of waiting methods at all. For this reason, an IdlingResources must be implemented in the test code and possibly on the source code of the application, leading to some tinkering work necessary at some points. For this reason, some understanding of the source code of the AUT might be necessary.
Based on this, Appium and Robotium get a score of 0, Tau a score of 1 since it was a small issue but one that was remedied after searching through the documentation, Espresso a score of 2 since it did require some modification of the source code to get the waits in place and a score of 3 for UiAutomator, since it was a cause of many test failures in the case of Wikipedia application.

**D5** Removing data saved in the application is an issue with the so-called instrumented testing tools (Espresso, Robotium, UiAutomator). They do not offer clean ways to clear the application’s data without adding additional code to the AUT to do this. This is of course not preferred, since it can require some knowledge about the source code of the AUT and programming skills to implement this. Preferences however were generally easy to remove inside the test code without modifications to AUT required.

On Appium and Tau this point was not an issue, since Appium offers a flag that is set when the testing is started and with Tau we can make an ADB call to remove application data. For this reason, the instrumented testing tools were given a severity score of 1, minor issue, and the other tools a score of 0.

**D6** Permissions are simple to handle for UiAutomator and Tau, UiAutomator can just wait for the known permission popup to appear and handle it, and Tau offers a simple “reactor” that pushes a certain button if a certain type of popup window, such as Android asking for permissions for an application, shows up. Issue score of 0 is given to these tools.

Appium had a small problem with the permissions, since the ordinary way of launching the application could not be used. If it was opened this way, Appium would wait for a time for the application to gain focus, which it never would since the permissions window is visible, and would then time out. However, if the application was launched as a user would do it, through the applications menu, it would be possible to push the permission button successfully. Appium is thus issued a score of 2, medium, since this is an issue but one that can be worked around.

However, for Espresso and Robotium the permissions were a deal breaker. It seems not to be possible to grant the permissions with these tools since the UI element asking for them does not belong to the AUT, so these tools can’t touch the pop up. The tools do not offer any other means of granting these permissions either. The permissions must be given before starting the test suite or in the case of Espresso the permissions could be granted through UiAutomator. For these reasons, Espresso and Robotium get an issue score of 4, critical.

**D7** Searching for elements in a list is simple with most tools. Usually, Robotium and Tau do not need anything special to be done, the tools will automatically scroll a list if they find one. UiAutomator needs to be pointed to a list specifically, but otherwise is simple to work with. Robotium however had the problem at Amaze where it would not properly search the list for an item in the case of a test where a grid view would be tested, which lead a workaround required. Espresso has a bit of a hassle since searching inside a “traditional” ListView widget and in the newer RecyclerView widget is done differently. For these reasons, Espresso and Robotium get severity scores of 1.

Appium however was problematic, since one must pass in UiAutomator code to an Appium method call to search inside the list, which is error-prone. For this reason, Appium gets an issue score of 3. The issue can be worked around, but it is quite bothersome to do for a simple issue of searching a list for an element.
**D8** For Appium, Tau and UiAutomator moving outside the AUT is not a problem. On the other hand, Robotium and Espresso are completely bound to operating inside only the AUT. Interacting with applications other than AUT is not possible at all. For Robotium a test doing this was a deal breaker, but with Espresso it was easy to mock out the attempt to launch another application and return a correct value that the application expects, at least in the case of Amaze. Since for Robotium it was not possible to make a work-around for this, it gets a score of 4, a critical issue. Espresso on the other hand can deal with this by mocking out the intent that would launch another application so it gets a score of 3, a medium issue.

**D9** With Espresso, Robotium, Tau and UiAutomator asserting if an element is on the screen is not too difficult. With Espresso one can build the search term in such a manner that it is expected that the item is not found. Tau has a simple method call for asserting if an item exists or not, and Robotium and UiAutomator just return a Boolean whether the item was found or not. Surprisingly, Appium does not seem to offer a clean method call for asserting if an item is visible on the screen or not. The method raises an error if an item with the identifier is not found leading to a test crash if the error is not caught. For this reason, it is necessary to unintuitively select all the items on the screen with certain criterion and make sure that the returned array is empty. For this reason, it gets an issue score of 1, a small issue.

### 4.2 Execution times (RQ 2.2)

In this chapter, the execution times of the test suites are presented. In general, Espresso proved to be significantly faster than the other tools in driving the UI of the applications and Appium was slowest in all the cases, but how much slower varied between the applications. 45 successful test suite runs were gathered for each tool and application, but the numbers change a bit between the different run sets. A successful run means in this case a run that does not include any failed tests. This was done to ensure that there are no failed tests that could affect the run time of the test suite.

The figures presented later in this chapter have tool names shortened: first letter is the letter of the testing tool A (Appium), E (Espresso), R (Robotium), T (Tau) or U (UiAutomator). Second letter is the name of the application: N (Notes), A (Amaze File Manager) or W (Wikipedia). In the names of the Wikipedia measurements there are also two more letters, first one being the portion of the application that is tested, either N (Native) or W (WebView) and final letter represents network type, either W (Wi-Fi) or 4 (4G).

First a look at the combined results is done, and the results for tools on individual apps are visible after this. While looking at the combined results, the run times are compared by adding together the medians of the execution times of the tools for the individual applications. The results are split into two parts, in the first two columns the median run times of Notes and Amaze apps are presented, in the third and fourth columns the Wikipedia app with native portion is presented. This is since Robotium was not able to be used with Wikipedia but it was felt that it could be interesting to show this case as well. Later in this chapter the execution times of test suites for the three applications are shown. For every application, first a table listing out key figures of the data are shown, and afterwards follows a boxplot of the execution times. The tables in the following chapters contain key values of the execution times of the tools for each application. The execution time presented is the execution time of one whole test suite. Besides mean, median and standard deviation, Cohen’s D was calculated to show the difference between
the means of the run times, although since the standard deviations of the results are so small that the positions of the execution times are clear. In addition to this, the last column shows percentage differences in the run times.

**Table 14.** Combined mean execution times from Notes and Amaze File Manager

<table>
<thead>
<tr>
<th></th>
<th>Total median Notes &amp; Amaze run time (seconds)</th>
<th>% median of the fastest (Notes &amp; Amaze)</th>
<th>Total median Notes, Amaze &amp; Wikipedia Native (Wi-Fi) run time (seconds)</th>
<th>% median of the fastest (all apps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appium</td>
<td>886.79</td>
<td>493%</td>
<td>1457.42</td>
<td>538%</td>
</tr>
<tr>
<td>Espresso</td>
<td>179.72</td>
<td>100%</td>
<td>270.99</td>
<td>100%</td>
</tr>
<tr>
<td>Robotium</td>
<td>488.73</td>
<td>272%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Tau</td>
<td>598.5</td>
<td>333%</td>
<td>913.5</td>
<td>337%</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>536.04</td>
<td>298%</td>
<td>782.79</td>
<td>289%</td>
</tr>
</tbody>
</table>

In the Table 14 above the results can be seen clearly. Espresso is the fastest of the tools by two to three times, while Appium is the slowest being around five times slower than Espresso. Robotium is the second fastest with UiAutomator being only slightly behind Robotium, and Tau on fourth position being only a bit behind UiAutomator. With Wikipedia’s native portion considered Appium’s slowness is highlighted more and UiAutomator is slightly closer to Espresso.

The results are similar through every application, only the second, third and fourth place changing hands and the execution times ranging to being from two to three times slower than Espresso is for these three tools, and Appium ranging from nearly four to nearly six times slower than Espresso.

**Table 15.** Comparison on execution times in the Notes application

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Median</th>
<th>Standard deviation</th>
<th>Cohen’s D</th>
<th>% larger median than fastest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appium</td>
<td>506.32</td>
<td>500.32</td>
<td>15.11</td>
<td>34.00</td>
<td>636%</td>
</tr>
<tr>
<td>Espresso</td>
<td>78.76</td>
<td>78.64</td>
<td>0.60</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>Robotium</td>
<td>171.02</td>
<td>171.00</td>
<td>0.19</td>
<td>208.61</td>
<td>217%</td>
</tr>
<tr>
<td>Tau</td>
<td>300.12</td>
<td>296.5</td>
<td>5.58</td>
<td>54.23</td>
<td>377%</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>296.17</td>
<td>301.31</td>
<td>3.39</td>
<td>92.73</td>
<td>383%</td>
</tr>
</tbody>
</table>

For the Notes application, presented in Table 15, the fastest tool was clearly Espresso. On a second place was also clearly Robotium, with over a double the execution time of Espresso. On third and fourth place were Tau and UiAutomator, both close in the execution time to each other but compared to Espresso both had nearly four times the execution time. On the last place is Appium, with a big gap between it and UiAutomator and Tau, and with over six times the execution time of the fastest tool.

Cohen’s D is large in every situation, much over the “huge” (2.0) category of effect sizes outlined for it (Sawilowsky, 2009). This is especially visible with Robotium, where the
standard deviation is such a small number when compared to the median execution time that it is very clear that the tool in question is the reason for the changes in execution time.

Figure 8. Boxplots of Notes Execution times

Figure 8 shows clearly how little variation there is in the execution times when compared to the differences that are visible between the tools. However, Tau does have good and bad runs which, some positioning quite far from the 50th percentile of the execution times. Appium similarly has some execution times that seem to be at least 10% off the main group.
Table 16. Comparison on execution times in the Amaze File Manager application

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Median</th>
<th>Standard deviation</th>
<th>Cohen’s D estimate</th>
<th>% larger median than fastest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appium</td>
<td>384.45</td>
<td>386.47</td>
<td>10.01</td>
<td>40.43</td>
<td>382%</td>
</tr>
<tr>
<td>Espresso</td>
<td>101.09</td>
<td>101.08</td>
<td>0.35</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>Robotium</td>
<td>301.58</td>
<td>317.73</td>
<td>0.31</td>
<td>657.94</td>
<td>314%</td>
</tr>
<tr>
<td>Tau</td>
<td>317.71</td>
<td>302.00</td>
<td>6.10</td>
<td>46.21</td>
<td>299%</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>234.99</td>
<td>234.73</td>
<td>3.54</td>
<td>52.27</td>
<td>232%</td>
</tr>
</tbody>
</table>

As seen in Table 16, results are similar as with Notes application, the winner and loser of execution time is clear: Espresso is the fastest of the tools and Appium is the slowest. However, on second place this time is UiAutomator with a bit over double the execution time of Espresso, while Robotium and Tau are close to each other, both taking three times as much time as Espresso to run through the suite of tests. On last position is again Appium, though this time with only four times the execution time of Espresso.
Figure 9 again presents the results of the execution times in a boxplot. Interesting about the Figure 9 is that there are quite a few outliers for both Appium and Tau, and that many of those outliers are below the 50th quartile. This is like results of Notes in Figure 8, so it seems that it is a trend with these tools, though it seems odd that some of the test runs are that much faster than the rest. Espresso and Robotium are again very tightly packed, while UiAutomator has a bit of variation.

Next the Wikipedia application’s results for the native portion of the app are visible in Table 17 and Table 18 with different network speeds used. Results for the Wikipedia application is split into the two different test sets, as was discussed in chapter 3. For both sets the different network configurations are also presented.
Table 17. Comparison on execution times in the native portion of Wikipedia application using Wi-Fi network

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Median</th>
<th>Standard deviation</th>
<th>Cohen’s D estimate</th>
<th>% larger median than fastest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appium</td>
<td>569.10</td>
<td>570.63</td>
<td>15.16</td>
<td>45.02</td>
<td>625%</td>
</tr>
<tr>
<td>Espresso</td>
<td>91.49</td>
<td>91.27</td>
<td>1.55</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>Tau</td>
<td>312.67</td>
<td>315.00</td>
<td>10.70</td>
<td>12.78</td>
<td>345%</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>244.65</td>
<td>246.75</td>
<td>15.82</td>
<td>28.05</td>
<td>270%</td>
</tr>
</tbody>
</table>

As presented in Table 17 and Table 18, Espresso being fastest seems again clear with Cohen’s D estimates being in the huge factor and differences being large. Tau and UiAutomator have a large difference however, UiAutomator being nearly 70-80 seconds faster than Tau.

Table 18. Comparison on execution times in the native portion of Wikipedia application using 4G network

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Median</th>
<th>Standard deviation</th>
<th>Cohen’s D estimate</th>
<th>% larger median than fastest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appium</td>
<td>591.62</td>
<td>591.04</td>
<td>16.40</td>
<td>41.91</td>
<td>531%</td>
</tr>
<tr>
<td>Espresso</td>
<td>111.70</td>
<td>111.37</td>
<td>4.03</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>Tau</td>
<td>335.12</td>
<td>319.5</td>
<td>33.5</td>
<td>8.80</td>
<td>287%</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>241.87</td>
<td>244.77</td>
<td>15.02</td>
<td>36.59</td>
<td>220%</td>
</tr>
</tbody>
</table>

Considering both network speeds, Appium is again clearly the slowest of the tools, taking five to six times more time than Espresso to run the suite. Tau and UiAutomator had a large difference, but it is similar as the difference in Wi-Fi execution times, UiAutomator taking two to almost three times as much time as Espresso to run the suite of tests and Tau taking three to three and a half times as long.

Unsurprisingly the slower and more unreliable network conditions do introduce some changes in the time the tools take to run the test set. The other tools in relation to Espresso do not take as long to run the test set as in the Wi-Fi network case. Interestingly, UiAutomator’s runs were faster in the 4G network case than they were in the Wi-Fi
network situation. Tau experienced quite a lot of deviation in the execution times using 4g network, which most likely are related to changes in the network speed.

In Figure 10 above the large standard deviation of Tau when using 4g network can be seen clearly, much larger than that of any other tool. Since the deviation is much larger for 4g network than it is for Wi-Fi network, it seems reasonable to conclude that it is because of the changes in the network speed. Tau had some variation on the other applications, but nothing comparable to this. It seems the more unreliable 4g network is the reason for this. Following this, the figures for the WebView portion of the application are presented below in Table 19, again with different network speeds.
Table 19. Execution times in the WebView portion of Amaze Wikipedia application using Wi-Fi network

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Median</th>
<th>Standard deviation</th>
<th>Cohen’s D estimate</th>
<th>% larger median than fastest</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Wi-Fi connection</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Appium</td>
<td>241.81</td>
<td>239.85</td>
<td>7.57</td>
<td>33.25</td>
<td>551%</td>
</tr>
<tr>
<td>Espresso</td>
<td>44.20</td>
<td>43.50</td>
<td>3.46</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td><strong>4g connection</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Appium</td>
<td>244.38</td>
<td>248.26</td>
<td>10.78</td>
<td>24.65</td>
<td>589%</td>
</tr>
<tr>
<td>Espresso</td>
<td>42.12</td>
<td>42.11</td>
<td>0.93</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

As seen in Table 19, the results follow the pattern established before, Espresso being from five and a half to almost six times quicker than Appium in executing the set of tests. Cohen’s D is also much above the value given for Huge by Sawilowsky (2009), giving a strong indication that the tool is the cause for this difference. No much changes are seen in the different network conditions.
Same trends that have been established before are visible in Figure 11. Some outliers are however visible in Espresso’s execution times, but when compared to Appium the results are tightly packed.

Figure 11. Boxplots of the WebView Wikipedia execution times using Wi-Fi and 4g connection
Figure 12. Combined execution times
As the last item is the boxplots for the combined results of all applications and test suites, visible in Figure 12. Three distinct groups can be seen from the figure: Appium (without the Wikipedia’s WebView test suite which is much shorter than the other test sets), Espresso and the rest. Robotium’s execution time comes close to Espresso’s with Notes, but with Amaze it is clearly higher. One box of Appium execution times is near the group Robotium-Tau-UiAutomator because the Wikipedia’s WebView test suite (6 tests) was such a small one that running it takes much less time than the other test suites (13 for Notes, 11 for Amaze). The 4g network configuration for the Wikipedia application was omitted from the table since the results are not that much different from the Wi-Fi network results.

4.3 Maintainability (RQ 2.3)

Maintainability of the code can mean different things to different people, but in this study, it was measured by the traditional way of calculating the lines of code that the software (or in this case the test suite for the software) requires and how many wait methods needed to be used to develop the test suite. This way was complemented with a more domain-specific method of calculating the number of different wait methods that the test suite requires to do its’ job. Most of the tools need these waits and seemed to the author that they can indeed diminish the readability – and thus maintainability – of the test code.

All the test suites included a utility class that includes many of the common actions that are done while the test suite is built. This was done since it seemed to the author to be a reasonable way to approach building the tests – retyping and copying the same code all over the place is in general not a good idea in software development. The utility classes were attempted to keep as similar as possible, but there are some differences between tools since with some tools some actions are very simple to accomplish where other tools might require many lines of code to accomplish the same thing. This could affect the lines of code required by each tool and the number of waits that are needed, since these helper methods can also include waits and these utility methods can be used throughout the test suite in many places. Inside each application there are however many same helper methods that are the same with each tool, such as ones for opening and closing a drawer or for adding a new file in the case of the Amaze app.

In the similar vein a base test class was created for all test suites with each tool, since the identical setup and every test needs some basic variables. An exception to this is Tau, since it does not really have any lengthy set-up code and since Python did not really seem like a language where this would be beneficial. The number of wait methods means individual methods that are used to explicitly wait for an element to appear on the screen before continuing the test execution, example of this can be seen in Figure 13.

4.3.1 Lines of Code

Comparing the lines of code (LOC) required by the test suite can be interesting since the task they are trying to accomplish is the same. The same steps are repeated with every tool in every test and since the tests were attempted to make as identical to each other as possible, comparing the lines of code that are needed to achieve the same functionality can be interesting. Since some specific actions can be easier to achieve with some tools than they can with other tools, the whole test suites were evaluated to diminish these
random occurrences. In the same vein, the total lines of code required by each tool are reported to furthermore diminish this issue. The lines of code were calculated from the classes the author wrote himself to do the testing activities, all the setting up and finishing activities related to each tool are included. Since all the tests were written by one person in the span of 3 months so the coding style should be very similar between the test suites and tools. The results of this are presented in Table 20. The figures presented on the table are LOC amount with and without import statements. The value inside parentheses is LOC with the import statements included.

It would seem reasonable to leave out the import statements from the lines of code comparison, since these lines are most likely not ones that one would be inspecting when he or she was updating or changing the test code. They would have no effect on how maintainable the code is in this manner. When the tests were developed, the author did not manually even modify the import statements since the IDE used handled adding and removing them automatically. Android Studio also automatically folds away the imports from view, putting even less focus on modifying them when looking at code. The lines of code with and without imports are reported and the reader can consider which would be preferred.

Table 20. Lines of code required by each tool to build the test suite, LoC with imports inside parentheses

<table>
<thead>
<tr>
<th>ockey</th>
<th>Notes</th>
<th>Amaze</th>
<th>Subtotal</th>
<th>Wikipedia native</th>
<th>Total</th>
<th>Wikipedia WebView</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appium</td>
<td>395 (476)</td>
<td>498 (580)</td>
<td>893 (1056)</td>
<td>328 (388)</td>
<td>1221(1444)</td>
<td>269 (313)</td>
</tr>
<tr>
<td>Espresso</td>
<td>397 (586)</td>
<td>488 (682)</td>
<td>885 (1268)</td>
<td>379 (537)</td>
<td>1264(1805)</td>
<td>284 (396)</td>
</tr>
<tr>
<td>Robotium</td>
<td>529 (626)</td>
<td>382 (436)</td>
<td>911 (1062)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Tau</td>
<td>336 (348)</td>
<td>372 (382)</td>
<td>708 (730)</td>
<td>284 (293)</td>
<td>992(1023)</td>
<td>-</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>387 (487)</td>
<td>477 (603)</td>
<td>864 (1090)</td>
<td>372 (470)</td>
<td>1236(1560)</td>
<td>-</td>
</tr>
</tbody>
</table>

This comparison can have its’ problems, since while 4 out of the 5 tools are using Java as the test implementation language, while Tau is using Python. However, it seems that the defect density between languages does not differ so this comparison should be valid. (Nanz & Furia, 2015)

The lines of code were calculated with a tool known as “cloc” (calculate lines of code). The program was not given any additional parameters besides the path of the folder where the test code resides. The tool reports lines of actual code and separates comment and empty lines. Before the lines of code were calculated, Android Studio’s “optimize imports” function was used, which will remove unused and duplicate import statements from the class file.

The test suites had the same number of files since they were constructed in a very similar manner. Tests logically related to each other were in the same classes, except for the case of Notes where every test was in its’ own class file. Exception to this is Espresso, where

1https://github.com/AlDanial/cloc
each of the test suites had one more class since the action required to change orientation of the device was not a simple method call as it was with other test suites, and a 3rd party class to do this was used instead. The test suite for Amaze also includes for every tool the code required to capture a screenshot of the device when a test fails, so this is included in the lines of code for every tool.

### 4.3.2 Number of waits

The number of waits required by the test suite is not a measurement that would be traditionally used as a metric for maintainability of the code, but it was included in here since with all the tools it proved to be a point that had to be addressed in some way. At least in author’s opinion the code was more difficult to write when one needed to consider all the time if a wait would or would not be needed for each action taken in the test code. Many waits also needed to be added afterwards the test suite was running to completion to reduce the flakiness of the test suite. The number of waits were calculated from the code files written by the author for the project, and the calculations include every method call that has some manner of wait command in it. These commands can come in different shapes with different tools, as is visible in Figure 13. The number of waits required by the tools is visible in Table 21 below.

**Table 21. Number of waits required for test suite per app**

<table>
<thead>
<tr>
<th></th>
<th>Notes</th>
<th>Amaze</th>
<th>Subtotal</th>
<th>Wikipedia native</th>
<th>Total</th>
<th>Wikipedia WebView</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appium</td>
<td>14</td>
<td>5</td>
<td>19</td>
<td>19</td>
<td>38</td>
<td>17</td>
</tr>
<tr>
<td>Espresso</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Robotium</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Tau</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>15</td>
<td>17</td>
<td>-</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>23</td>
<td>40</td>
<td>63</td>
<td>37</td>
<td>100</td>
<td>-</td>
</tr>
</tbody>
</table>

As can be seen, Tau requires very few of these method calls while UiAutomator requires many of them. In the case of Espresso, the code related to waiting is either not needed at all, or it is done differently than with other tools. Like in earlier chapters, the results of Notes and Amaze are presented separately from the Wikipedia, since Robotium could not automate that application.

In Figure 13 below one can see how other tools handle waiting for a certain element on the application. In Espresso’s case one registers a so-called idlingResource for Espresso and it checks this resource automatically at every step before it continues the test execution. This resource is then updated in the source code of the actual application, hence possibly requiring some modification of the source code of the application. This idlingResource is not required with every application, but in the case of the Wikipedia app it was needed to be implemented because of the network calls that are made when articles are searched and opened. In the case of Amaze some small updates to the source code to implement this were also necessary to be added.
As can be seen in the Figure 13, Tau’s way of waiting is quite easy to read and understand, since the waiting is simply done as an additional parameter to the test function call. Robotium is as well quite easy to read and understand, although it should be noted that for Robotium one can’t combine waiting for an element and then interacting with it, one needs to first wait for it and then do something with it with two separate method calls, unlike can be done with the other tools.

With the rest of the tools one has different methods for waiting for an element and for selecting them, although in the case of Appium and UiAutomator searching an element and waiting for it are similar, same kind of selector is given to the function when an element is searched for (By.text and By.xpath in the example in Figure 13). Different from UiAutomator, Appium requires a specific WebDriverWait object to be initialized before one can wait for elements, where one would pass in the duration that this object uses to wait for the elements.

### 4.4 Reliability (RQ 2.4)

Reliability in this study means how few of the tests fail without a good reason. The combined results of all applications’ test suite runs are presented below, and individual application failures are presented in further subchapters. Since the test suites constructed do not contain tests that fail on purpose, at least in the current versions of the applications, in theory all the tests should pass every time the test suite is run. However, this is not the case as can be seen in the Table 22 below and the following subchapters.

The aim was to get at least around 45 successful test runs with each configuration. Since the test suites were run in batches, the results contain slightly different amounts of test suite runs, since it was impossible to know how many of the test suites would fail.

The Table 22 shows how many individual tests in total failed in the test set runs. Table 23 shows how many of the test suite runs had at least one test that failed, leading to a failed test run in total. Since Robotium was unable to automate reliably the Wikipedia application, subtotals are again shown separately for Notes and Amaze, previous two with Wikipedia native included and finally the previous two with Wikipedia’s WebView tests included.
### Table 22. Number of failed tests per tool per app

<table>
<thead>
<tr>
<th>Tool</th>
<th>Notes</th>
<th>Amaze</th>
<th>Subtotal</th>
<th>Wikipedia native (Wi-Fi)</th>
<th>Total</th>
<th>Wikipedia WebView (Wi-Fi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appium</td>
<td>0/650 (0%)</td>
<td>8/605 (1.32%)</td>
<td>8/1255 (0.64%)</td>
<td>76/900 (8.444%)</td>
<td>84/2155 (3.90%)</td>
<td>33/474 (6.96%)</td>
</tr>
<tr>
<td>Espresso</td>
<td>0/650 (0%)</td>
<td>1/550 (0.18%)</td>
<td>1/1200 (0.08%)</td>
<td>0/450 (0%)</td>
<td>1/1650 (0.06%)</td>
<td>4/300 (1.33%)</td>
</tr>
<tr>
<td>Robotium</td>
<td>4/715 (0.56%)</td>
<td>0/550 (0%)</td>
<td>4/1265 (0.32%)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Tau</td>
<td>0/676 (0%)</td>
<td>0/550 (0%)</td>
<td>0/1226 (0%)</td>
<td>11/612 (1.80%)</td>
<td>11/1838 (0.60%)</td>
<td>-</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>14/780 (1.80%)</td>
<td>12/715 (1.68%)</td>
<td>26/1495 (1.74%)</td>
<td>230/1881(12.23%)</td>
<td>256/3376 (7.58%)</td>
<td>-</td>
</tr>
</tbody>
</table>

As we can see in the Table 22, almost all the test suites had some failures. In the case of Notes and Amaze, only Tau did not have a single failure. The test suite built for UiAutomator was clearly the most unreliable with many times more failures than the other ones, however, the failure percentage was still under 2%.

### Table 23. Number of test suite runs with at least one failure for each tool and app

<table>
<thead>
<tr>
<th>Tool</th>
<th>Notes</th>
<th>Amaze</th>
<th>Subtotal</th>
<th>Wikipedia native (Wi-Fi)</th>
<th>Total</th>
<th>Wikipedia WebView (Wi-Fi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appium</td>
<td>0/50 (0%)</td>
<td>8/55 (14.55%)</td>
<td>8/105 (7.6%)</td>
<td>53/100 (53%)</td>
<td>61/205 (29.76%)</td>
<td>29/79 (36.70%)</td>
</tr>
<tr>
<td>Espresso</td>
<td>0/50 (0%)</td>
<td>1/50 (2%)</td>
<td>1/100 (1%)</td>
<td>0/50 (0%)</td>
<td>1/150 (0.67%)</td>
<td>3/50 (6%)</td>
</tr>
<tr>
<td>Robotium</td>
<td>2/55 (3.63%)</td>
<td>0/50 (0%)</td>
<td>2/105 (1.9%)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Tau</td>
<td>0/52 (0%)</td>
<td>0/50</td>
<td>0/102 (0%)</td>
<td>11/68 (16.18%)</td>
<td>11/170 (6.47%)</td>
<td>-</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>14/60(23.33%)</td>
<td>12/65 (18.46%)</td>
<td>26/125 (20.8%)</td>
<td>143/209 (68.42%)</td>
<td>169/334 (50.60%)</td>
<td>-</td>
</tr>
</tbody>
</table>

In the Table 23 above three figures are reported: failed test suite runs out of the total number of test suite runs and the percentage of failed runs inside the parentheses. In this case UiAutomator’s failure percentage climbs high to 20%, meaning that every fifth test run has at least one test that fails even though there is nothing wrong with the application. When considering the Wikipedia application, the numbers are even worse. Almost 70% of UiAutomator’s test runs had at least one failed test, and the figure for Appium is not much better at 53%. Espresso was the only tool where failure rates with native portion of Wikipedia stayed at 0%.
4.4.1 Failures in the Notes test suite

Notes test suite consists of 13 tests. The Table 24 below presents the percentages and numbers of individual tests that failed when the test suites were run. On the rows showing the tool’s name the number of failed tests out of the total number of tests is presented (the total number of tests is: tests in the test suite * number of test suite runs shown in Table 23 in chapter 4.4). The rows under the tools show tests that had at least one failure and are linked to the tests presented in chapter 3.3.2 Devising the test set. This row also presents the number of times that test failed out of the number of times the test suite was run in total.

Table 24. Failed tests in Notes test suite

<table>
<thead>
<tr>
<th>Tool</th>
<th>Failed tests out of total tests run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appium</td>
<td>0/650 (0%)</td>
</tr>
<tr>
<td>Espresso</td>
<td>0/650 (0%)</td>
</tr>
<tr>
<td>Robotium</td>
<td>4/715 (0.56%)</td>
</tr>
<tr>
<td>T6: Create and delete a note</td>
<td>2/55</td>
</tr>
<tr>
<td>T9: Clear done tasks</td>
<td>2/55</td>
</tr>
<tr>
<td>Tau</td>
<td>0/676 (0%)</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>14/780 (1.80%)</td>
</tr>
<tr>
<td>T8: Create task list and delete it</td>
<td>11/60</td>
</tr>
<tr>
<td>T12: Add a note, rotate the screen and check if note is visible</td>
<td>2/60</td>
</tr>
<tr>
<td>T13: Add task lists, scroll navigation drawer down</td>
<td>1/60</td>
</tr>
</tbody>
</table>

In the case of UiAutomator the test suites that failed all had only a single failing test. The test suites are overall quite reliable, though for some reason the test creating and deleting a task list (T8) was failing quite often.

4.4.2 Failures in Amaze test suite

The amaze test suite consists of 11 tests. Table 25 below presents the failed tests and the percentages per tool. As can be seen in the table, the test suites were reliable. Worst tool was UiAutomator but even there under 2% of the tests failed.
Table 25. Failed tests in Amaze test suite

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>Failed tests out of total tests run</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Appium</strong></td>
<td>8/605 (1.32%)</td>
</tr>
<tr>
<td>T2: Add a new text file and delete it</td>
<td>1/55</td>
</tr>
<tr>
<td>T7: Test if grid view / list view works</td>
<td>7/55</td>
</tr>
<tr>
<td><strong>Espresso</strong></td>
<td>1/550 (0.18%)</td>
</tr>
<tr>
<td>T8: Add to bookmarks</td>
<td>1/50</td>
</tr>
<tr>
<td><strong>Robotium</strong></td>
<td>0/550 (0%)</td>
</tr>
<tr>
<td><strong>UiAutomator</strong></td>
<td>12/715 (1.68%)</td>
</tr>
<tr>
<td>T8: Add to bookmarks</td>
<td>1/65</td>
</tr>
<tr>
<td>T9: Edit a bookmark's name</td>
<td>11/65</td>
</tr>
</tbody>
</table>

In the case of Appium the test *testing the grid view* (T7) seemed to be the test that caused most of the failures, while on UiAutomator the *edit bookmark test* (T9) test had the most issues. For some reason, Espresso also had a single test, *Add file to bookmarks* (T8) fail once.

4.4.3 Failures in Wikipedia test suites

The Wikipedia native test suite consists of 9 individual tests. As can be seen in Table 26, the Wikipedia’s test suites with Appium and UiAutomator were much less reliable than the test suites for the other applications. Almost 9% of the tests on Appium failed while 12% of the tests on UiAutomator failed. Tau had a manageable failure rate of under 2%, but sadly because of the different recording method the tests that failed with Tau were not recorded.
Table 26. Failed tests in Wikipedia native test suite using Wi-Fi connection

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>Failed tests out of total tests run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appium</td>
<td>76/900 (8.44%)</td>
</tr>
<tr>
<td>NT1: Search for an article, make sure its title is shown on top of the screen</td>
<td>2/100</td>
</tr>
<tr>
<td>NT2: See that recent searches shows an article</td>
<td>1/100</td>
</tr>
<tr>
<td>NT3: Test changing language when searching</td>
<td>29/100</td>
</tr>
<tr>
<td>NT4: Test scrolling table of contents down and clicking a subheading, assert title is no longer visible</td>
<td>23/100</td>
</tr>
<tr>
<td>NT5: Test changing language in an article</td>
<td>3/100</td>
</tr>
<tr>
<td>NT6: Make multiple tabs, see that you can switch between them</td>
<td>7/100</td>
</tr>
<tr>
<td>NT7: Add an article to a reading list, see it is available and can be clicked</td>
<td>8/100</td>
</tr>
<tr>
<td>NT9: Test opening table of contents and rotating phone</td>
<td>3/100</td>
</tr>
<tr>
<td>Espresso</td>
<td>0/450 (0%)</td>
</tr>
<tr>
<td>Robotium</td>
<td>-</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>230/1881(12.23%)</td>
</tr>
<tr>
<td>NT1: Search for an article, make sure its title is shown on top of the screen</td>
<td>3/209</td>
</tr>
<tr>
<td>NT2: See that recent searches shows an article</td>
<td>48/209</td>
</tr>
<tr>
<td>NT3: Test changing language when searching</td>
<td>24/209</td>
</tr>
<tr>
<td>NT4: Test scrolling table of contents down and clicking a subheading, assert title is no longer visible</td>
<td>9/209</td>
</tr>
<tr>
<td>NT5: Test changing language in an article</td>
<td>19/209</td>
</tr>
<tr>
<td>NT6: Make multiple tabs, see that you can switch between them</td>
<td>13/209</td>
</tr>
<tr>
<td>NT7: Add an article to a reading list, see it is available and can be clicked</td>
<td>93/209</td>
</tr>
<tr>
<td>NT8: Test history</td>
<td>14/209</td>
</tr>
<tr>
<td>NT9: Test opening table of contents and rotating phone</td>
<td>7/209</td>
</tr>
</tbody>
</table>

In the case of Appium the most problematic tests seem to be the tests NT3 and NT4. It is unclear in the change language test (NT3) why the test is failing; the test reports seem to indicate that the language of the article is not correct when the final assertion is done. The test for clicking subheading in table of contents (NT4) on the other hand seems to be failing since the element that is searched inside a navigation drawer is not found. Without observing the test runs again it is unclear whether this problem is related to the navigation drawer opening and Appium not waiting long enough until it starts searching inside it or if the element that is searched inside it is not found in a reasonable timespan. It is also possible that the opening of the drawer is not reliable enough.

In the case of UiAutomator the most problematic tests were check the recent articles test (NT2) and one adding article to a reading list (NT7). The test NT2 fails a quarter of the time. This test seems to be failing for some reason always when a new article is searched after an article has already been opened. It could be that the cause of this failure is related to some timing issues where UiAutomator is not waiting long enough for the article to load before it attempts to continue with the test script. The test NT7 failed almost half of the time when the test suites were run. With this high failure number, it could be that there
is something wrong with the test itself, but it seems that the test is failing at different sections of the test script, making it difficult to figure out afterwards why the test is failing.

**Table 27. Failed tests in Wikipedia WebView test suite using Wi-Fi connection**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Failed tests out of total tests run</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Appium</strong></td>
<td></td>
</tr>
<tr>
<td>WT1: Test that table of contents shows correct subtitles</td>
<td>4/79</td>
</tr>
<tr>
<td>WT2: Test following a link by full text</td>
<td>4/79</td>
</tr>
<tr>
<td>WT3: Test following a link by partial text</td>
<td>1/79</td>
</tr>
<tr>
<td>WT4: Test expanding and closing references</td>
<td>16/79</td>
</tr>
<tr>
<td>WT5: Open an article in a new tab</td>
<td>3/79</td>
</tr>
<tr>
<td>WT6: Test opening tabs and rotating phone</td>
<td>5/79</td>
</tr>
<tr>
<td><strong>Espresso</strong></td>
<td></td>
</tr>
<tr>
<td>WT1: Test that table of contents shows correct subtitles</td>
<td>1/50</td>
</tr>
<tr>
<td>WT3: Test following a link by partial text</td>
<td>1/50</td>
</tr>
<tr>
<td>WT6: Test opening tabs and rotating phone</td>
<td>2/50</td>
</tr>
</tbody>
</table>

Wikipedia’s WebView portion test suite consists of 6 tests. Results of the failed tests for this test suite are visible in Table 27. In the case of Appium the most failing test seems to be the one that *opens and closes the references inside an article* (WT4). This test is failing a fifth of the time, and the cause for this seems to be that when searching inside the navigation drawer the correct element is not found. This is a similar issue as was with the native portion of the Wikipedia application, and it could be that there is something wrong with either the code or the UiAutomator portion that Appium uses behind the scenes to search for the element inside the list.

Of the failing Espresso tests three seem to be related to Espresso waiting for too long for the application to go idle. Reason for this is not clear, but it is possibly related to either the device going idle and shutting down the screen for some reason, or that it takes too long for the article to load up and Espresso times out before this. The last test fails because the search bar in the application was not found and it is unclear why this would be the case.

**4.5 Ranking of the tools**

The tools were given ranks based on how well they performed in the different categories, as discussed in the previous chapters. The tools were given score on how high they rank in each category, and the tool with the lowest score could be considered as “the best” in the category. Several different ranking schemes was used, since all of them give a bit different results. The results for these different schemes are visible in Table 28, in Table 29 and in Table 30. In all the schemes Espresso and Tau are the tools with the best scores. Appium, Robotium and UiAutomator change in positions depending on which scheme is used.

Since Robotium was unable to automate the Wikipedia application, it was decided that the scores for execution time, lines of code, number of waits and reliability are based on the test suites ran on the Notes and Amaze applications. The results including Wikipedia’s native test suites is included in parentheses.
Table 28. Ranking of the tools, no weighting of tool performance. If Wikipedia’s test suite is also considered is the score shown in parentheses.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Issues when developing test suite</th>
<th>Execution time</th>
<th>LOC</th>
<th># of waits</th>
<th>Reliability</th>
<th>Compatibility with applications</th>
<th>Total</th>
<th>Total with Wikipedia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Espresso</td>
<td>4 (4)</td>
<td>1 (1)</td>
<td>3 (4)</td>
<td>1 (1)</td>
<td>2 (1)</td>
<td>1</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Tau</td>
<td>1 (1)</td>
<td>4 (3)</td>
<td>1 (1)</td>
<td>2 (2)</td>
<td>1 (2)</td>
<td>3</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Appium</td>
<td>2 (2)</td>
<td>5 (4)</td>
<td>4 (2)</td>
<td>4 (3)</td>
<td>4 (3)</td>
<td>1</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>2 (2)</td>
<td>3 (2)</td>
<td>2 (3)</td>
<td>5 (4)</td>
<td>5 (4)</td>
<td>3</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>Robotium</td>
<td>5 (5)</td>
<td>2 (-)</td>
<td>5 (-)</td>
<td>3 (-)</td>
<td>3 (-)</td>
<td>5</td>
<td>23</td>
<td>-</td>
</tr>
</tbody>
</table>

Scoring used in Table 28 is very naïve, the tools just get a score depending on which position they get based on the criterion in the previous chapters, with no regard on how well relatively they fared. In this table, Espresso and Tau score higher than the rest of the tools. Espresso’s main strengths are execution time, number of waits and compatibility, while Tau’s are the lack of general issues, lines of code and reliability. When the Wikipedia application is considered the scores switch around a bit, but Espresso and Tau are still the winners. UiAutomator is the worst when also Wikipedia application is considered.

The column compatibility with applications was added to the tables to show how usable in different situations the tools were. There were 3 applications and 4 different test suites, so Appium and Espresso, being able to reliably handle all situations, were given a score of 1. UiAutomator and Tau were not able to handle WebView portion of Wikipedia reliably so they were given a score of 3. Finally, since Robotium was not able to reliably automate the testing of Wikipedia at all, it was given the highest number of points, 5.

For the Lines of code column (presented in more detail in chapter 4.3.1) the value with no imports included is the number that is used and for the reliability the individual test failure percentages were used.

Standardization was also used to consider how well a tool performed in a certain category in relation to the best or the worst tool. Standardization means that the mean values of the data sets are used to correct the data (Ruijter et al., 2006). The values presented in the Table 29 below are the standardized results from the different categories discussed in earlier chapters. The formula used is as follows:

\[ X_{SCORE} = \frac{X_i - X_{MIN}}{X_{MAX} - X_{MIN}} \]

where \( X_{SCORE} \) is the score that the tool gets from a certain category, \( X_i \) is the result of a tool for a category, \( X_{MIN} \) is the “best” result for the category and \( X_{MAX} \) is the “worst” result for the category. (Etzkorn, 2011)

The formula gives the best tool a score of 0, the worst tool a score of 1 and the rest of the tools get a score in between depending on how close they were to either. Different
categories have different meanings of best or worst values, for example in the case of execution time the best means the fastest execution time, where as in the case of reliability it means the smallest percentage of failed tests. This scoring has the weakness that the best tool gets a score of 0 and the worst 1, with no regard on how small the differences are. If the differences are very small, the formula still gives a very large score to all but the winner.

Table 29. Weighted ranking of tools using formula by Etzkorn, B. If Wikipedia’s test suite is also considered is the score shown in parentheses.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Issues when developing test suite</th>
<th>Execution time</th>
<th>LOC</th>
<th># of waits</th>
<th>Reliability</th>
<th>Compatibility with applications</th>
<th>Total</th>
<th>Total with Wikipedia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tau</td>
<td>0 (0)</td>
<td>0.59 (0.46)</td>
<td>0 (0)</td>
<td>0.03 (0.17)</td>
<td>0 (0.07)</td>
<td>0.5</td>
<td>1.12</td>
<td>1.20</td>
</tr>
<tr>
<td>Espresso</td>
<td>0.42 (0.42)</td>
<td>0 (0)</td>
<td>0.87 (1)</td>
<td>0 (0)</td>
<td>0.05 (0)</td>
<td>0</td>
<td>1.34</td>
<td>1.42</td>
</tr>
<tr>
<td>Appium</td>
<td>0.08 (0.08)</td>
<td>1 (1)</td>
<td>0.91 (0.84)</td>
<td>0.30 (0.38)</td>
<td>0.37 (0.51)</td>
<td>0.37 (0.51)</td>
<td>2.66</td>
<td>2.81</td>
</tr>
<tr>
<td>Robotium</td>
<td>1 (-)</td>
<td>0.44 (-)</td>
<td>1 (-)</td>
<td>0.21 (-)</td>
<td>0.184 (-)</td>
<td>1</td>
<td>3.83</td>
<td>-</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>0.08 (0.08)</td>
<td>0.50 (0.43)</td>
<td>0.77 (0.9)</td>
<td>1 (1)</td>
<td>1 (1)</td>
<td>0.5</td>
<td>3.85</td>
<td>3.91</td>
</tr>
</tbody>
</table>

In Table 29 the scores in category columns have inside parentheses the score the tool gets if the Wikipedia application is also considered. As we can see in the table, there are 3 distinct groups on how the tools get their score. In the best performing group is Tau and Espresso, both with under 1.35 points, with Tau being the winner and Espresso on the second place. In second group at the middle is Appium with 2.7 score. In the worst-performing category are Robotium and UiAutomator with a very similar score, 3.8 and 3.9 respectively.

Finally, the tools were also ranked by using a weighting function by calculating the score a tool gets in relation to the “worst” tool. The formula used is following:

$$X_{SCORE} = \frac{X_i}{X_{MAX}}$$

Where $X_i$ is the result for a tool in a category and $X_{MAX}$ is the worst result in the category. Compatibility with applications was considered in this case so that how many applications or test suites were incompatible with the tool, so for Appium and Espresso that would be 0, for Tau and UiAutomator it would be 0.5 since they were not able to handle the WebView portion of Wikipedia and for Robotium it would be 1 since it was not able to handle Wikipedia application’s native or WebView portion.
**Table 30.** Weighted ranking of the tools, using \((X/X_{\text{MAX}})\). If Wikipedia’s test suite is also considered the score shown in parentheses.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Issues when developing test suite</th>
<th>Execution time</th>
<th>LOC</th>
<th># of waits</th>
<th>Reliability</th>
<th>Compatibility with applications</th>
<th>Total</th>
<th>Total with Wikipedia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Espresso</td>
<td>0.63 (0.63)</td>
<td>0.20 (0.19)</td>
<td>0.97 (1.0)</td>
<td>0.0 (0.0)</td>
<td>0.0 (0.0)</td>
<td>0</td>
<td><strong>1.8</strong></td>
<td>1.82</td>
</tr>
<tr>
<td>Tau</td>
<td>0.37 (0.37)</td>
<td>0.67 (0.63)</td>
<td>0.78 (0.78)</td>
<td>0.0 (0.17)</td>
<td>0.0 (0.08)</td>
<td>0.5</td>
<td><strong>2.32</strong></td>
<td>2.53</td>
</tr>
<tr>
<td>Appium</td>
<td>0.42 (0.42)</td>
<td>1.0 (1.0)</td>
<td>0.98 (0.97)</td>
<td>0.3 (0.38)</td>
<td>0.37 (0.51)</td>
<td>0</td>
<td><strong>3.07</strong></td>
<td>3.28</td>
</tr>
<tr>
<td>Robotium</td>
<td>1.0 (1.0)</td>
<td>0.55 (-)</td>
<td>1.0 (-)</td>
<td>0.2 (-)</td>
<td>0.18 (-)</td>
<td>1</td>
<td><strong>3.93</strong></td>
<td>-</td>
</tr>
<tr>
<td>UiAutomator</td>
<td>0.42 (0.42)</td>
<td>0.60 (0.54)</td>
<td>0.95 (0.98)</td>
<td>1.0 (1.0)</td>
<td>1.0 (1.0)</td>
<td>0.5</td>
<td><strong>4.47</strong></td>
<td>4.44</td>
</tr>
</tbody>
</table>

In Table 30 the worst-performing tool gets a score of 1 and better tools some fraction of that. The number in parentheses is the result if Wikipedia application is also considered. As can be seen, with this scoring scheme Espresso would take the top position and Tau the second. Espresso’s lack of waits, high compatibility and a fast execution time give it the edge. Again, UiAutomator would be on the worst position because of the large number of waits and unreliability.
5. Discussion and Implications

In this chapter, the results presented in previous chapter are discussed. This chapter is split up to 6 subchapters, first four subchapters discussing the results individually in the same order as findings were presented in the previous chapter. After this some conclusions are drawn in chapter 5.5, and finally limitations are discussed in chapter 5.6.

5.1 Encountered issues while developing the test sets (RQ 2.1)

When looking at the figures in Table 13, Appium, Tau and UiAutomator scored highly. These tools had in general less issues than Espresso and Robotium did, and these three tools have very similar scores. However, it should be considered that not every category is applicable with every application. Issues presented in the Table 12 and in chapter 4.1 are ones that surfaced with the three applications included in this study. Issues D1, D6 and D8 for example might well not be applicable with the application, since not every application has a WebView component inside it nor does every application require specific permissions. Similarly, one could use some other method for granting the permissions for the application besides the testing tool itself, especially in a dedicated testing environment where one could run Android Debug Bridge command before running the test suite. Of course, testing if the application functions correctly after a specific permission is not granted to it through the UI of the device is desirable, so a testing tool that can do it would be good.

Appium’s main issues were searching inside a list, opening navigation drawer and handling permissions. Opening a navigation drawer reliably is an important feature, since many modern Android applications include one and it is a feature that should be tested. It is surprising that permissions are an issue with Appium since it can easily push buttons from applications besides AUT. However, the issues were related to launching the AUT through the recommended way with Appium and the permission popups appearing at that point. Searching inside lists is cumbersome with Appium, since one needs to feed it UiAutomator script as a String. This is very error-prone, and along with the slow execution times discussed in chapter 4.2, it negatively affects test development.

Espresso had more problems than Appium, and they are related to touching the UI not belonging to the AUT. This manifests in the problems with permissions and in AUT opening another application. Asynchronous UI updates can also be more difficult to deal with on Espresso, especially since usually a tester does not need to bother with them at all. Since per the documentation Espresso is not meant for testing other applications besides AUT, these problems are not surprising. It does however make test development more difficult at a times. A positive note is that Espresso code can be mixed with UiAutomator code, so it could be used to interact with the other applications’ UI instead.

Robotium had much more issues than the other tools. All but two categories were at least a minor issue with this tool. It is surprising that Robotium had issues with WebView, since the API description promises that it can interact with that widget. It could be that it worked with an older version of Android, but with some new issues can have surfaced. Opening a navigation drawer was flaky, a swipe action used to emulate this was difficult to make reliable and the API call to push the correct button is named oddly. Since contentDescription of an UI widget can’t be used with Robotium this method call was however necessary. Similarly to Espresso, Robotium can’t touch UI of applications besides AUT so permissions and AUT opening another application are problematic.
Robotium can’t really work around the problem with permissions, since there is no possibility to use UiAutomator unlike with Espresso. These issues make Robotium somewhat uninviting to use.

Tau had the best score of the tools in general issues. It only had issues with WebView and interactions with navigation drawer items. Tau could not interact reliably with the WebView components, since it seemed to fail occasionally to find an element in the UI. There did not seem to be much logic to this and in the end, it seemed impossible to build a reliable set of tests for the WebView portion of Wikipedia. Touching a dynamic element inside a WebView (such as an expandable list) seemed to be impossible with Tau. Tau’s documentation did not promise that it would be able to drive the WebView at all, so it was a surprise that it could click some of the elements at all. The other, more minor, issue was that pushing elements inside a navigation drawer was unreliable. With Tau one can limit interactions with the UI to a certain part of the screen, but it does not make sure that the element is visible to the user. This can lead to it clicking coordinates of an element that is behind the navigation drawer instead of clicking the element inside the drawer, which can lead to surprising test failures that are difficult to debug.

UiAutomator also had only two large issues, WebView and asynchronous UI updates. WebView was impossible to automate reliably with UiAutomator. As was with Tau, some of the elements could be clicked but it was not always reliable. Some elements could arbitrarily be found and others could not. Also like Tau, UiAutomator could not touch an expandable list element inside a WebView. The other major issue with UiAutomator was clearly visible with the Wikipedia application. UiAutomator had very large issues in waiting for the proper amount of time until an element was finished loading. Much fidgeting needed to be done to the test suite to get it working somehow, and in the end, it was the most unreliable one as can be seen in results for reliability in chapter 4.4. It is not clear what is the reason for this issue, since it was not met with asynchronous activities in the Amaze application. Developing the test suite for Wikipedia was difficult with UiAutomator, but it is not clear if other more complex applications would run into the same problems.

Navigation drawer seems to be a problematic UI component to handle, since all the tools had some problems with it, be it either opening it or interacting with elements inside it. WebView is also problematic for some tools, and it indeed seems that UiAutomator and Tau do not even make promises that they would be able to handle it. If Espresso and UiAutomator would be considered as one tool, they would score very high in this category. UiAutomator’s main weaknesses could be complemented with Espresso easily, making it perhaps the most preferable of the tools. However, in this category the tools that can automate the UI of the whole phone seem to have an edge, since those scored the highest. In the end however Tau wins this category very slightly.

5.2 Execution times (RQ 2.2)

It has been shown that Java is generally faster than Python is (Nanz & Furia, 2015). However, it should be clear that how the tool drives the UI is much more impactful on the execution time of the test suite than the language chosen is, since the testing tools differ in architecture and their style of driving the UI.

As is with the differences between programming languages in the study by Nanz & Furia (2015), the tools have wildly varying execution times. In that study, it was found that Java is a faster programming language than Python is. The results of this study are different
than that, since some of the Java-based tools are faster and some slower than the Python-based Tau is. It seems that only the programming language used is not the deciding factor, and the way tool handles the UI of the AUT and how the testing tool’s architecture looks like is of more importance. Another point can well be that the tasks done in the tests are not likely very processor-intensive, which is where the differences between languages could surface.

Espresso was clearly the fastest tool in this study, being from 270% to almost 500% faster in the median execution time for the Notes and Amaze applications, and similarly large differences when the Wikipedia’s native portion was considered. Appium was also clearly slowest of the tools, being over 625% slower than Espresso in the cases of Notes and Wikipedia native, and 380% slower in the case of Amaze.

In a benchmark study between two web-based testing tools Watir and Selenium, it was found that differences between the tool/browser configurations range from 3.7% to nearly 92% longer execution time taken by some configurations when compared to the fastest configuration (Kuutila, Mäntylä, & Raulamo-Jurvanen, 2016). In this study the slowest tool using the fastest browser was 25% slower than the fastest tool. In another study comparing Watir-webdriver and Selenium Webdriver the differences between those tools were found to be around 12% in execution speed (Angmo & Sharma, 2014). Yet another study reports that differences between web testing tools Sahi Pro and Selenium Webdriver were 55%-40% between the tools, depending on the browser that was used (Singh & Sharma, 2015). The differences found in this study seem to be much larger particularly if Espresso and Appium are considered. Between Robotium, Tau and UiAutomator the differences are closer.

When looking at the test execution Appium is slow at typing text. This could explain why it is much slower at Notes and Wikipedia when compared to Amaze, since those applications require quite a lot of typing in the test scripts.

For the very simple Notes application, Tau and UiAutomator are also almost four times slower than Espresso is, while Robotium is only at 217% execution time of Espresso. It could be that in the case of Robotium the difference in Notes application comes from the fact that Robotium does not actually type in the text letter by letter but just adds the text to the UI widget, which in the typing-heavy Notes test suite could make the difference.

With the Amaze File Manager application Robotium and Tau are around 300% of Espresso’s execution time while UiAutomator is at around 200%. It is unclear where these differences could come from. It is possible that for some reason UiAutomator is either faster at searching elements in lists or that searching for some unorthodox elements in the lists is faster with UiAutomator. Appium, perhaps because it is using UiAutomator behind the curtains, also experiences a bump in performance in the case of the Amaze application: only 382% longer execution time than Espresso when comparing to 636% larger with Notes and 625% larger in Wikipedia Native portion.

As we can see in the Figure 10 in chapter 4.2, the spread of the execution times with Appium and UiAutomator are very similar. Since behind the curtains Appium uses UiAutomator, it could make sense that this would be the case. It could be that the increased execution time is just the overhead of what Appium does to enable the client-server communication per to its’ architecture. In the case of Amaze (Figure 9) and Appium there’s many outliers and 50% of the values are packed more densely with UiAutomator, but there is overall similar kind of spread in the results. In the case of Notes (Figure 8) there are even more outliers with Appium and the 50th percentile is more
densely packed than it is with UiAutomator. Except for Tau with Wi-Fi network on native Wikipedia application, the other tools have overall more densely packed results than Appium and UiAutomator have.

Espresso, Robotium and Tau have quite little variation in their execution times. Tau does have some more variance in the Wikipedia application, especially on 4g connection, but it is possible that this could be because of the network connection, as in the tests the actual articles were loaded over the network from Wikipedia’s servers. Tau also does have some outliers in the result set, but these do not seem too numerous. It is also unclear as to why some of the outliers are under the 50th percentile of the execution times – for some reason some test runs were exceptionally fast.

If the execution time of the testing tool is considered, it does not make that much difference whether Robotium, Tau or UiAutomator is chosen. Espresso is at all cases much faster than the other tools and Appium is the slowest. It is possible that Espresso’s way of looking at the application’s main thread event queue and continuing test execution as soon as that is empty is faster than some alternate way the other tools are doing it. When physically looking at the test execution it also seems that Espresso is much faster in scanning the device screen for the UI elements, along with being faster when searching for elements inside lists. When searching inside a list Espresso polls the class holding the elements in that list directly for the element that is searched and clicks it, while the other tools seem to scan the screen for the element and then swipe to some direction and scan the screen again. This can reduce the time to run the test significantly for Espresso.

Also, Appium’s client-server communication adds much overhead to the execution time, since the server needs to handle the commands before sending them to the test device, and it needs to construct a reply before sending it to the client which needs to do its’ own processing before it sends the next test script step to the server. Another thing is that Appium might be doing something in between the tests related to clearing the application’s settings and database, and this could be visible in the execution time, as discussed in chapter limitations (6.1).

When looking at standard deviations for the execution times instrumented tools are more predictable. Espresso’s and Robotium’s standard deviation is well below 1 second, while UiAutomator’s is a bit over 3 seconds with both Notes and Amaze. Tau has a bit more variance in the execution time, and Appium climbs even higher in this. Considering Appium’s higher execution times overall this is to be expected. It seems that since the instrumented tools are completely run on the device it reduces unpredictable communication between the host machine (and different processes on the host machine with Appium) and the device lessening the deviation in the execution times. It could be that the Android Debug Bridge used for this host machine - Android device communication brings some unpredictability to the communication speed.

5.3 Maintainability (RQ 2.3)

The lines of code required to build the actual test suite (not including the import statements) differs between the tools by a decent amount, almost 30% difference between the most verbose tool, Tau, and the worst tool, Robotium. The differences are not staggeringly large however. If not considering Tau, since it uses a different language than rest of the tools, there is only 5% difference between the tools. The differences between the tools are larger when looking at the number of waits, and UiAutomator is the worst in this regard.
In a study where different programming languages were compared, it was found that of the compared languages, C#, Go, Java, C, Haskell, Ruby, F# and Python, Python is the most concise of the languages (Nanz & Furia, 2015). This can be seen in the test code constructed in here as well, since the Tau test script had 20% less lines of code than the UiAutomator, which had the second least (when looking at only Notes and Amaze File Manager). However, the results reported by Nanz & Furia (2015) are much larger than what are observed here, they report Java programs being two times longer than Python programs, which is much less than is noticed in this study.

When looking at the lines of code with import statements it looks much different. Tau is still the winner but with much larger margin, requiring only 60% of the lines of code when compared to the worst tool, Espresso. Rest of the tools are closer to Tau, requiring only around 30% more code than it. The same trend continues if Wikipedia application is taken into consideration as well, though at this case Appium is the most concise of the three tools in middle positions. The margin is however so small that it could well come down to individual coding style.

The results in this study are similar as those of Kuutila et al. (2016) when they compared the conciseness of the test scripts created with Watir and Selenium. In all cases the Python test scripts for Selenium were shorter than the ones written in Java, although with smaller differences. Although the environment where these tools are used is different it seems Python is more expressive.

When looking at the number of waits required by the test suite, the results are very different. As mentioned, one main feature of Espresso is that it handles synchronization automatically, meaning that no wait methods should be needed when the test suite is developed (“Testing UI for a single app” n.d.). Espresso is the winner in this category since it requires no wait method calls. As a test developer, it is much easier to develop the test if one does not need to consider after every script step if the next element looked for should be waited for instead of just clicking it or asserting on it. Tau was also very easy to use in this manner since it only required two wait methods in the Notes and Amaze applications, and as mentioned earlier in chapter 4.3.2 in Figure 13 the Tau’s way of waiting for an element is simpler than it is for Appium or UiAutomator.

The Wikipedia application is very asynchronous and this causes problems. Tau’s number of wait parameters rose clearly from 2 to 17 since it does not know that it should wait for a slow network call. In this case Appium’s number of waits also doubled from 19 to 38. UiAutomator also needs many of these method calls, going from 63 to 100.

UiAutomator was clearly the worst in the number of waits, since it required 63 wait methods in the case of Notes and Amaze. When compared to 19 of Appium or 13 of Robotium the differences are clear. Building a reliable suite of test with UiAutomator is difficult. The lack of waiting methods required by Espresso makes the test script easier to read in the author’s opinion. Some tools, as presented in Figure 13 in chapter 4.3, can require code that takes multiple lines to wait for an element.
As mentioned earlier, in the case of Espresso the source code of the application needs to be updated per the official guidelines presented for Espresso ("Bonus: Idling resources" n.d.). This can however be a bad idea, since additional complexity to the software can never be a good thing. Another source mentions that this is a practice that should not be done and that can be circumvented by using dependency injection (Dupree, 2016). If this fact is not an issue, Espresso is the clear winner in the number of wait method calls since it requires none, but sadly it is not a tool where a test developer can completely ignore asynchrony either. Tau is the winner in the conciseness of the code, as the code required by it is shorter and is straightforward to read.

5.4 Reliability (RQ 2.4)

As can be seen in chapter 4.4, Espresso takes the top spot regarding reliability. When considering only Notes and Amaze applications Espresso (with 0.08% failure rate) is worse than Tau (0% failure rate), but only negligibly. If also Wikipedia is considered, Espresso takes the lead with 0.06% failure rate to Tau’s 0.6% failure rate. The other tools however have several percentage failure rates.

When looking at only Notes and Amaze File Manager, Robotium is quite reliable with under 0.5% test failure rate. Appium has also very few failed tests with Notes and Amaze File Manager (0.6% failure rate), but the failure rate climbs higher with the Wikipedia application to 3.9%. Throughout the line UiAutomator is in every case at the worst spot when it comes to the test suite reliability, going from 1.7% with Notes and Amaze to 7.6% if Wikipedia is included. Largest failure rate was with Wikipedia using Wi-Fi connection. It had over 12% of the tests that failed during the test runs, leading to 68% of the test runs including at least one failed test run. However, if considering Notes and Amaze File Manager the failure rates are under 2%, and under 8% if Wikipedia is included as well.

When comparing the failed run numbers to those found by Kuutila et al. (2016), the numbers are in a bit similar fashion. In their study the authors only reported test runs that had failed tests, not how many individual tests failed. In their study, several of the tools had no problems when running the tests, while in this study only Tau with Notes and Amaze accomplished this. In the study by Kuutila et al. (2016) the test run failure percentages ranged from 0% to 32%, while in this they ranged from 0% to 20.8% if considering Notes and Amaze and from 0.7% to 50.6% if considering also the Wikipedia tests, as can be seen in Table 23. Clearly not only Android UI testing tools struggle with failures.

Appium and Tau had, along with UiAutomator, a noticeable bump in the failure numbers with the Wikipedia application. It is possible that this is related to the large number of asynchronous calls that the application makes: every test in the suite had one or more asynchronous action, mostly related to loading up articles from Wikipedia. In their study about flaky tests Luo et al. (2014) mention that 45% of flaky tests are related to asynchronous waits (Luo et al., 2014). The reasons for test flakiness were not investigated in this study, but at a glance it seems that most of the flakiness issues are related to asynchronous waits. The other main reason for test failure in this study is faulty way the tool searches the UI and attempts to click on an element. Luo et al. (2014) also mention that over half of those tests could be fixed by using wait method calls. Many of the tests in this study could indeed be fixed by inserting wait call to a correct place in the code, but as can be seen in the results not every problem with asynchrony could be fixed, especially regarding the failure rates with Wikipedia’s test suites for Appium and UiAutomator. It
is not clear what is the reason that using proper waiting method calls does not fix all the issues regarding asynchrony.

When considering Notes, Amaze and Wikipedia’s native portion, Espresso was a clear winner in the reliability. It had only 0.06% of the tests (1 test in 1650) that failed. Espresso’s method for ensuring the stability of the application before continuing with the test execution is very effective. Tau could have reached similar levels of reliability, but it seemed to have some problems with finding some elements in the Wikipedia application. Often the test failed right at the start when Tau failed to search an article.

The individual test failures with Notes and Amaze range from 0% (Tau) to 1.7% (UiAutomator). These numbers are not that large and it would not be such a large issue if any of these tools was used. However, UiAutomator is still clearly the flakiest of the tools and it seems that it is difficult to build a reliable testing suite with UiAutomator. The differences become much clearer with the Wikipedia application (from 0% test failure rate with Espresso to 12.2% with UiAutomator). Since it is the only application where the differences are this large, it is possible that it just happens to be an application that is particularly difficult to test reliably. Notes and Amaze are however very static applications when compared to Wikipedia and they have very few elements that update asynchronously. Wikipedia also has calls over a network that can be unreliable. This seems to make the application more difficult to test reliably.

There is a possibility that the test suites could have been tweaked further to make them more reliable, but at with of UiAutomator and Appium, creating a non-flaky test suite seemed a difficult task indeed. This can, be related to the skills of the person developing the test suite, but it seems that in this instance it was easier to develop a non-flaky test suite with Espresso and Tau when compared to Appium and UiAutomator.

Overall it seems that Espresso and Tau are the best tools when it comes to reliability. Both tools had a very small number of flaky tests. The difference between these tools is that with Espresso asynchrony can be ignored at a times, but with Tau one is more likely to run into it. On the other hand, when using Espresso source code of the application might have to be modified so that Espresso knows what it needs to wait. This can be much more cumbersome than just passing an additional parameter to function call like can be done with Tau. Robotium also had very few failed tests, but the fact that it could not automate the Wikipedia application does not show well for its’ reliability. Tau’s failure rate did bump significantly with the Wikipedia application while Espresso’s stayed at 0% at the native portion of that application, making Espresso the winner in reliability category.

5.4.1 A look at the tests that fail

Both Notes and Amaze file manager had small fail percentages with all tools. Notes had a failure percentage of 0.6% with Robotium and 1.8% with UiAutomator, while other tools had no flaky tests. For Amaze Appium had failure rate of 1.3%, Espresso 0.2% and UiAutomator 1.7%, and similarly other tools had no failing tests.

For UiAutomator the test that caused most of the failures with Notes was create task list and delete it (T8). It seems that this test fails since UiAutomator tries to click a button that is not on the screen. For some reason UiAutomator does not seem to wait long enough for the button to appear even though a wait method is used here.
At Amaze with Appium the test *testing the grid view* (T7) seemed to be the flakiest one, and the problems seem to be related to pushing the overflow button on the upper right corner of the application. It is not clear why this happens, although it could be related to the situation where a swipe action must be done in the test script before this action can be done, and this swipe action could fail at a times. With Amaze and UiAutomator the *edit bookmark test* (T9) test was the main source of flakiness. When looking through the test repots it seems the problem is related to interaction with the navigation drawer. For some reason at a times UiAutomator could not find the navigation drawer.

Appium and UiAutomator had a large fail percentage with the native portion of Wikipedia. Appium being at 8.4% and UiAutomator at 12.2% failure rates can already be seen when running the test suites, and should be addressed. With Appium the failed tests for native Wikipedia app, as presented in Table 26, the failures seem to be able to be attributed mostly on two tests, NT3 and NT4. When looking closer at the reports of these test runs, it seems that NT3 failed because for some reason the language of the article was not changed. It could be that Appium was picking up an element that is not visible on the screen but that is still somehow in the UI structure of the app. NT4 was the other test that was failing quite often. It seems that most of the time the reason this test was failing since Appium was unable to start searching for elements inside a navigation drawer that is opened before. It is not clear if this issue would be solved if one would tweak the method call that should wait for the navigation drawer to be fully opened before continuing.

With UiAutomator on native Wikipedia app there was much more tests that failed often, but the two tests that had clearly highest rates were NT2 and NT7 Table 26. Overall there was not a single test that had not failed at least once. NT2 seems to fail mostly because for some reason opening the search while already in an article seems to fail often with this test. Particularly this test fails since the search bar is not found by UiAutomator. It is possible that this is since UiAutomator fails to wait long enough for the article to load before it continues with the test script. This same functionality is used in other test suites that also seem to be failing at a times, but not nearly as much as this test.

The other test that failed was NT7, which was the most failing test with 93 fails out of 209 times it was run. When studying the test script and the run reports afterwards it seems that there was a bug in the test script which could cause a portion of these failures. It could be that if this bug was fixed the failure percentage for UiAutomator and this test would go down. It is however unclear that why the test still passes half of the time if the test script itself is flawed. If we calculate that NT7 would fail at the same rate as the other tests, median for these is 13.5 and rounded up to 14, the test suite with UiAutomator would have only a failure rate of 8% (151 out of 1881 failed tests) instead of 12.2%, putting it a bit below Appium’s failure rate of 8.4%. This is all of course a speculation, but it is all that can be done short of re-gathering the results.

It is possible that if more time was spent on fixing these few tests that seem to fail often, the overall failure percentage for Appium and UiAutomator could be brought down closer to that of Espresso and Tau. However, it should be mentioned that these kinds of tests that fail very often were not really encountered when the test suites were built with Espresso and Tau. It seems then that with these two tools building a robust test suite is easier.
5.4.2 Costs related to failures

At Microsoft, it has been estimated that inspecting a test failure costs as much as $9.60 per a failed test (Herzig et al., 2015). On the other hand, because of the fragmentation on Android, the number of devices on which the application should be tested can be high. For an example, in 2012 Netflix used 14 different devices on with which to “smoke test” their application (Kher, 2012). If we consider this with the modest 13 tests that had been made for the Notes application in this study, this would lead to 182 (13*14) tests being executed every time the test suite is run. If 1.8% of these (as is the failure percentage for Notes with UiAutomator) fail, every time the test suite is run there would be on average 3.3 failed tests, leading to a theoretical $29 cost per test suite run from false failures alone.

Of course, it is arguable that the test suite would not have to be run on every device every time a change is pushed by a developer to the continuous integration environment, and that there could be a same place where a test fails with different devices leading to less inspection needing to be done, but nevertheless this can be a source of unnecessary costs especially if the test suite grows.

For an example in the Google’s open source Google I/O 2016 (https://github.com/google/iosched) application that is annually developed for the visitors of their I/O conference has 136 instrumented tests (tests that are run on an actual device or on an emulator). Very many those tests seem to be actual Espresso UI tests rather than just instrumented unit tests. If this test suite would be run on the 14 devices as Netflix did, it would lead to 1904 tests run, and at worst (with UiAutomator and 1.8% test failure rate), 32 of those tests on average would fail. This could lead to much headache for the developers and wasted resources as time is spent inspecting whether the failed tests failed because they are flaky or whether the application was actually broken.

5.5 Tool rankings

As can be seen in Table 28 in chapter 4.5, over all it seems that either Espresso or Tau takes the top spot as the “best” Android testing tool. In the categories discussed in the earlier chapters both tools scored overall high, though with some differences in areas of strength. Their ranks also differed depending on the scoring formula used. Appium, Robotium and UiAutomator overall even, the positions for the tools differed depending on the ranking scheme used. Appium usually had a bit lower total score (meaning it is a better tool) than Robotium than UiAutomator. UiAutomator seemed to perform worst of the tools. The high failure rates and the large number of wait method calls required by UiAutomator make it somewhat uninviting tool, since it seems that it is difficult to make very reliable tests with it.

Both Espresso and Tau had as their main strengths the low number of waits and reliability, while Espresso was better in execution speeds and application compatibility while Tau pulled ahead in the general issues found and in lines of code required for the test suites.

Since Espresso does not rely on any waiting for elements and Tau requires very little of it, it seems this is where they get their edge on the other tools. Many test failures seem to be related to the tool not waiting long enough for an element (with Appium T8 on Notes, NT3 and NT4 on Wikipedia native, NT2 with UiAutomator on Wikipedia native). Robotium does require some of these waiting calls, but it seems to be more reliable with the waits than UiAutomator and Appium are. Of course, Robotium was unable to drive the Wikipedia application at all and it seemed that the reason for this seemed to be that
Robotium somehow broke the application when it was waiting for elements, and the cause for this remained unclear.

When the categories where the tools’ main strengths lie, the author’s opinion is that Espresso would be considered the “best” of these tools. This is since the categories where Espresso was clearly better than Tau, namely in execution time and particularly compatibility with applications is in author’s opinion more valuable than categories where Tau won, general findings and lines of code. In addition to this Espresso’s absolute reliability and the lack of waits is very visible when the more complex application Wikipedia’s test suite is also taken into accord. In addition, in a real-world situation one would most likely not use either Espresso or UiAutomator, but one would use them together since they can be used in such a way. When looking at the results discussed before, it would seem reasonable that if one was to use the test library provided with Android one would write majority of the tests with Espresso to get the speed and reliability from Espresso and patch the test suite with UiAutomator on portions where Espresso can’t be used, such as when manipulating UI of an application different than AUT or granting permissions to the application. Finally, Espresso seems more inviting of the tools since it is freely available to anyone starting testing efforts unlike Tau, which requires a license to be bought. This can make it more attractive to the open source community or individual developers.

Tau would be on the second place when looking at the score it received in Table 28. Third place would be divided between UiAutomator and Appium. Appium is certainly slower than UiAutomator is, but it seems to be more reliable. Appium also has more popularity, as discovered in chapter 2.4, so it would make sense that it has more community support and making it thus easier to troubleshoot if problems are run into. Appium also can easily handle WebView and has a bit better scores when they were standardized, so it would seem to be a bit more preferred than UiAutomator. The third place seems to then go to Appium and fourth place to UiAutomator. It is interesting that Appium did not end up being one of the top tools, considering the popularity it seems to hold in the mobile testing world, as discovered in chapter 2.4. On the last place in the score is Robotium, which seemed to have most issues of the tools overall. It performed decently well in execution times and in reliability. It was overall faster than UiAutomator and Tau, but not by a large margin. It was also the only tool that seemed to completely break down when automating the Wikipedia application and it seems that there are some differences between the promises it makes in the API description and what it does, making it a bit unreliable at a times. When looking at the issues with Robotium discovered in this study it is not surprising that its’ popularity has waned over time, as discovered in chapter 2.4.
6. Conclusions

This is one of the first attempts to benchmark the different Android UI testing tools. Much of the previous research has focused on the automated testing of Android applications or is limited to presenting the different tools. In this study, several Android UI testing tools were identified, the popularity of these tools was shortly studied and the structure of the most popular tools was looked at. Of the available tools Appium and Espresso seem to be the two most popular tools. Robotium is a tool that was in the past very popular but seems to have lost that position. Robotium, Espresso and UiAutomator share similarities in how they work being instrumented tools, while Appium and Tau have a more novel approach to driving the applications. A look at the automated testing of Android applications was done at as well, and scientific research regarding this was identified, and it seems that this area has been much more active scientifically than the UI testing tools where a tester manually writes a test script.

Next several criteria for benchmarking the tools was identified. The criterion are ones used traditionally in software development, the speed, maintainability, and reliability of the tool. In addition, the different issues that arose while the tools were inspected were compared as well, as is discussed in chapter 4.1. The criteria were then examined by benchmarking them against each other by developing an identical suite of tests with each tool for three different Android applications. Data about these test suites was gathered by running the suites and by analyzing the test scripts generated.

Based on this analysis, several conclusions could be drawn, as is discussed in chapter 5. Espresso is the fastest of the tools at executing the test script, as is it the most reliable of them in test flakiness. Asynchronous UI updates are also easy to handle with it, making it a very desirable testing tool. Tau’s test scripts however seemed to be the most concise of the tools considered, and it had least amount of issues that arose when developing the test suites. These reasons make it also an appealing choice as a testing tool.

This piece of research will hopefully help people in the industry in choosing the best tool for their Android UI testing needs, or at least work as a starting point when doing a comparison of the tools for their specific needs. The different categories can give insight on what could be considered in a testing tool when choosing one. This research is also a step in expanding the research on benchmarking testing tools both for Android and for other areas as well.

6.1 Limitations

The study conducted does have some limitations that need to be considered. First, a more formal structure for comparing the different tools might have been useful, since this could make the reproducibility of the experiment be easier. In a similar vein, scientific sources for the popularity of the tools would help to make the results more convincing. However, since no industry studies regarding this were found, it was necessary to utilize non-scientific sources. While the comparison was not rigorously structured, the categories on which the tools were considered can be considered things that are usually of interest in the software world – speed, reliability, and maintainability.

In this study, the tools were inspected in a situation where not all capabilities of the tools were considered. For an example, Appium is a cross-platform testing tool rather than Android specific one, as the other tools are. This can give an edge for Appium since it
can significantly reduce the test writing efforts needed. Tau can also do much else besides simply driving the UI, but only that aspect was focused upon in this study. The evaluation of tools could be expanded in other ways as well, for an example maintainability could be better studied if a long-term study was carried out where the efforts needed to keep the test suite up-to-date could be researched. The evaluation could also be expanded by seeing how the tools support mocking of certain portions of the application – for an example in the Wikipedia app the server responses could have been mocked out. This would have had an additional effect of removing the effect of network speed on the test execution speeds.

Regarding the execution times, comparing the times does have the problem that since Appium takes care of other tasks besides just running the test, such as reinstalling the application after tests, but this could not really be avoided. If on other hand just the time it took from executing the command-line command to the moment test execution finishes is recorded, there are problems as well. The Gradle task might re-compile the application or do other activities besides running the test set arbitrarily, and these activities were noticed to take as much time as running the whole test set in some cases. Another point at running the tests is that Tau’s tests were not run through Gradle, which could have some effect on the recorded runt times. Overall, the tools are all somewhat different and some compromises when evaluating them must be made.

The author developing the test suites is not an expert tester nor a professional software developer, so this can influence the developed test suites and the generalizability of the results to industry world. The results should be put in perspective where a novice developer would start to develop a suite of UI tests to a previously unknown application.

The test sets could be more comprehensive and possibly include another more complex application to make the results possibly more generalizable, for an example the Notes application was a quite simple one with no too many asynchronous activities performed. Also, regarding the test set, it could have been expanded by adding in several tests that are supposed to fail to study if the tools give any false positives. Also, if the test suite for Robotium could have been finished properly the results would have been more comprehensive.

It might be interesting if the same study was carried out in a situation where a continuous integration environment would also be used, this would give a more real-world situation for the development and some new issues about the tools could be brought forward. In that case, the testing harness could also take care of some of the activities like cleaning up the environment after a test has been run and granting permissions to the application before a test is run. Finally, it has been noted that testing-as-a-service platforms and automatic cloud-based testing solutions can be how mobile applications are tested in the future (Starov et al., 2015). It is then unclear if these kinds of tools will still have widespread use in the future.

6.2 Further research

Since there were no scientific sources found regarding the popularity of the tools a study of this could be carried out. Along this, it seems that studying what are the most important aspects of a testing tool could be of interest, especially in the context of mobile application testing tools. A more focused study could be carried out after finding out which aspects of the tools industry values the most, focusing only on those aspects and thus making the study more useful to the industry setting.
Also, the study done in here could be re-made in a couple of years with the tools that are at that point the most popular ones so it could be seen if the tools are developing in certain directions regarding their strengths and weaknesses, and to give up-to date information to industry users of these tools. To better compare the tools complex applications should be used to clearly make out the differences between the tools. The comparison could also well be expanded by adding more functionalities of the tools, such as test recording or comparing the output of the tools, to the comparison.
7. Acknowledgements

The tool Tau is developed by company called Profilence, and they provided the tool for this thesis for free of charge. There was some contact between a few employees of the company and the author of this thesis during the writing process and while developing the test suites, but this did not affect the results presented in this thesis.
References


Bonus: Idling resources. (n.d.). Retrieved December 16, 2016 from
https://codelabs.developers.google.com/codelabs/android-testing/index.html?index=..%2F..%2Findex#9

https://developer.android.com/guide/webapps/webview.html#AddingWebView

Calabash. (n.d.). Retrieved December 10, 2016 from
https://github.com/calabash/calabash-android


Configure build variants. (n.d.). Retrieved September 21, 2016 from

Creating an android project. (n.d.). Retrieved February 20, 2016 from
https://developer.android.com/training/basics/firstapp/creating-project.html

Dashboards - platform versions. (n.d.). Retrieved August 31, 2016 from
https://developer.android.com/about/dashboards/index.html


Mobile automation testing. (n.d.). Retrieved November 17, 2016 from

http://www.ranorex.com/mobile-automation-testing.html

Mobile testing tools. (n.d.). Retrieved September 16, 2016 from

http://www.testingtools.com/mobile-testing/


Monkeyrunner. (n.d.). Retrieved November 17, 2016 from


https://developer.android.com/training/testing/ui-testing/espresso-testing.html

https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html

UI/application exerciser monkey. (n.d.). Retrieved December 19, 2016 from

Vasanth, K. (n.d.). How to do mobile app automation testing using MonkeyTalk tool?
Retrieved December 21, 2016 from
http://www.perfomatix.com/appdevelopmentblogs/mobile-app-automation-testing/


https://www.mediawiki.org/wiki/Wikimedia_Apps/Team


*A practical guide to writing solid UI tests on android [video file]*. Zakharov, V. (Director). (2016).[Video/DVD]

Appendix A Notes of developing test suites for each application

This appendix has notes about developing each of the test suites with each of the tool for each of the applications.

The issues in table N in chapter 4.1 is referenced in the notes about different issues that rose while developing the tests.

Notes about developing the test set for Notes

In general, developing the test suite for Notes was easiest of the all three applications. There were some issues that did take some time to handle, but mostly they were minor ones.

The biggest issue was dealing with the custom date and time picker that the application had. In the end the test that dealt with this UI widget (T7 in table XYZ: “create notes and order by due date”) was left out of the comparison since author did not find a way to implement this test with Espresso and it seemed to be not possible to implement with Robotium.

The application’s source code had to be modified slightly for the instrumented tests (Espresso, Robotium, UiAutomator) to handle removing data from the application’s database.

A bug in the AUT was discovered when building the test suite. When a big number of notes was added to a list of notes, the application would correctly show several them and then start showing the notes from the first one forward again. An issue was made on the GitHub repository of the application about this. Most of the tools would search the list for the note by scrolling it up and down and eventually reporting that an item with that identifier was not found, but Espresso functioned a bit differently.

Notes with Robotium

A problem with Robotium was that when searching from inside the navigation drawer the tool would start to search in the list behind the drawer, on the main portion of the visible screen instead (D3, table 9). This was later fixed in a newer version of Robotium after a GitHub issue was made.

Opening the navigation drawer was somewhat problematic as well – at a times it would fail to open properly (D2, presented in table 9). In the case of this application it was decided to open the navigation drawer by the hamburger icon on top left of the screen. This was a problem since Robotium does not offer a simple method to access elements by their ContentDescription (a text that Android’s Accessibility functions use), which would make it possible to click the hamburger icon, since that was the only way it was identifiable. Robotium’s own methods (solo.setNavigationDrawer(Solo.OPENED))) for easily opening the drawer did not seem to function either in this case. For this reason, it was necessary to emulate a swipe input to open the navigation drawer.

Notes with Espresso
Opening the navigation drawer and interacting with it was very easy with Espresso. Espresso provides a neat method for opening the navigation drawer without the need to simulate a dragging method or pushing a button. It is not clear to the author how Espresso does this, but it was reliable.

Since Espresso does not actually search the displayed list for an element but asks the component storing those elements what is the position of the element, Espresso would click on a position where it thinks the correct note should be at. This lead to some confusion on the part of the author on what was going wrong with the test, before he realized that there is a bug in the applications.

Notes with Appium

After initial learning period, there was no problems with Appium regarding the Notes application.

A point of difficulty was also searching inside a list of items for an item. Per a GitHub issue the way to search in a list in the current version of Appium is to use a method where a string of UiAutomator code is passed to Appium. Writing this is very error prone and can take a while to get right, especially considering the slow execution of the Appium tests.

One minor problem encountered, that was again experienced at the Amaze File manager application, is that Appium does not offer a clean way to assert if an element is visible on the screen or not (D9, presented in table 9). One must get all the elements that are on the screen and check if the array returned by this method is empty. Other tools either return a Boolean value if the item is found (making it possible to just make an assertion on this), or offer methods that fail the test if the item is not found.

Notes about developing the test set for Amaze

Handling the Android M’s permissions (D6, presented in table 9) proved to be a point of some difficulty with all the testing tools besides Tau and UiAutomator. For this reason, the permissions were granted to all tools besides Tau outside the test script itself to keep the results as consistent as possible. For Tau, this seemed to not be possible so the permission button was clicked before the set of test suites was left running.

One general point of difficulty was that the top part of the application’s UI will disappear when the user scrolls down the list of folders that are visible in the application. This is somewhat normal in Android applications and it is of course debatable whether it is a pro or a con that the testing tool scrolls down a list when searching in instead of just jumping at the correct location.

A difficult situation for all the tools was clicking a certain button that is shown in every row of the main list in the application – this button does not really have any distinguishable identifier on its’ own, the only identifier is the text field that is next to the button. However, with every tool it was in the end possible to reliably interact with this button, although it was not simple in any case.
Another point of problems was removing the files that the test suite creates during the tests. For instrumented testing tools (Espresso, Robotium, UiAutomator) this could be accomplished through running a shell command in the teardown phase of the whole test suite, but doing this requires methods that were added only in API level 21, which would make it impossible to use this method in apps that target older API levels than that. Appium and Tau make it possible to use other means to remove the files the application creates, since they are not bound to the life cycle of the AUT.

Finally, different tools had different ways to handle a situation where an external application would be launched by the AUT (D8, presented in table 9). This would happen when the user tried to open a video file or a picture file, for an example.

Amaze File Manager with Espresso

Espresso was slightly difficult to start to work with since the AUT in question has two screen portions which both have similar contents and Espresso “sees” these both as available which would lead to an error thrown stating that the View could not be uniquely identified.

Espresso’s insistence on uniquely identifying elements on the screen proved to be a point of difficulty with this application. The application has two very similar lists that user can swipe between, and since Espresso is aware that both of these lists are available to the user, swiping between these requires one to supply an additional “isDisplayed()” parameter when choosing the element with which to interact with.

Another point of difficulty was that for some reason clicking the navigation drawer items (D3, presented in table 9) was very difficult. The AUT’s source code actually had to be modified slightly, a toString() method was added for the items inside the navigation drawer so they could be identified properly. In addition, the author did not find a way to properly click the first item of the navigation drawer in the proper way (using onData method for it), but instead a workaround where simply the text is found and clicked was used. If the same text would be present on the screen elsewhere this approach would not work.

Espresso also has the problem of not being able to handle app permissions (D6, table 9).

Finally, it was somewhat more difficult to handle the asynchronous operation of copying a file from one folder to another (D4, table 9), the application takes several seconds to do this and this make it necessary to make Espresso aware that it should wait for this operation. With other tools one could simply wait for the proper text to appear on the screen.

However, Espresso had no problems with the situation where AUT would launch another application (D8, table 9), this action could very simply be stubbed out with the methods Espresso provides leading to no external activity being launched. Espresso also had no difficulty with the top part of the UI disappearing, since when searching in a list Espresso simply jumps to the location where the element that is searched is, instead of scrolling the list. This could of course be a problem in some situations, but in this instance, it only positive.
Amaze File Manager with Tau

Tau’s main problem at the time was that interacting with the navigation drawer was difficult. Both D2 and D3 (table 9) were problematic, the navigation drawer had to be opened through the “hamburger button” instead of swipe action like with other tools, and interacting with the navigation drawer’s contents was not straightforward. Tau provides ways to specify which area of the screen to search in for an element, but if a text is searched and the text happens to be both inside the navigation drawer and under it in the main portion of the screen (inside the area specified to be searched in), Tau could at a times click in the wrong place leading to the test case failing. This had to be worked around by scrolling the specific text out of the screen so there would be only one instance of it visible.

Another point of problem with Tau was that in this application the topmost part of the UI disappears when lists under it are scrolled, so this required one to occasionally emulate a small scroll upwards in a list to bring this top-bar back to visibility.

Tau offered a very simple way to handle the permission popups (D6, table 9). This was done through “reactors”, which can be specified at the start of the test case and it will do the action only when a popup screen appears on the screen.

Amaze File Manager with Appium

In the case of Appium, major difficulty was that resetting the application’s preferences and data proved to be problematic (D5, presented in table 9). Since Appium can’t access the source code of the AUT, one can’t simply make utility methods to clear the app’s database and preferences. Appium offers flags that can be set when it is initialized, which could for an example indicate that the application is to be re-installed on the device after every test run. This would naturally have an impact in the execution time of the test suite, which is already larger for Appium than it is for any other tool, as is further discussed in chapter 4.2. When reinstalling the application after every run one would also run into a problem with handling the permission model of Android M as discussed earlier.

Appium’s automatic application launch would also not work if Appium was to be used to grant the permissions as well, since it expects the activity of the application that is to be launched to be on the foreground after the application is launched, which would not be the case if the application would be asking for permissions since that pop-up window would be on the foreground.

Appium had the similar problem that Tau had with the need to scroll the screen up a bit at a times to make visible the topmost part of the UI.

As with the Notes application, it was required to assert if an item is not on the screen (D9). Again, this is something that Appium API does not offer.

Finally, as is the case with the other applications as well, Appium had its’ problems when searching for an item in a list (D7, table 9). UiAutomator code had to be written as a parameter to for Appium method to search for elements in the lists, although this is naturally simple to wrap inside a utility method.

After dealing with the issue of searching elements inside a list, interacting with navigation drawer was easy and worked without issues.
Amaze File Manager with UiAutomator

UiAutomator had similar problems as Appium with the topmost part of the UI disappearing, it was necessary to add some small scrolls upwards at some points of the test script to make it visible again. UiAutomator would have been the only tool besides Tau to reliably handle the permission popup that appears.

UiAutomator also had a problem with performing a “long click” on an object with the provided methods, to a point where it was necessary to replace the provided method call with a swipe action in place instead. UiAutomator would at a times fail to perform a proper long click on an object and would instead just click it. Using the swipe action proved effective but a non-intuitive replacement.

UiAutomator also required many wait commands to be used with this application, making the test code less readable and harder to write.

UiAutomator was quite reliable when handling the navigation drawer, some waits were necessary to wait for the navigation drawer to open fully, and there were occasional failures where UiAutomator would not properly wait for the navigation drawer before continuing with the test suite causing it to fail.

UiAutomator however had no problems with granting the permissions to the application since unlike other instrumented tests – Espresso and Robotium – it is not bound to the single application.

Amaze File Manager with Robotium

Robotium had similar problems with interacting with the navigation drawer as Tau had. If there is an element with the same identifier (text in this case) in the navigation drawer and elsewhere on the screen, Robotium would not return the correct element. This lead to the need to manually iterate through all elements on the screen and check for proper values of those elements.

Initially searching inside the navigation drawer was also a point of difficulty since Robotium would scroll the view behind the navigation drawer when searching it instead of the navigation drawer itself, but this was later addressed by the developers when an issue of this was opened on the project’s GitHub page.

Robotium also was not able to handle the test where an item would be added to the “quick access” portion of the application by opening a file multiple times. Since Robotium can’t handle applications besides AUT and the author did not find a way to work around this issue the test had to be dropped from the final test suite.

Robotium also had the problem of a very flaky opening of the navigation drawer (D2, table 9), which necessitated the use of the “hamburger” button on the top left corner of the device’s screen. For this reason, this method was used in every test suite, except for Espresso which provides its’ own way to reliably open up the navigation drawers. The button could be accessed for some reason through method clickOnActionBarHomeButton(), possibly named so because in early versions of Android in place of the “hamburger icon” there was often a button that would take the user to the home screen of the application. This however was not very clear to the author while browsing the API and it was discovered through browsing questions on the StackOverflow forum.
Finally, Robotium was not able to handle a list with the elements laid out in grids properly. The application provides a way to lay out the files and folders displayed to the user in a grid instead of list of items. This was one of the test cases built for the system and Robotium did not properly scroll the grid of items to the end to verify that the folder is displayed to the user in the grid view as well. This made it necessary to make an additional item in the grid view when running the tests to get comparable results to other testing tools, which is naturally not something one would want to do when building the test suite.

Robotium however had the same property as Espresso did regarding scrolling top part of the UI disappearing. Robotium, as Espresso, does not actually scroll the list down when searching for an element but immediately jumps to the location where the element is and interacts with it.

Notes about developing the test set for Wikipedia

The native tests were run with Appium, Espresso, Tau and UiAutomator. The tests that handle WebView were run with Appium and Espresso. Robotium was left out because it seemed to be very difficult to build the tests in a way that they would be run reliably many times in succession.

The official guidelines for automating Android GUI testing mentions disabling animations when doing automated testing (Testing UI for a single app.n.d.). The source speaks of turning off animations from the device settings, but in the case of this app there were also custom animations built that were not affected by the device’s settings. The source code of the application was thus modified slightly to remove the animations, which seemed to resolve many problems. This could of course be an unwanted edit in a real-world software project, but it could possibly be solved by utilizing different Gradle build variants (Configure build variants.n.d.).

Opening the navigation drawers for this application was mostly done in every case, except for Espresso since it provides dedicated method for reliably opening it, through emulating swipe methods.

Wikipedia with Espresso

In general Espresso was the most reliable of the tools. The test suite, when finished, was quite reliable with very few failing test runs. Per the nature of Espresso, the source code of the application had to be modified slightly further, a so-called Idling Resource was added to handle asynchronous UI updates (D2, presented in table 9). This approach is mentioned in the first party documentation of Espresso (Espresso idling resource.n.d.).

One of the navigation drawers proved to be somewhat problematic to work with (D2 presented in table 9) since the navigation drawer was built with some library that made it impossible to use the proper identifiers for the UI elements. It was possible to work around this but it required to not do the UI interaction in the way it is recommended by the first-party guidelines.

WebView (D1, presented in table 9) was somewhat easy to be manipulated with the help of the Espresso Web library that comes with the same repository as Espresso and
UiAutomator come with. Syntax of the library is a bit different than native Espresso’s is, but it was reliable at interacting with the WebView element.

Opening the navigation drawers was very easy with Espresso, as was the case at other applications as well.

**Wikipedia with Appium**

Appium was good at driving the UI of the Wikipedia application. It was easy to develop the test suite for the application, although there were some situations that came up. The first major problem was that when the tests were being developed interacting with the WebView was almost impossible. This was because at the time the ChromeDriver component that Appium uses was not updated yet to the version that was required to reliably switch to interacting with the WebView portion and the native portion of the application on Android API level 23 and above. This of course is not fault of Appium itself but the third party that develops the component. After this issue was settled Appium was very good in interacting with the WebView part of the application.

However, it was quite difficult to develop a reliable way where Appium would always wait until a list of articles was presented by the application before continuing (D4, presented in table 9). In the same vein, waiting until an article was fully opened was at a times quite flaky with Appium, and this is later expanded upon in the chapter about Reliability.

Appium also had a problem at reliably opening the navigation drawer. For this reason, it was necessary to use the “hamburger icon” to open the main navigation drawer so the test suite would be somewhat reliable. It is however quite possible that part of the failure percentage, discussed later in chapter 4.7, is because the other navigation drawer in the application (table of contents in articles) was opened through swiping as was done with other tools.

As was with the other applications as well, when searching in lists the code Appium requires is quite messy (D7, presented in table 9). Finally removing text from the search bar in the application did not function properly, doing it the way it is presented in the documentation would take up to a minute to accomplish, so a workaround had to be implemented to keep the execution time of the test in reason.

**Wikipedia with Tau**

Tau was not able to handle the WebView element of the application reliably, but proved to be quite easy to work with otherwise. The tests required the author to write in waiting times to significant portion of the interactions with UI. After these were added Tau was reliable enough to properly wait for the different asynchronous interactions (D2, table 9) with the application. Tau would occasionally simply fail to find the main search bar of the application though, leading to some tests failing right at the start for no good reason.

When attempting to interact with the WebView element of the application Tau would occasionally correctly identify the different elements inside the WebView and could click on them, but this did not work correctly every time. This was since for some reason elements inside a WebView present as their ContentDescription (attribute of an UI element in Android) the text of the URL they are pointing to. It also seems that Tau was simply not able to identify all the elements inside the WebView. Tau also does not “officially” support interactions with the WebView, or at least mentions of this in the
documentation were not to be found at the time. For this reason, it was decided to not to attempt to bring Tau to the WebView comparison.

**Wikipedia with UiAutomator**

UiAutomator proved to be similar in some ways to Tau, it had similar difficulties when interacting with the WebView. It could identify some of the elements inside it at a times, but not reliably.

UiAutomator however was very difficult to work with when trying to implement proper waits to wait for the search results and articles to load in the application (D4, table 9). This lead to very flaky tests runs, as is discussed further in the chapter Reliability. UiAutomator would also at a times simply fail to find the elements in a search results list. It was attempted in several different ways to implement this but none proved to be particularly effective.

For some reason opening the search bar while inside the article was difficult for UiAutomator, this would very often be a source of a failed test with this tool. It is unclear if this is related to UiAutomator not waiting long enough for the article page to load before attempting to click the search bar, or due to some other reason.

**Wikipedia with Robotium**

Robotium proved to be completely ineffective at driving this application. It would successfully do some interactions with the application but then it would for some reason freeze up completely the UI of the application, making it so that the UI is not updated at all any more. This would lead to the test set completely failing. It is not clear if this is because Robotium messes up the AUT’s UI thread somehow. Adding in manual arbitrary sleep periods helped at some situations but made no difference in others.

As with UiAutomator and Tau, Robotium seemed to pick up elements that were not visible to the user at the time, Robotium attempting to interact with those elements and then failing since the UI had not been updated. Robotium’s documentation mentions that it should be able to interact with the WebView element, but this did not function at least in the way it was described in the documentation. It was however possible to address the elements inside WebView in the same way as with Tau and UiAutomator – through ContentDescription. However, as discussed earlier, addressing elements through ContentDescription is not very clean in Robotium, requiring much more complex code than addressing by text or by ResourceID (an attribute that most UI elements in Android have).

When searching for a text Robotium took a very long time to scan the page. At worst, Robotium took good 1 minute to scan the page for the text that was searched for. Robotium also had a bad habit of clicking the very first text element that contains the text that should be searched for, instead of searching for an element with the exact text that is given. This lead to the need to build regexes to search exactly the correct text, which could make the test code more difficult to grasp and to write.