# A review of available testing methods for mobile applications and potential for improvement

September 2016

MSc Software Engineering

**UNIVERSITY OF SOUTHAMPTON**

# ABSTRACT

Faculty Of Physical Sciences and Engineering

Electronics and Computer Science

Thesis for the degree of MSc Software Engineering

**A review of available Testing Methods for Mobile Applications and potential for improvement**

This study assesses the state of the testing tools and methods offered to mobile application developers today by evaluating the testing tools and methods offered by the Android OS, an open-source OS which ships on the majority of mobile devices today. An attempt to identify gaps in the testing tools and methods offered is made, followed by an attempt to close those gaps by proposing technical solutions and creating new software tools, aiming to improve the quality of testing for mobile apps and make testing more time-efficient for the developer and the tester. In addition, a series of experiments were performed to judge whether the new solutions and new software met the goals that had been set, and whether the gaps in functionality have been successfully covered.

The study focuses on areas which are unique to mobile applications. Those 5 areas are 1) GUI testing for mobile GUIs (which are different from desktop/laptop GUIs), 2) location-services testing for highly mobile devices, 3) security testing with a focus on the peculiarities of mobile OSes and mobile application, 4) testing for network switching between different kinds of network (WiFi and Cellular) and finally 5) estimating power consumption of a mobile application running on a mobile device.

A particular emphasis is placed on assessing the current degree of testing automation for those five areas. In other words, the aim of this study is to find out what degree of testing automation and testing coverage the currently-available tools offer to the developer, and whether testing automation and testing coverage can be improved (further) with custom-written software, and to what degree.

# Table of Contents

# List of Tables

# List of Figures

# List of Accompanying Materials

- Zip file with source code (a copy of the zip file can be downloaded from: https://

# Chapter 1: Introduction

## 1.1 Background

### 1.1.1 General Statement

Up until the first half of the previous decade, mobile applications used to be simple both in terms of complexity and functionality. As a result, testing used to be done manually and was relatively limited in nature. However, as smartphones evolved rapidly during the last decade (both in terms of hardware as well as in terms of operating systems), the complexity of mobile applications increased. Mobile applications are now approaching the complexity of some traditional applications, while at the same time having their own unique needs when it comes to what needs to be tested. Also with more and more security critical tasks are being performed using mobile apps, as for example payments using NFC connectivity or mobile purchases, the topic of security and prevention of information leakage becomes of importance. In addition, since a great percentage of the available mobile applications are meant to be used as either entertainment (content consumption) applications, the ease-of-use, adequacy and correctness of a mobile application's GUI also become of significant importance and a possible differentiator from competing solutions, hence to a large degree determining the potential for the commercial success of the application.

For the purposes of this dissertation, the focus will be on the Android operating system, as it is by far the most dominant operating system in terms of market share and is also an open (and mostly open-source) operating system that can be studied in a greater depth than a closed operating system could be.

### 1.1.2 Unique testing needs of mobile applications (the need for specialised testing tools)

When developing a mobile application, there is a number of factors that need to be taken into account that is not normally taken into account in a traditional application, or which contain peculiarities not found in desktop applications.

**i) GUI testing**

GUI testing is of particular importance for mobile applications, for reasons mentioned in the introduction. However, most GUI testing of mobile applications still happens manually, with application developers simply loading the application on an emulator or a physical device and "trying it out". In fact, only 34% of developers employ automatic testing of any kind in their mobile applications [JOORABCHI]. However, the increasing complexity of mobile applications means that the application must often go through multiple debugging and test cycles before it can be declared "release candidate", which means an ever-increasing amount of labour that needs to be done for manual testing. In addition, fragmentation of screen sizes and resolutions in mobile devices increases the labour needed to test the GUI even more. Testing on one combination of

screen size and aspect ratio isn't enough, as GUI elements may overlap or be misaligned when rendered on other screen types. As the variety of screen resolutions and aspect ratios offered by Android devices increases, it's essential that GUI testing shifts to a more automated approach, which should result in significant time savings.

ii) **Location services**

One other factor is location services input. Mobile applications are typically expected to receive input from different kinds of location services (such as the GPS receiver, WiFi geolocation service, or cell-tower location service), while location input at a constant rate is expected for some kinds of some mobile applications, such as navigation applications. The unique challenge imposed by that requirement is that most traditional testing tools do not allow a developer to "inject" a series of inputs to some in an automated manner. In order for automated testing to be successful, the ability to inject input in varying frequencies and with some erroneous values must be offered to the tester, so the application's ability to handle all possible input can be tested [JOORABCHI].

**iii) Energy consumption**

Energy consumption is not always a major concern when developing traditional applications, despite the fact those applications can also be run on a laptop operating on battery power. However, energy consumption on mobile devices is paramount. Modern mobile OSes allow applications to run as background tasks, and hence applications should not excessively drain the phone's battery when operating either in interactive mode or running as a background service. However, the current set of developer tools offer no way to measure the total amount of energy used by the application (that is, accounting for energy used by the CPU, GPU, and peripherals such as Bluetooth, WiFi, and 3G/4G networking). Methods to estimate energy usage for an application have being proposed, but none of them has materialised in an actual product [MUCCINI].

**iv) Change of network testing**

Another, not often taken into account peculiarity of mobile applications is a change of network. A mobile application is expected to change networks often, such as changing between WiFi networks or switching from WiFi to 3G/4G and then back to WiFi, as the user moves around. As a result, it's important to test that any interruptions or changes in internet connectivity are handled by the application gracefully. In order to achieve that, it's essential that a way to simulate such events is provided by the testing tools.

**v) Security Testing**

Older smartphone operating systems (such as the first versions of Symbian OS) had very little security to protect from malicious apps, with apps being able to simply write into each other's folders and access practically every resource in the system without restrictions. However, modern mobile operating systems have built-in protection mechanisms to protect from malicious apps. In particular, a permission system has been implemented in Android to protect security-sensitive operations such as the ability to write to user storage and access to networking features like Bluetooth, or other features such as the telephony stack. However, sandboxing cannot prevent security breaches caused by exploits injecting malicious code in the application's running code, since the application already has been granted permissions. Essentially, any malicious code

injected in the application's code execution will have access to any permissions the legitimate application code has. This is of particular importance for two reasons: First, because some applications have access to high-privilege permissions, such as the ability to take a screen capture the screen of other apps, draw over other apps and change the operating system's settings. Secondly, the limited hardware resources of the average smartphone (compared to a desktop computer) and battery life requirements mean that the vast majority of smartphones do not run an antivirus scanning every piece of code that is being executed in memory. For those reasons, it's important that applications are thoroughly checked against bugs that could cause exploits from malicious input. For that reason, fuzzing testing must be present as a feature in every mobile application testing tool.

However, no widely-accepted fuzzing tool exists for either iOS or Android [MAHMOOD]. In addition, applications are expected to run on older versions of the operating system, which may have security vulnerabilities or provide operating system components which contain vulnerabilities. An application should be able to detect those vulnerabilities and either work around them by using different libraries or disable certain sensitive features.

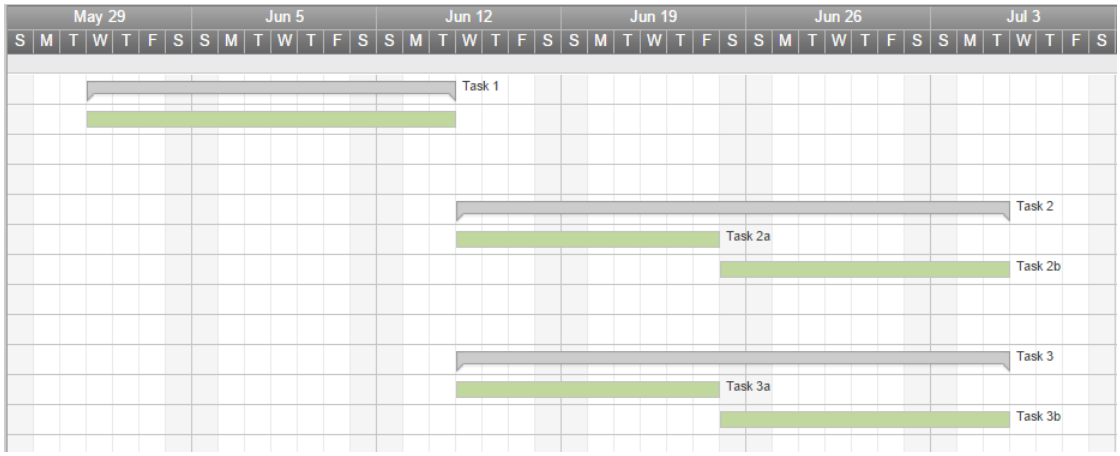
## 1.2 Project aims and objectives

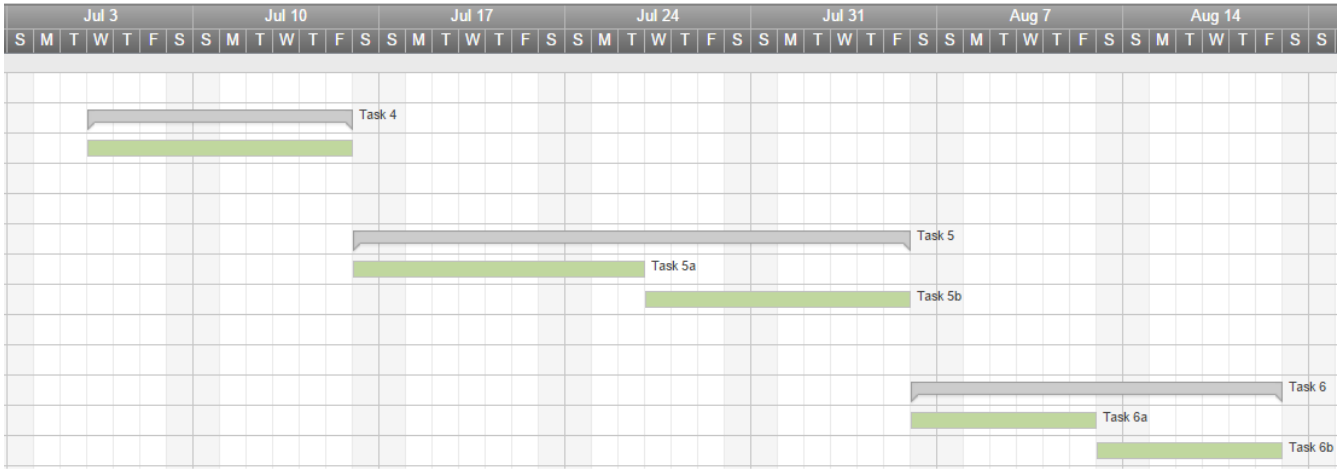
The aim of this project is to:

1) Provide a review of the existing tools which are available and intend to assist the task of automated testing of mobile applications for the Android operating system. The existing tools will be evaluated according to their ability to cater to the peculiarities of mobile applications and their testing needs.
2) Identify areas and needs which are not adequately addressed by existing tools, attempt to propose improvements and then attempt to design novel, custom-created software that will cover those gaps.
3) Measure the savings in testing time and measure the gains in testing effectiveness (in testing coverage) achieved with the new tools.

## 1.3 Project Plan

When the project was started, the project plan stood as follows (Gantt chart):




Description of the tasks:

Task1:

Task 1: Familiarization with development methods for mobile applications, their SDKs and testing tools. A solid understanding of how a mobile application is being developed and the software development lifecycle of a mobile app is required. An attempt to create one or more mock applications will be made, with the software development lifecycle recommended by the OS vendor and best-practices guides being followed.

Task 2:

Task 2a: Familiarization with sensor input APIs and location services. Familiarization with current testing tools by development of mock applications accepting sensor input.

Task 2b: Development of method to “inject” sensor input and location input. This can be achieved by two methods: One method is by examining the source code of any current unit-testing tools, and attempting to modify them for the purpose, a second methods is to develop new software for the purpose from scratch.

Task 3:

Task 3a: Familiarization with common GUI-building techniques. Familiarization with current testing GUI tools.

Task 3b: Development of tool for automated testing of GUI of mobile applications. Research into existing tools and into the general principles of automated GUI testing is needed, which will allow for understanding of common automated GUI testing techniques. A tool putting that knowledge into use can be developed.

Task 4:

Task 4: Research into how simulate network changes, and how it can be integrated into testing software.

Task 5:

Task 5a: Research into security testing for applications in general and mobile applications. Research into existing security testing tools for mobile applications. Research into application privacy

Task 5b: Development of software for automated fuzzing testing. In order to fuzz all possible inputs as well as the GUI, tasks 2b and 3b must have been completed beforehand.

Task 6:

Task 6a: Find a method to measure energy consumption of an application running in a mobile device. A method to do that could be to approach the issue experimentally, which is make a sample application using the different components of a mobile device (such as CPU, GPU, networking, storage) in different combinations at each time. For example, only the CPU, only the GPU or both. Then a mathematical formula can be derived based on those experimental observations, which would allow the estimation of energy consumption of applications, as long as their CPU usage, GPU usage, networking usage and storage usage (read/write time) is known for those applications.

Task 6b: Try and develop software that automatically calculates energy consumption.

In practice, each task took 2 days more than budgeted, with the exception of task 1 which was completed on time. This led to a delay of about a week and a half. However, dude to the 3 weeks of reserve time left at the end, it was not a problem and didn't threaten the completion of the project.

The dissertation text was written in parallel with the research and software development work, with the finalization of the text and formatting/presentation tasks taking place the last week (29 August 2016 to 1 Thursday 2016).

# Chapter 2: Review

## 2.1 Relevant literature

Literature used for this dissertation includes academic textbooks, documentation and IEEE research papers. In particular:

- The *Software Test Automation* textbook by Mark Fewster and Dorothy Gragam (Addison-Wesley) and *Testing Computer Software* textbook by Cem Kaner, Jack Falk and Hung Quoc Nguyen (Wiley) were used to gain a theoretical foundation for software testing methods and automated software testing methods.
- The official documentation found in *developer.android.com* and *source.android.com* was used for API reference, for official tools and SDKs reference and as a source about the internals and architecture of the Android OS, its frameworks, services and utilities.
- Various IEEE research papers (outlined in the List of References section at the end of the document) about mobile application testing were used as references for the five specialised topics of this dissertation.

References to literature are put in brackets [], with the name of the reference inside the brackets. The reader can find the name of the reference in the List Of References section and next to it the source itself.

## 2.2 Relevant technology

This dissertation will mainly use the version of Android OS provided by Google Inc., without however being dependent on any proprietary Google services and APIs. The version of Android OS provided by Google Inc. was chosen because it is the one included in the Android Studio SDK and the Eclipse ADT tools.

### 2.2.1 Creating Android applications

Android applications are usually written and compiled with the Android Studio SDK or the Eclipse SDK with the ADT plugin. Besides offering a full compiler and IDE, those SDKs also generate necessary “boilerplate code” that would otherwise have to be written manually by the developer.

### 2.2.2 Running Android applications

There are two ways to run the compiled application: The first method is to run it on an “emulated” Android device, which essentially is an emulator of mobile devices with a complete image of Android OS pre-installed. The developer can create virtual devices running an image of Android OS The second method is to run the compiled application on a physical Android device connected to the computer via a USB cable.

### 2.2.3 General overview of the Android OS

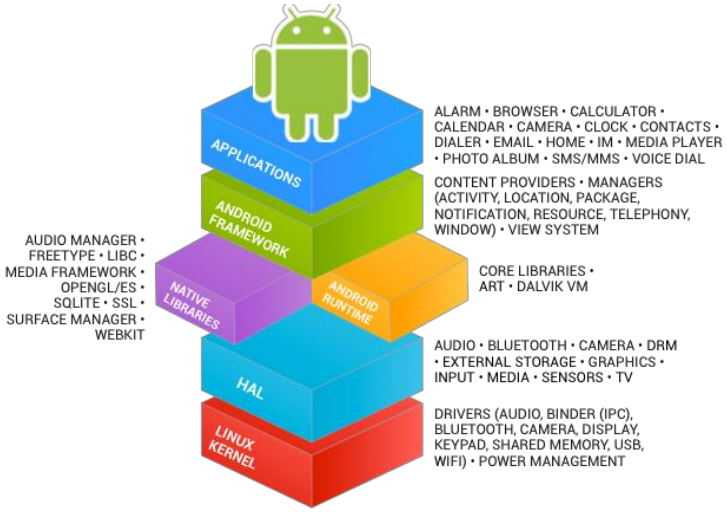



Figure 1: The Android OS architecture (copyright source.android.com)

The Android OS runs on top of a (generic) Linux kernel and HAL. On top of those runs the “Android Runtime”. Most Android applications are written in the Java programming language and then compiled into “dex bytecode”, which is a virtual instruction set architecture (virtual means it is not actually implemented in hardware). This instruction set architecture is a modified version of the Java VM ISA. The purpose of the Android runtime is to allow dex bytecode to run on the physical ISA of the device (which is usually ARM, x86 or MIPS64) by recompiling it. The Android Runtime is implemented either by a JIT-based virtual machine (called Dalvik VM, found on older devices) or an ahead-of-time recompiler (called ART, found on newer devices). In both cases, the Android Runtime also implements the core libraries of the Java programming language. Applications that run on top of Android Runtime are usually called “Dalvik apps” or “dex apps”. The advantage of Dalvik apps is that the developer can target multiple physical ISAs using only one application package. In addition, Dalvik apps have access to all the features of the Java language (such as garbage collection and the java core libraries) [ANDROID].

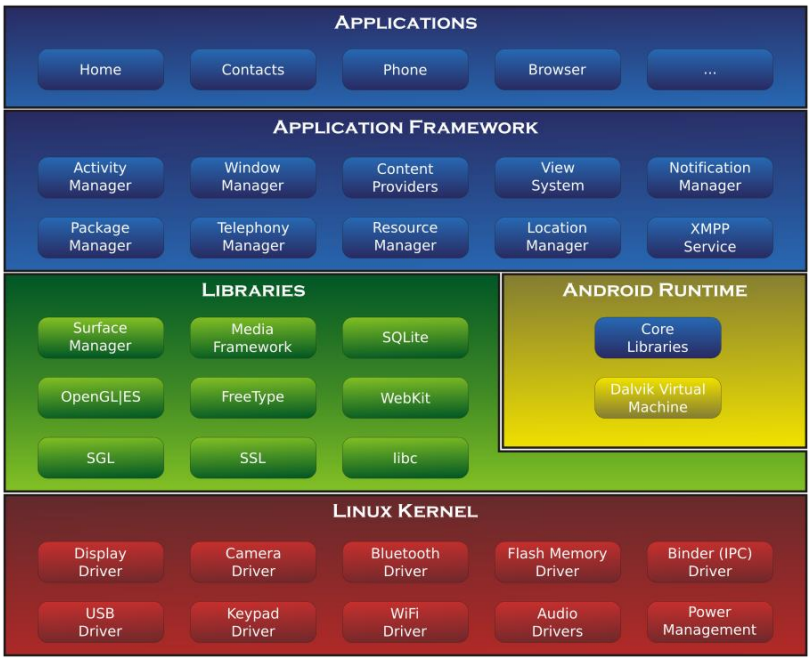



Figure 2: More detailed view of the architecture and components of Android (copyright androidteam.googlecode.com)

Another type of Android apps is “native apps”. Native apps are usually developed in C or C++ and are compiled into binaries targeting some physical ISA (ARM, x86 or MIPS64) which of course run directly on the CPU without the need for Android Runtime. Essentially, native apps bypass the Android Runtime in order to avoid the overhead it imposes and run faster with less memory consumption. This is necessary only for a specific subset of applications, mainly computationally intensive applications. 3D rendering applications is an example of such category of apps. Another set of applications which is usually developed as a “native app” are video encoding and decoding applications, which have to run directly on the CPU in order to take advantage of the SIMD instructions of the physical ISA (SIMD instructions massively accelerate video decoding and encoding). Native apps have access to a set of libraries which have also been compiled into physical ISA binaries, called “Native libraries”. Native libraries provide libraries for 3D rendering (OpenGL and OpenGL ES libraries), libraries for 2D screen rendering (Surface Manager) and font rendering (Freetype), among others [NATIVE]. Native apps have executables with a .a extension.

It is also worth noting that Dalvik apps can also have access to functions offered by Native libraries, using the JNI interface of the Android Runtime.

The fact the Android OS supports two kinds of applications results into certain complexities for the developer. The most obvious complexity is that he must decide on one type of application (Dalvik or native). Another is that there are two kinds of SDKs, one for each type of application. Android Studio and the Eclipse ADT plugin are SDKs used for developing Dalvik apps, while the “Native Development Kit” (NDK), another SDK offered by Google, is used for developing Native apps.

What is of interest in the context of this dissertation is that this distinction also creates certain complexities when it comes to the testing of applications too. For example, jUnit is available for Dalvik apps but is not available for Native apps. Generally, any kind of testing framework designed

as a Java package is unsuitable for testing Native apps, and any testing framework designed for C and C++ code is unsuitable for testing Dalvik Apps.

#### 2.2.3.1 The Android framework

Android also includes a set of shared libraries called the “Android framework” (written in Java) available to Dalvik apps only. Those libraries provide a set of APIs that helps the developer quickly develop Android applications. For example, they provide APIs for the creation of UI elements (buttons, menus, text fields etc), the creation of windows, and also provide support for Activity management, Service management and sending and receiving Intents (an in-app and inter-app messaging service). An Android application is usually developed as a set of Activities (processes that the user interacts with), Services (processes that run in the background), and Intent receivers (processes that receive information from other apps or events and are launched when this happens) [FRAMEWORK]. It is not mandatory to develop Dalvik apps using the Android Framework APIs, although it is generally recommended. Of course, the official SDKs (Android Studio and Eclipse ADT) produce code that makes use of the Android Framework APIs.

As a final note, it is worth saying that “hybrid” application packages, consisted of part dex, part native code can also exist. Of course, the ability to have one package targeting multiple physical ISAs doesn’t exist with “hybrid” apps. However, “hybrid” apps are useful when you have an application such as a full-fledged media player and some parts of the application need the efficiency of native code (mainly the encoder and decoder part) and some parts of the application need to access the Android Framework (for example, to easily create the options menu or easily create the “share to social network” functionality).

An application which doesn’t need the efficiency of native code because it doesn’t need to do computationally intensive 3D rendering or video encoding/decoding (such as a photo viewing application, or a messaging application or a simple 2D game) will usually be a Dalvik app (not native or hybrid).

### 2.2.4 APK files

Android applications are packaged in apk files, which contain the executables, resources (such as strings or images) and arbitrary files. The Android Package Manager, which found in all versions of the Android OS, can install applications packaged in .apk files and uninstall them. It is not possible to just send a plain executable to an Android device and run it because the “execute” permission is always disabled for the folders on which the user has write access on, and the “execute” permission setting cannot be changed by the user. All applications must be packaged into an apk file and subsequently installed by the package manager (which runs with root privileges) inside a folder with executable permissions (/data/app), to which the user doesn’t have write access to (only the system) [APK].

The executables inside an apk are files with a .dex extension for Dalvik apps, and a .a extension for Native apps. The apk of a hybrid app will include both types.

### 2.2.5 The Android debug bridge

Android application development is mostly not self-hosted. The application runs on a physical device or an emulator running the Android OS, while the writing and compiling of the code usually happens on another OS (most of the times on a laptop or desktop running a different operating system).

#### 2.2.5.1 Adb daemon, server and client

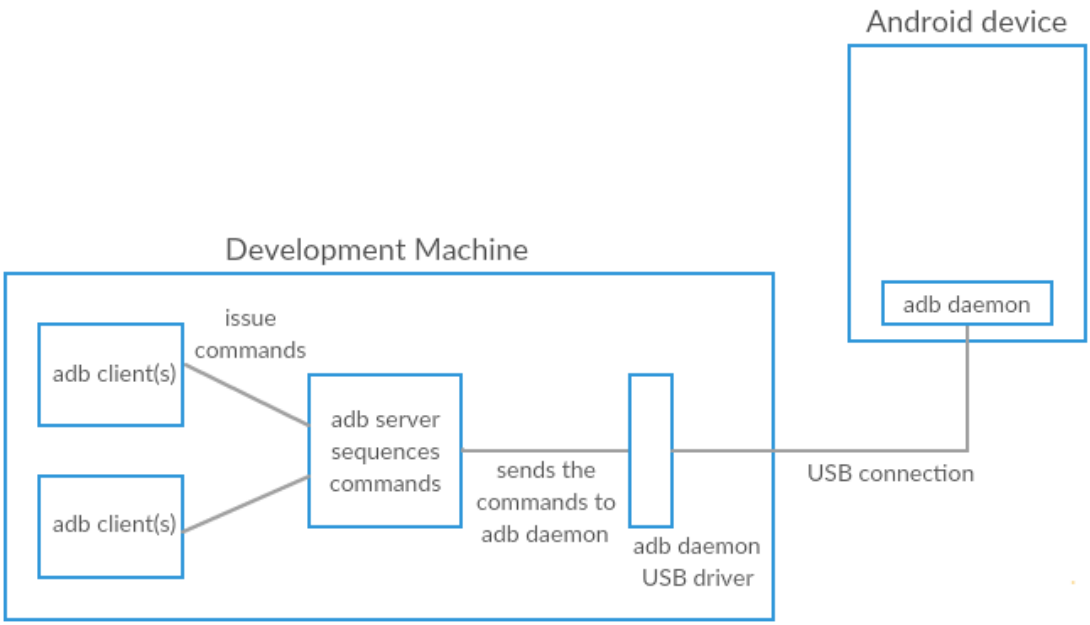



Figure 3: The Android Debug bridge (own work)

Because of the not self-hosted nature of development, a way to quickly send and install the apk file to the Android-running device (the mobile device) as well as quickly issue debug commands while the application is running has to exist. Indeed, Android provides a mechanism called the Android Debug Bridge. It is consisted of the following components: [ADB]

adb daemon:

The adb daemon runs on the Android device and performs the actual app installation and debug commands on the device. The daemon is launched by Android OS after enabling the “USB debugging” option is system settings. When using the official emulator or when having a physical device connected to the development machine, the adb daemon USB driver reserves two TCP ports in the *localhost* address on the development machine, starting from port numbers 5554 and 5554 for the first mobile device to connect to the development machine. The adb daemon reserves ports on the development machine with the help of a driver running on the development machine, and the same driver also manages connectivity between the two devices. The first port number is reserved for the “console” debug functionality, while the second is reserved for issuing debug commands and apk installation.

adb server:

On the development machine, an adb server runs in the background which listens for new debug commands from adb clients. The purpose of the server is to "sequence" debug commands which can be issued from multiple adb clients and run them one-by-one (although in most cases only one client at a time is launched). The official SDKs provide adb server and client versions for Windows, MacOS X and most variants of Desktop Linux. The adb server is launched from an executable binary in the development machine (for example, in the Microsoft Windows version of the adb server, the binary is found in *%USERPROFILE%\AppData\Local\Android\sdk\platform-tools\adb.exe*). Launching the adb server can be accomplished using the *adb start-server* command. Once launched, the adb server scans the TCP ports in the 5554 to 5585 range to find any running adb daemons (in other words, find connected devices).

adb client:

The adb client also runs on the development machine. The adb client is launched by the same binary the adb server is launched. It is also worth noting that if an adb client is launched and doesn't find a running adb server, it will launch one automatically, hence launching the adb server manually is rarely necessary (only in the very rare case the adb server unexpectedly quits). When a debug command needs to be issued, an adb client is launched by the developer. This is done by opening a shell (command line) and running the adb binary with some parameters (presented below). If more than one devices are connected, the -s flag allows the developer to choose to which device the commands will be directed to (using its port number to identify it). The debug command is sent to the server and after that, it is the server's responsibility to send the command to the daemon corresponding to that device.

The Android Debug Bridge provides several features to the developer. Of them, important to application testing are:

**Application installation:**

*adb install*: The install command can install an application into the target device. The two official Android IDEs (Android ADT plugin for eclipse and Android Studio) will execute this command automatically when selecting the "compile and run" option (provided a physical device has been connected or a virtual device has been created in the emulator). The user can select the target device if there are more than one. Alternatively, the command can be executed manually by typing the *adb install <path_to_apk>* command on the command line.

**Data transfer:**

*adb pull*: The pull command copies a file from the Android device to the computer the adb client is running on.
*adb push*: The push command copies a file from the computer the adb client is running on to the Android device.

**Shell:**

*adb shell*: The shell command opens a remote linux-like shell on the Android device, which allows the developer to issue shell commands [ADB SHELL]
*adb shell <command>*: Opens a remote shell, executes the command and closes the shell.

**Other commands**

Other adb debug commands exist which are useful for testing (for example adb commands useful for location services testing), and will be presented in the following chapters of this dissertation.

### 2.2.6 Existing testing tools and technology

One of the aims of this dissertation is to analyse existing automated testing tools that are available to the developer. Due to the fact there are two kinds of applications (Dalvik and Native), existing tools will have to be examined for each case separately.

#### 2.2.6.1 Testing for Dalvik apps

##### 2.2.6.1.1 Unit testing:

Unit tests are meant to test a specific class. Unit tests save development time, as the developer doesn't have to wait for the application to be fully compiled and run. Since Dalvik apps are written in Java, jUnit 4 is the preferred unit testing framework. An implementation called JUnitRunner is provided. jUnit tests are performed in a manner similar to ordinary Java applications [UNIT TESTING].

More specifically, the JunitRunner classes are imported, and the method that will be executed before the test is written by the developer, which in most cases is the test setup method (which is usually a piece of boilerplate code). Then the method which actually performs the test is written, and the method which will be run after the test has been completed (that is, the method that is being tested has finished) should be written. As with the @before method, the after method is usually a piece of boilerplate code required by the testing framework.

```
import android.support.test.runner.AndroidJUnit4;
import android.support.test.runner.AndroidJUnitRunner;
import android.test.ActivityInstrumentationTestCase2;

@RunWith(AndroidJUnit4.class)
public class CalculatorInstrumentationTest
        extends ActivityInstrumentationTestCase2<CalculatorActivity> {

    @Before
    public void setUp() throws Exception {
        super.setUp();

        // Injecting the Instrumentation instance is required
        // for your test to run with AndroidJUnitRunner.
        injectInstrumentation(InstrumentationRegistry.getInstrumentation());
        mActivity = getActivity();
    }
```

```
    @Test
    public void typeOperandsAndPerformAddOperation() {
        // Call the CalculatorActivity add() method and pass in some operand
values, then
        // check that the expected value is returned.
    }

    @After
    public void tearDown() throws Exception {
        super.tearDown();
    }
}
```

Copyright of source code fragment above: developer.android.com

One general issue with unit tests is how to handle classes which depend on other classes (packages), as those other classes are not compiled and will not be running during the test. In the context of Android development, this becomes an important issue, since lots of classes in an application depend on the Android Framework. For example, if a method of a class needs to receive an *Intent* (an Intent is a kind of message sent between app components and is provided by the Android Framework) and a test for that method must be written, then we must have some way to trigger the code behavior that is normally triggered by an Intent message. This issue is solved with mock objects. Mock objects are fake objects which mimic the behaviour of real objects just for the purposes of a test. In our case, we must create a mock *Intent* message. For that purpose, Google recommends using the Mockito framework. The Mockito framework provides an easy way to create mock objects for everything inside the Android Framework [MOCKITO]. In our case, the following code can be written:

```
Context context = Mockito.mock(Context.class);
Intent intent = MainActivity.createIntent(context, "query", "value");
```

A *Context* is the base class for the most fundamental services of the Android Framework (including the *Intent* messaging service) and an object of *Context* type is always created when the application is launched. However, such object is not available when running a unit test because the application is not running, and for that reason, we must create a mock object that mimics the behaviour of *Context*-type objects, by calling Mockito.mock(Context.class).

After that, we can easily have some other class (sich as the MainActivity class) create an intent, with some content the “query” and “value” strings as the content of the Intent message (for the case of the code above). This can be done by writing a static method for that very purpose in the MainActivity class. The behaviour of the Intent message will be mocked by the fake Context.

As is the best practice with mock objects, we must make sure that the mock object actually behaves in the way we want it to. For example, it must not be null. So, the following code is also needed.

```
assertNotNull(intent);
```

It is also a good idea to verify the contents to the message (“extras” as they are called by the Android Framework) and make sure they are not null too and they have the content we want them to have.

```
Bundle extras = intent.getExtras(); //get the contents of message
assertNotNull(extras);
assertEquals("query", extras.getString("query"));
assertEquals("value", extras.getString("value"));
```

With jUnit and Mockito, comprehensive unit tests can be written for every Dalvik application, with very little to be left desired by the developer.

2.2.6.1.2 Espresso UI tests:

The Espresso UI testing framework is a way to create white-box UI tests for the entire application. White-box UI tests are tests where the developer needs to have knowledge of (and access to) the application’s source code [ESPRESSO].

The Android Framework provides numerous widgets (UI elements) called “views” (all UI elements are in fact children of the View class). The Espresso framework allows the developer to find a particular view inside the application. This practice is called “matching”. The developer can find a UI element (view) by its class name, by its unique id (“R.id” as it is called), by the text it contains (if it contains text) or by providing a content description for that UI element. Obviously, searching by R.id is the most reliable method.

For example, in order to find a button which has an ID of “my_button”, the developer would have to write the following code:

```
Espresso.onView(withId(R.id.my_button));
```

The onView method will return a reference to the object that was found.

With having access to the object, a number of actions can be performed on that object, which would mimic actions performed by users. Actions such as clicking, swiping, typing text or opening a link can be performed, or whatever other actions a certain widget allows.

For example, the following code finds a button with ID of “my_button” and clicks on it.

```
onView(withId(R.id.my_button)).perform(click());
```

With Espresso, the developer can create entire UI-test classes with relative ease.

#### 2.2.6.2 Testing for native applications

##### 2.2.6.2.1 Unit testing

Since no widely-accepted unit-testing tool exists for the C and C++ languages, Google developed the Google Test testing framework [GOOGLE TEST]. It aims to resemble the testing philosophy of jUnit. Testing with Google Test involves creating a test fixture class, which includes methods for setting up a test (in other words, a method which is called before the test), and a method which will be called after the test has completed. Google test is provided as a header file (gtest.h) and the test fixtures must inherit from the *testing::Test* class. The test themselves are in the form of:

```
TEST (FooTest, Testname) {
   EXPECT_EQ (“expected_value”, FooObj.FooMethod(parameters));
}
TEST (FooTest, TestName2) {
   ASSERT_EQ (“required_value”, FooObj.FooMethod(parameters));
}
```

The tests can be found anywhere in the code. It is worth mentioning that the identifiers in caps are in fact macros. The macros will expand the test to valid C++ code. FooTest is the test fixture and TestName is the name of a particular test. The Expect_EQ will call the method it takes as argument (FooMethod) and check for equality of its return with a prescribed value (note: there are more types of checks than equality). Expect_EQ will notify the developer if a check fails but will not consider it a fatal failure. In contrast, Assert_Eq will require the check to be a success otherwise, the failure will be considered fatal and the testing will stop.

The developer can run all tests in the main() function with the following code:

```
int main(int argc, char **argv) {

 ::testing::InitGoogleTest(&argc, argv);

 return RUN_ALL_TESTS();

}
```

#### 2.2.6.3 Testing for all kinds of applications (Dalvik, Native and hybrid apps)

UI automator:

The UI automator is a tool intended for “black-box” testing, where the implementation details of an application are not known [UI AUTOMATOR]. It provides more high-level actions to the developer (compared to Espresso), such as pressing the home button (of the device), or finding UI components which match a certain description such as the icon name (in our case, it is the “Apps” icon on the home screen) and performing a click on the button and then waiting for it to bring up the launcher.

```
// get device to perform test on (boilerplate code)
mDevice = UiDevice.getInstance(getInstrumentation());

mDevice.pressHome();

//press Apps button on home screen
UiObject allAppsButton = mDevice
        .findObject(new UiSelector().description("Apps"));

// click and wait
allAppsButton.clickAndWaitForNewWindow();
```

While the UI automator is a well-designed framework that helps the developer perform black-box testing on some apps, it is not without considerable limitations. For example, it provides no way to capture actual input from a real user performing real actions on a device (emulated or physical). This is a significant limitation since a test which completes successfully for a mocked action (for example, a mock swipe motion) may fail on some real action of the same kind (because for example, the swipe might have been too slow and the particular application does not handle slow swipes well). Also, having to manually write code to perform even the most trivial UI actions can prove time-consuming and tiring for the developer.

In addition, some applications of the Native kind do not use UI elements to draw their user interfaces, but instead use OpenGL(ES) line-drawing to do so. Most Native games and some Native applications (such as the “Kodi” application) are examples of this. The reason why some developers prefer this approach is because it allows them to have a single codebase of C++ and OpenGL(ES) drawing code which they can port to different platforms (for example Android and iOS) with minimal alterations in the drawing code. This is considered an advantage for applications where the drawing code comprises a significant portion of the application, such 3D games or multimedia applications with rich UIs. When testing such applications, UI automator can do little more than press the home and back keys of the device, as it relies on clearly defined UI elements to create its tests (in fact, some of those applications have UI elements that aren’t standard icons and named buttons but arbitrary 2D polygons). There is also the issue of applications which would have poorly-named or foreign-language descriptions for their UI elements, and although testing them with UI automator is theoretically possible, it would be impractical without access to the source code.

With this in mind, it’s clear that UI automator is more “grey-box” than “black-box” testing, as some knowledge about the application to be tested is needed, and the application has to be built in a certain way to be testable.

#### 2.2.6.4 Other types of tests:

While UI testing is one of the most common cases where black-box tests are used, some other kinds of testing can benefit from black-box testing. Such areas include testing of location services, testing of network changes and network interruptions and fuzz testing.

### 2.2.7 **The need for Black-box testing:**

It is obvious that a true Black-box testing tool is needed for Android applications. A tool which would not need to know anything about the application that is running, besides the fact it is an Android application running on Android OS. In the following parts of this dissertation, I will describe an attempt to create a prototype of such a black-box testing application for UI testing, location services testing, testing of network changes and network interruptions, and fuzz testing. Additionally, an attempt to measure the gains in efficiency compared to manual testing will be made. A separate part of the dissertation will concern itself with an attempt to measure the power consumption of Android applications on a variety of devices, also in a black-box manner.

# Chapter 3: Research Methods

## 3.1 Exploration of technical approaches

Since the testing tool has to be completely black-box based, only the minimal assumptions should be made. For example, the testing tool cannot be a library or framework which "plugs" into the application or which assumes the existence of objects created by certain frameworks, such as the Android Framework. However, it is reasonable to assume that the application will be running on Android OS and that the version of Android OS it is running on will have the Android Debug Bridge (since the Android Debug Bridge is required by the Android Compatibility Definition and any OS image lacking it cannot bear the Android trademark). It is also reasonable to assume that the device will offer a display which is pixel-addressable and will also feature some way to click (or tap) on the pixels of the screen, and that some kind of keyboard (virtual or physical) will be present. Finally, the device may feature a GPS receiver (and location services in general) and network capability, but it is not a requirement. The application itself is assumed to be packaged into an apk file, to be installed on a device and finally to be running on that device.

## 3.2 Marking and success criteria

The resulting test tool will be tested against manual testing of the application, as the tool is expected to be used on applications which cannot be tested by the other automated tools or to be tested by testers which are not developers. The research will be considered successful if significant time savings have been achieved compared to manual testing and if the testing tool can be successfully applied to real-world application of all kinds, and can be used to test all the expected use cases of the application.

For the case of the fuzz-testing tool, the research will be considered successful if the tool can successfully fuzz-test all applications which accept external input and can successfully uncover security bugs stemming for incorrect handling of specific input cases.

Regarding the part of the dissertation concerned with energy consumption measurement, the aim of the project is to create a way to estimate the power draw of a random application on a specific number of devices. The purpose of this part of the dissertation is to create a proof-of-concept measurements table and calculation utility and not a complete product covering all types of devices that exist.

# Chapter 4: Results

## 4.1 Results of attempt to automate black-box GUI testing

With the restrictions set above, a promising method for black-box GUI testing is the adb shell commands.

### 4.1.1 Shell commands that are of interest to GUI testing

Shell commands can automate GUI testing, by allowing an adb client to issue repeated GUI input commands on a target device. The *shell input* command is of particular interest to GUI testing, as it can issue most GUI commands. Some of the GUI commands are:

*adb shell input tap x1 y1*: Simulates a tap on the screen on the pixel position defined by x1,y1

*adb shell input swipe x1 y1 x2 y2 t*: Simulates a swipe from pixel position defined by x1,y1 to pixel position defined by x2,y2. For example, the swipe needed to pull down the notification bar can be simulated with the command adb shell input swipe 10 10 10 1000 (a vertical swipe starting 10 pixels from the top of the screen and ending 1000 pixels from the top of the screen (which should be enough to pull down the notification bar even on very hi-dpi screens). Optionally, a time parameter can be added, which can specify the time it takes for the swipe to be completed. This can be useful in testing the robustness of the application against fast or slow swipes.

*adb shell input text "alphanumeric_sequence"*: Simulates a virtual keyboard input of alphanumeric sequence. It is mainly intended to be used for text field input. The alphanumeric sequence can be any Unicode character and can be used to test the robustness of an application against special characters, such as non-Latin characters and whitespace control characters.

*adb shell input keyevent <event_code>*: Simulates input from a hardware button or from a hardware keyboard. Each button or keyboard key supported by Android has been assigned a special code (a special non-negative integer). A hardware button can be the power button, the volume up or down button, the back button. A full list of those keycodes can be found in Appendix B.

### 4.1.2 Automation of GUI testing using software:

All the adb commands (along with the initial initialization commands to connect the client with the server) can be put together in batch files and executed sequentially.

In order for automation to be achieved, it is necessary for a program which can create and chain together adb commands into a batch file to exist, and subsequently execute those commands. Documentation of an attempt to create such a program is described below.

### 4.1.3 The GUITestAutomation Software Package

The GUITestAutomation software package is a UI-driven software package written in the Java programming language, created for the purposes of this dissertation. It runs on a development machine (assumed to be Microsoft Windows for the purposes of this study) running an Android Emulator or with physical Android devices connected to it. The software package was created using the Netbeans IDE and depends on the Java Swing library for the creation of its own UI. The application's main purpose is to create and sequence adb input commands and send them to a target device with an actively running adb daemon.

The application offers the ability to create commands for common forms of input, such as touch tapping, touch swipe, hardware keyboard input, hardware button input and virtual keyboard text input.

In addition, the application can capture a sequence of real user input from a target device, store them in the file in the developer's computer, and play them back later. This feature is very useful for automating repeated UI testing that otherwise would have to be performed manually.

### 4.1.4 Design of the software

The software has been designed according to the Model-View-Controller (MVC) design pattern.

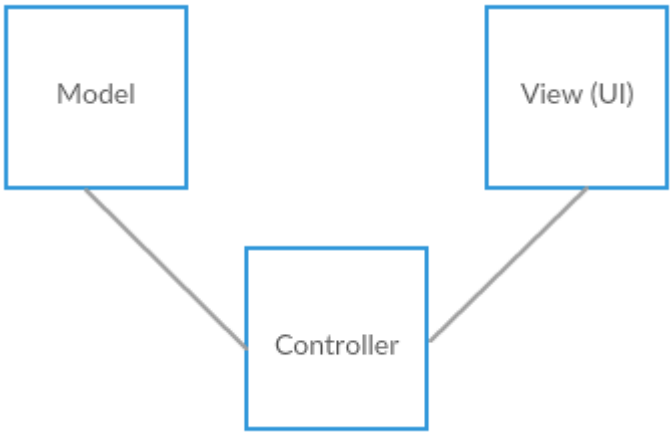



Figure 4: The MVC pattern (note that model and View don't interact directly) (own work)

#### 4.1.4.1 The Model of GUITestAutomation

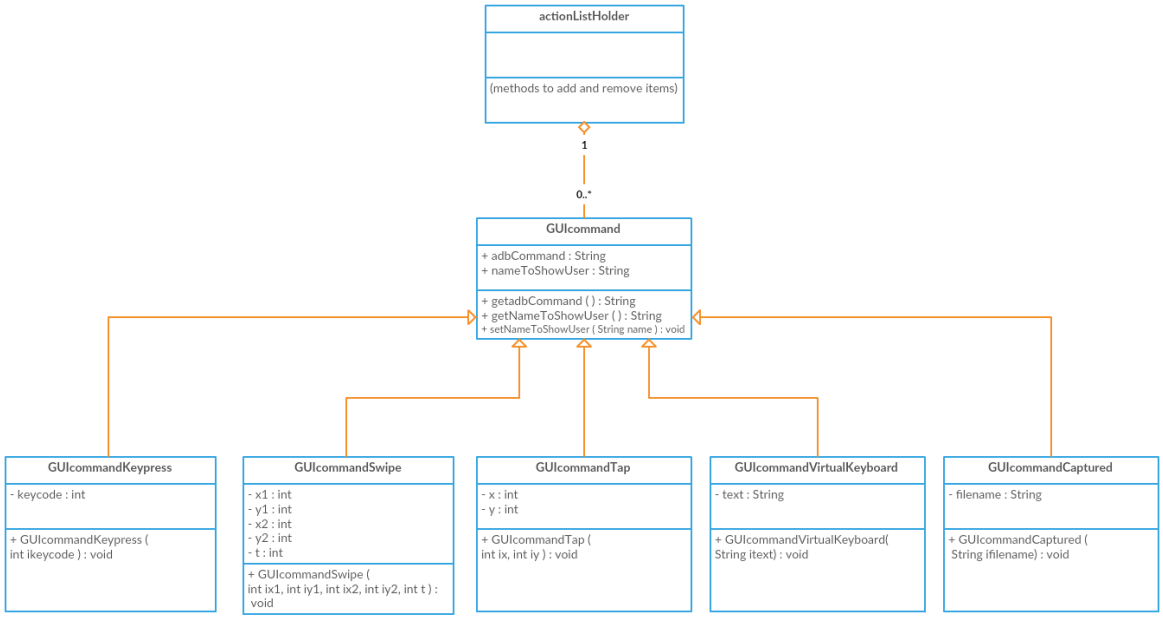



Figure 5: UML diagram of the model (own work)

The Model is implemented inside the *actionListHolder* class. This class contains an array of object whose class is *GUIcommand*. Each *GUIcommand*-class object holds a single test command to be issued to the Android device, be it an "artificial" command (an action generated by the testing software) or a captured command (captured input from real user actions). Of course, methods to add and remove *GUIcommand* objects from *actionListHolder* are provided, as well as methods to retrieve information from those objects.

A *GUIcommand* object consists mainly of the command itself (held inside a string called *adbCommand*) and a name which describes the command in a way that is easy-to-understand by the users of the program and is meant to be displayed on the GUI (and is held in a string called *nameToShowUser*). In the case of captured commands, the string does not contain any actual command, but name of the file containing the captured data.

Five child classes extend that base class, one child class for each kind of supported input commands (tap, swipe, hardware keypress and virtual keyboard input) plus one more class for captured events. The child classes have additional fields containing pieces of information unique to each kind of command. For example, the class for the "tap" input event holds the co-ordinates of the tap, while the class for virtual keyboard input holds the string to be typed.

There are two more child classes used for fuzz testing and testing of network changes actions, but those will be analyzed in subsequent parts of the dissertation.

#### 4.1.4.2 The UI of GUITestAutomation

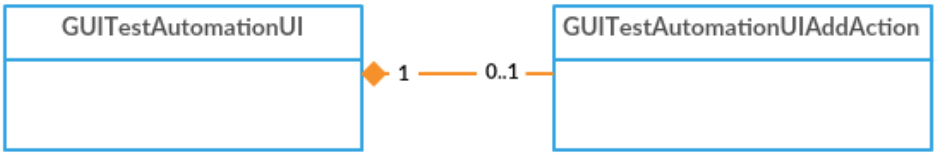



Figure 6: The UI class diagram (method names were auto-generated, see source code) (own work)

The functions of the UI are: Present to the user the sequence of commands to be executed, allow the user to create new artificial commands and delete existing ones, present buttons for capturing live commands, and then a button for executing the sequence of commands on the device. The GUI is implemented in the classes containing the substring *GUITestAutomationUI* in their names. All those classes are children of *jFrame*. Each window is being implemented in its own class. The most basic functions of the UI are:

-Add new artificial command: This window that allows the user to create a new artificial command and is implemented in the *GUITestAutomationUIAddAction* class. 8 options are available to the user, as to make special cases easier to handle. For example, when it comes to the "swipe" event, the user can choose to either specify just the swipe event (the starting pixel and the end pixel) using the swipe option and get a predetermined completion time of 1 second (1000 miliseconds) for the swipe, or choose option number 3 which has allows the user to explicitly define the completion time for the swipe, allowing the user to test for edge cases such as very fast or very slow swipes, for when robustness testing must be performed. Additionally, an option called "swipe down notification bar" is being provided. Swiping down the notification bar is an action done relatively frequently in Android operating systems. This command is essentially an ordinary vertical swipe from the top of the screen (for example pixel (10,10)) to 1000 pixels down with a duration of around 1000ms, and hence could have been done using appropriate inputs in command type 2. However, since it is such a common command, having an easy way to create it (by having a dedicated command for it) saves the tester significant time. A similar time-saving measure was chosen when it comes to the hardware keypress input event. Options #7 and #8, which simulate a press of the home and back hardware keys accordingly, are nothing more than ordinary keypresses and could have been created (by the user) using option #6 (keypress) using the appropriate keycodes. However, since those actions are very heavily used in Android applications, it is beneficial to have them as separate options. The input events corresponding to single tap and virtual keyboard input correspond to only 1 option each.

-Capture live actions: The Start/Stop capture button starts and stops capture of a live event to a file. The user can optionally provide a filename for the capture file. If a filename is not provided, the name "Default" is used.

-Run live on open adb server: This button runs the artificial and captured commands sequentially in the order they are shown in the table. It is assumed that an emulated or physical device with a running adb daemon is present. If not, the button will not do anything.

- Export inspection log: This button exists mainly for debugging purposes and saves the commands that will be executed on the adb server in a log file, so the software can be inspected even without the presence of an adb server.

There are two extra buttons related to fuzz testing and network testing, which will be explained in subsequent chapters of this dissertation.

#### 4.1.4.3 The Controlller of GUITestAutiomation (and feature implementation description)

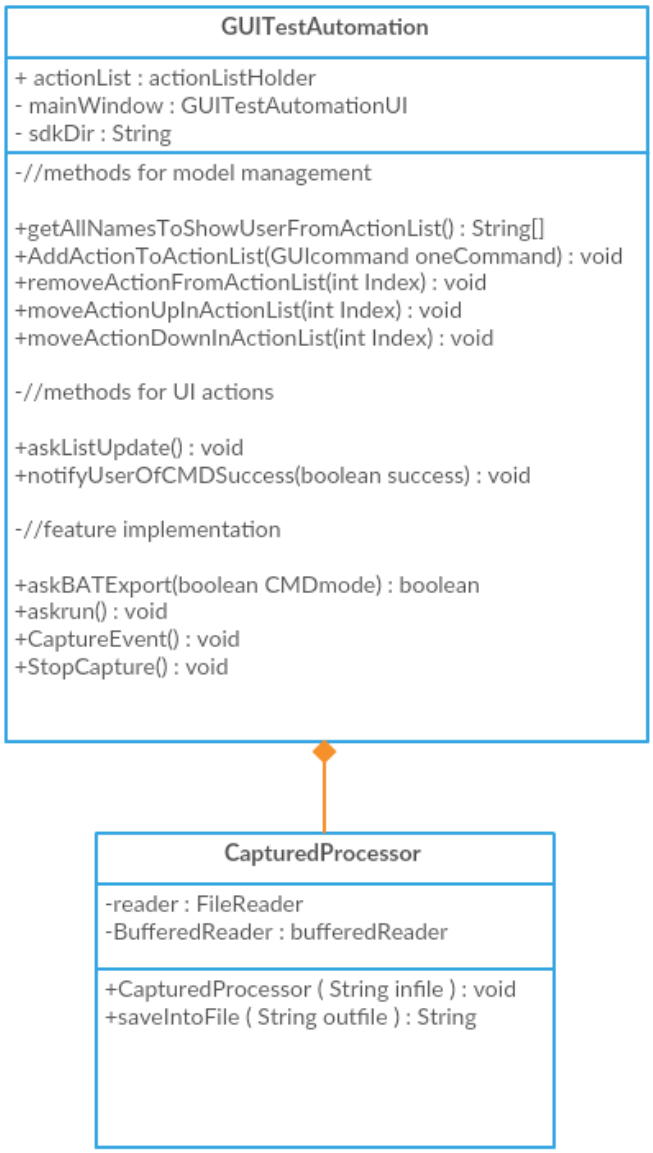



Figure 7: UML diagram of the controller (own work)

The Controller is implemented in the *GUITestAutomation* class. It implements the core features of the software and manages the model and the UI. Since this is the class containing the main method (entry point) of the software, it is the class that instantiates the model and also creates the main UI window.

In addition, the controller acts as an intermediate between the UI-related classes and the model. Every time the GUI must change something in the Model or retrieve information from the Model, a function in the Controller is called.

Generating and executing artificial commands:

The UI-related classes request from the controller to add the appropriate command to the model, by calling the *addActionToActionLis*t method. The new command is always added to the bottom of the list. The syntax of each of the commands is shown in chapter 3.1.1 above, and is basically the adb executable with the appropriate parameters.

In order for the adb commands to be executed, a OS shell with them must be launched. This functionality is being implemented in the *askrun* method. Since the software is intended to run in Microsoft Windows operating systems, creating a BAT file and launching it (from inside Java) was chosen as a simple solution. Hence, the *askrun* method retrieves each command one-by-one from the Model and creates a BAT file with it, with is then executed using the *Runtime.getRuntime().exec* method Java provides. The only issue with this approach is that Java normally doesn't wait for the execution of the BAT file to finish and instead continues with the execution of the next command (the method Java provides to wait for execution to finish didn't work). A way to avoid the issue is to have Java create a file called *runningbat* which is meant to be deleted by the BAT file when it finishes, and have Java wait until the file no longer exists.

Although this is not a perfect solution, it has the advantage that is does not rely on any idiosyncrasies of either Java or the Windows command-line processor. In addition, the generated BAT files do not rely on any features of the Windows command-line processor. The obvious advantage of this is that the software can be ported to any other OS and to any other language with minimal effort, as long as the ability to create a file and execute it are provided by the programming language.

Capture and playback of commands:

The feature is implemented in the *CaptureEvent* method. In order to capture events from the Android device, the *getevent* adb command is used, which provides a list of all the input events happening on the device. This list is then redirected to a temporary file (called "event.log"). The complete command which stats the logging is as follows:

*adb shell getevent>event.log*

The capture is stopped by the *StopCapture* method. In order to stop the capture, the method must kill the adb processes that had been previously launched.

After that happens, the temporary file must be converted into a file containing *sendevent* commands that can be executed. An interesting detail is that the format *getevent* uses to capture the events is different from the format *sendevent* expects.

The captured event in event.log is in the following format:

*/dev/input/event1: 0003 0039 00000000*

While the command that must be executed should be in the format:

*adb shell sendevent /dev/input/event1  3 57 0*

The numbers in event.log are in the hexadecimal format (with option for decimal provided by *getevent*) while *sendevent* requires decimal. For that reason, a class called *CapturedProcessor* (processor of captured events) was created which performs the necessary conversions, in the *saveIntoFile* method. The class skips events unrelated to user input and creates an executable batch file (which is also the file storing the captured events permanently). The file is saved in the directory of the SDK (sdkDir in code) The sole parameter of saveIntoFile is the desired filename, which is defined by the user. If no filename is provided, the name "Default" is used. If a file of that name already exists, then the method adds a "_new" suffix to the filename and retries. The *saveIntoFile* method returns the filename of the file created. A further peculiarity of *getevent* worthy of note is that the numerical values do not correspond to pixel values and there is no easily-obtainable documentation explaining what exactly they represent and whether it differs from device to device, hence "decoding" the commands proved to be rather difficult, although further research could be made into the subject.

After that, all that is left for the *StopCapture* method to do is add an entry into *actionList*, which of course is of class *GUIcommandCaptured*.

The created file will be executed when the *askrun* method comes across the entry while executing the sequence in the list. The way the BAT is executed is the same as the way *askrun* uses to run all the other BAT files.

#### 4.1.4.4 Final comment on the GUITestAutomation software

All in all, a complete black-box testing tool was developed, capable of sequencing artificially-created commands and captured touch events. An advantage of the software is that it can be used by users with minimal computer skills and does not require any kind of programming knowledge to use, unlike the UI automator tool provided by Google. This would allow the black-box testing of the application to be outsourced to a much wider number of people, or even to some of the intended users of the application.

In the following section, a comparison of the time-saving achieved using this tool compared to manual testing for different kinds of applications will be analyzed.

### 4.1.5 Identity of experiment

The experiment involves testing a set of use cases (with each use case being a sequence of GUI actions) multiple times each for a single application. This experiment will be repeated for three applications, believed to be representative of the most commonly used Android applications. Since the black-box testing tool developed for the purposes of this dissertation (GUITestAutomation) is meant to be used where no Google-provided testing tool can be used, its efficiency will be measured against manual testing.

How the experiment will be conducted:

For each application-to-be-tested, a number of use cases will be chosen. First, the use case is tested only once, manually and with the use of the tool (first repetition of test). Then, the use case will be tested one more time, for a total Number of Repetitions (NoR) of 2. Then, the use case will be tested a third time for a NoR of 3 etc. Each of the use cases of the application will be tested with the same number of NoRs, that is six, for the sake of presentation simplicity. The total time taken to test a use case manually and then the time taken to test with the help of the automated tool will be measured for each NoR value. For example, for a NoR of 2, the total time taken will be the time spent to do repetition one and then repetition two. When measuring the time it takes to test with the automated tool, the setup time and the execution time will be taken into account. The setup time applies only to the first repetition.

As an additional note, the NoR value for which testing in an automated manner takes an equal amount of time as manual testing for all a particular use case, is called the “break even” point. The experiment will be considered a success if the “break even” points for most use cases represent a reasonable number of repetitions in the single digits, as testers are not expected to test their use cases for a higher number of repetitions.

The experiment was performed by running the Android emulator, with an image of Android 6 “Marshmallow” (latest stable version as of writing) running and having the applications to be tested already installed.

Expectations of efficiency in general:

When each use case needs to be tested only for a very small number of repetitions, the efficiency of manual testing is expected to be better than the efficiency of the GUITestAutomation tool, due to the one-time overhead involved in setting up the automated tool (for example, one kind of overhead is that the user cases to be tested must be converted into a sequence commands for the GUITestAutomation tool, using artificial commands or the capture functionality). However, as the number repetitions increases, the efficiency of GUITestAutomation is expected to improve and ultimately surpass the efficiency of manual testing significantly, because the automated tool can submit input events and commands much faster than a human tester could.

Threats to validity:

The only threats to validity for this experiment is the variance in applications, and the fact some testers are faster in manual testing than others (and maybe slower in automated testing than others). In order to counteract those threats, a variance in applications will be chosen, and in

addition, the best case scenario will be used for manual testing (which means the fastest time will be chosen as the value for all repetitions) while the average case scenario for automated testing will be used (where the median value from the times needed for the setup of the automated tool will be chosen). The time it takes to execute the script is fixed (at least to the degree it can be measured by a stopwatch) and will be treated as such. Also, the device chosen will be the Android emulator, a slow device which adds to the total execution time, which benefits manual testing, adhering to the "best case scenario for manual testing" principle.

Applications chosen:

The applications chosen is one C++ and OpenGL application with a rich and complex UI (Kodi), and one application partially consisted of C++ and OpenGL code with a moderately-complex UI (Google Maps), and another application with a moderately-complex UI (AOSP Browser).

All numbers are seconds and refer to time taken to do the task.

### 4.1.6 Results of experiment

Some extra notes about the experiment: We assume all use case tests start from the start screen of the application and return to the start screen after they have finished. This is necessary because in real-world app testing, the different use-case tests will be performed one after the other, and returning to the start screen is a precondition for this.

Tool setup time happens only once (first repetition).

**Kodi application, Use case 1: Opening a local video file**

| **Task:** | **NoR=1** | **NoR=2** | **NoR=3** | **NoR=4** | **NoR=5** | **NoR=6** |
|---|---|---|---|---|---|---|
| **Manual testing** | | | | | | |
| Time spent for this repetition (use fastest manual time for all reps): | 12.70 | 12.70 | 12.70 | 12.70 | 12.70 | 12.70 |
| Total time spent for manual testing of this use case: | 12.70 | 25.40 | 38.10 | 50.80 | 63.50 | 76.20 |
| **Automated black-box testing** | | | | | | |
| Tool setup time (capturing input): | 14.12 | --- | --- | --- | --- | --- |
| Script execution time: | 6.59 | 6.59 | 6.59 | 6.59 | 6.59 | 6.59 |
| Time spent for this repetition (setup plus script execution time): | 20.71 | 6.59 | 6.59 | 6.59 | 6.59 | 6.59 |
| Total time spent for automated testing of this use case: | 20.71 | 27.30 | 33.89 | 40.48 | 47.07 | 53.66 |

Table 1: Kodi application GUI testing, Use case 1, Opening a local video fie

Note: this use case involves reaching the file and returning to the application's start screen.

**Kodi application, Use case 2: Reaching the settings screen**

| **Task:** | **NoR=1** | **NoR=2** | **NoR=3** | **NoR=4** | **NoR=5** | **NoR=6** |
|---|---|---|---|---|---|---|
| **Manual testing** | | | | | | |
| Time spent for this repetition (use fastest manual time for all reps): | 9.10 | 9.10 | 9.10 | 9.10 | 9.10 | 9.10 |
| Total time spent for manual testing of this use case: | 9.10 | 18.20 | 27.30 | 36.40 | 45.50 | 54.60 |
| **Automated black-box testing** | | | | | | |
| Tool setup time (capturing input): | 11.81 | --- | --- | --- | --- | --- |
| Script execution time: | 4.70 | 4.70 | 4.70 | 4.70 | 4.70 | 4.70 |
| Time spent for this repetition (setup plus script execution time): | 16.51 | 4.70 | 4.70 | 4.70 | 4.70 | 4.70 |
| Total time spent for automated testing of this use case: | 16.51 | 21.21 | 25.91 | 30.61 | 35.31 | 40.1 |

Table 2: Kodi application GUI testing, Use case 2, Reaching the settings screen

**Google Maps, Use Case 1: Finding a place on the map**

| **Task:** | **NoR=1** | **NoR=2** | **NoR=3** | **NoR=4** | **NoR=5** | **NoR=6** |
|---|---|---|---|---|---|---|
| **Manual testing** | | | | | | |
| Time spent for this repetition (use fastest manual time for all reps): | 12.15 | 12.15 | 12.15 | 12.15 | 12.15 | 12.15 |
| Total time spent for manual testing of this use case: | 12.15 | 24.30 | 36.45 | 48.60 | 60.75 | 72.90 |
| **Automated black-box testing** | | | | | | |
| Tool setup time (capturing input for selecting text field): | 7.82 | --- | --- | --- | --- | --- |
| Tool setup time (generate command for virtual keyboard input): | 9.68 | --- | --- | ---- | --- | --- |
| Tool setup time (capturing input for searching for the place and going back to main screen): | 8.87 | --- | --- | --- | --- | --- |

| | | | | | | |
|---|---|---|---|---|---|---|
| Script execution time: | 5.89 | 5.89 | 5.89 | 5.89 | 5.89 | 5.89 |
| Time spent for this repetition (setup plus script execution time): | 32.26 | 5.89 | 5.89 | 5.89 | 5.89 | 5.89 |
| Total time spent for automated testing of this use case: | 32.26 | 38.15 | 44.04 | 49.93 | 55.82 | 61.71 |

Table 3: Google Maps GUI testing, Use Case 1, Finding a place on the map

Note: The command for virtual keyboard input is a generated (an artificial) command instead of been captured so the text can be changed easily

**Google Maps, Use Case 2: Opening settings**

| **Task:** | **NoR=1** | **NoR=2** | **NoR=3** | **NoR=4** | **NoR=5** | **NoR=6** |
|---|---|---|---|---|---|---|
| **Manual testing** | | | | | | |
| Time spent for this repetition (use fastest manual time for all reps): | 4.08 | 4.08 | 4.08 | 4.08 | 4.08 | 4.08 |
| Total time spent for manual testing of this use case: | 4.08 | 8.16 | 12.24 | 16.32 | 20.40 | 24.48 |
| **Automated black-box testing** | | | | | | |
| Tool setup time (capturing input): | 7.76 | --- | --- | --- | --- | --- |
| Script execution time: | 3.98 | 3.98 | 3.98 | 3.98 | 3.98 | 3.98 |
| Time spent for this repetition (setup plus script execution time): | 11.74 | 3.98 | 3.98 | 3.98 | 3.98 | 3.98 |
| Total time spent for automated testing of this use case: | 11.74 | 15.72 | 19.7 | 23.68 | 27.57 | 31.55 |

Table 4: Google Maps GUI testing, Use Case 1, Finding a place on the map

**Google Chrome, Use Case 1: Visiting a bookmarked webpage**

| **Task:** | **NoR=1** | **NoR=2** | **NoR=3** | **NoR=4** | **NoR=5** | **NoR=6** |
|---|---|---|---|---|---|---|
| **Manual testing** | | | | | | |
| Time spent for this repetition (use fastest manual time for all reps): | 6.60 | 6.60 | 6.60 | 6.60 | 6.60 | 6.60 |
| Total time spent for manual testing of this use case: | 6.60 | 13.20 | 19.80 | 26.40 | 33.00 | 39.60 |
| **Automated black-box testing** | | | | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| Tool setup time (capturing input): | 7.50 | --- | --- | --- | --- | --- |
| Script execution time: | 3.70 | 3.70 | 3.70 | 3.70 | 3.70 | 3.70 |
| Time spent for this repetition (setup plus script execution time): | 11.20 | 3.70 | 3.70 | 3.70 | 3.70 | 3.70 |
| Total time spent for automated testing of this use case: | 11.20 | 14.9 | 18.6 | 22.3 | 26 | 29.70 |

Table 5: Google Chrome GUI testing, Use Case 1, Visiting a bookmarked webpage

**Google Chrome, Use Case 2: Accessing browser history**

| **Task:** | **NoR=1** | **NoR=2** | **NoR=3** | **NoR=4** | **NoR=5** | **NoR=6** |
|---|---|---|---|---|---|---|
| **Manual testing** | | | | | | |
| Time spent for this repetition (use fastest manual time for all reps): | 8.20 | 8.20 | 8.20 | 8.20 | 8.20 | 8.20 |
| Total time spent for manual testing of this use case: | 8.20 | 16.40 | 24.60 | 32.80 | 41.00 | 49.20 |
| **Automated black-box testing** | | | | | | |
| Tool setup time (capturing input): | 10.00 | --- | --- | --- | --- | --- |
| Script execution time: | 4.70 | 4.70 | 4.70 | 4.70 | 4.70 | 4.70 |
| Time spent for this repetition (setup plus script execution time): | 14.70 | 4.70 | 4.70 | 4.70 | 4.70 | 4.70 |
| Total time spent for automated testing of this use case: | 14.70 | 19.40 | 24.10 | 28.80 | 33.50 | 38.20 |

Table 6: Google Chrome GUI Testing, Use Case 2: Accessing browser history

### 4.1.7 Comment on results

It is obvious from the results that using the automated testing tool instead of the manual method results in an overhead of about 2 extra seconds compared to manual testing, plus an overhead of 4-6 for script execution on the first repetition. However, for subsequent repetitions, the overhead is not only diminished, but significant savings in time are achieved even for low NoR figures.

Hence, the experiments are considered a success.

## 4.2 Results of attempt to automate location-services testing

### 4.2.1 Types of location services that need to be tested

Location-based applications comprise a significant portion of any mobile application "ecosystem", in contrast to applications created for desktop and laptop applications, where a comparatively small portion of them utilizes location data.

As with all mobile operating systems, Android provides a default API for interacting with location-services. In fact, Android offers three distinct types of location services. The reason for the existence of three types of location services is because they offer different levels of accuracy and vary in terms of power-consumption and availability. Accuracy, in the context of location services, is defined as the "margin of error" from the reported location, and is usually defined in meters. For example, an accuracy of 10 meters means that the actual location of the mobile device can be anywhere inside a circle with the reported location as it's center and a diameter of 10 meters. Power-consumption is self-explaining, and is of course the increase in power-consumption from using that particular location service. Availability is the likelihood the particular location service is available at a particular moment, depending on several conditions.

More specifically, the three types of location services Android offers are:

**Cellular network:** In all mobile devices which have a GSM or CDMA module, the operating system has access to the ID of the cellular tower the mobile device is connected to. The precise location of the cellular tower corresponding to that ID is provided by the network provider. In addition, there is an area (on the map) which is covered by this tower and is defined by a radius around the location of the tower. Since the user is serviced by this tower, the user is expected to be inside this area, and hence, an approximation of the user's location can be derived from this information alone. The accuracy of this location method varies from several hundred meters in dense urban areas to several kilometers in sparsely-populated areas. As a result, this location method provides a low level of accuracy, and is referred to as "coarse location" or "network provider location". This information is available to the system even if transmission of data using the mobile network ("mobile data") has been turned off by user and is unavailable only when the mobile device has no signal (for example when there is no service or when the mobile device has been set to "flight mode"), and is considered to have high availability. This location method is generally used to get an initial, non-accurate location fix before more accurate location services (the ones described below) become available. Since the "coarse" location information is always available to the device, power consumption is near-zero.

**WiFi:** When the user of the mobile device has the WiFi adapter enabled, a list of all the WiFi hotspots found in close proximity is available to the operating system (along with their MAC addresses). From those WiFi hotspots, a number of them are fixed in terms of location. In addition, databases containing the precise location of WiFi hotspots exist online. Indeed, Google Inc -the vendor of the Android OS- maintains such a database. Since WiFi has a real-world range of around 50 meters, a closer approximation of the user's location (compared to network provider location) can be made. The user does not need to be connected to one of the WiFi hotspots for

this to happen. This location service (often called “WiFi location”) is available only when the WiFI adapter is enabled, and only when internet access is available (not necessarily via WiFi), so the internet database can be accessed, and as a result, it’s not always available. Power consumption is considered to be near-zero if the WiFi adapter is already enabled.

**GPS:** GPS (and other similar standards like GLONASS and Gallileo) location service is the only type of location service that can provide a truly accurate location fix, with an accuracy as high as 1 meter, making services like turn-by-turn navigation (tracking user movement) possible. However, requesting for a GPS fix consumes a considerable amount of power, and the time required to obtain the fix can be several seconds (for devices with GPS and GLONASS receivers) or up to half a minute (for GPS-only devices). This location service is only available when the user has the GPS adapter enabled and when there is line-of-sight contact with the GPS/GLONASS satellites. For that reason, this location service is never used exclusively, but as a way to obtain a more accurate fix after a “coarse” location fix has been achieved (by using one of the other services described above). In applications where the user not expected to move, the GPS service must be polled until an accurate location fix has been obtained and then not be polled again or polled sparingly. In applications where user movement is expected (such as navigation applications), an accurate location fix should be obtained (like in the previous case) and the GPS location service should be polled periodically, in order to track user movement. In addition, the GPS location service tends to have high “jitter” in some devices, meaning that the location fix can be somewhat inaccurate and report user movement when there is none, since not all mobile devices have GPS receivers sensitive enough to guarantee a jitter-free fix. For that reason, applications should be able to cope with this kind of jitter, and a way to test for this kind of scenario would be an advantage. One way of copping would be to take the average of several fixes, and when determining the location of the user, give preference to locations the user is most likely to be (for example, preference for locations corresponding to roads and sidewalks in navigation applications).

### 4.2.2 Obtaining a location fix

The code snippet below details the general way of obtaining a location fix on a device running the Android OS:

```
// Acquire a reference to the system Location Manager
LocationManager locationManager = (LocationManager)
this.getSystemService(Context.LOCATION_SERVICE);

// Define a listener that responds to location updates
LocationListener locationListener = new LocationListener() {
    public void onLocationChanged(Location location) {
      // Called when a new location is found by the network location provider.
      makeUseOfNewLocation(location);
    }

    public void onStatusChanged(String provider, int status, Bundle extras) {}
```

```
    public void onProviderEnabled(String provider) {}

    public void onProviderDisabled(String provider) {}
  };

// Register the listener with the Location Manager to receive location updates
// Get location by Network/cellular tower ID (coarse)
locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 0, 0,
locationListener);

// Get location by GPS
/* locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0,
locationListener); */
```

Copyright of source code fragment above: developer.android.com

The *LocationManager* class is provided by the Android Framework, and provides methods which poll the locations services (using *requestLocationUpdates* function). Every time a location service is polled, the *locationManager* object sends a notification which contains the location fix returned by the service, which is "listened" by the *locationListener* object (which belongs to the class of *LocationListener*). In our example, the co-ordinates of the location fix are encapsulated inside the "location" object (of type Location). This object is automatically passed as an argument to the *onLocationChanged* method. In essence, the *onLocationChanged* method is the method which is called when a new fix is obtained by *locationManager*. This means that if the application must perform an action when a location fix is obtained (for example show the co-ordinates to the user), this action must happen inside *onLocationChanged*.

The *requestLocationUpdates* method can either poll the "network" location service (for a coarse location based on cell tower ID) or the GPS service. Which of the two services will be polled is defined by the first argument of the method. The second argument is the minimum time interval between the location fix notifications. In real-world code, this value would be higher than 0, to prevent high power consumption (because 0 would tell the location service to return location fixes as fast as it can, resulting in high battery drain). The third argument is the minimum change in distance between location fixes. In other words, the service will send a notification when the device moves by a distance higher than the minimum specified in the third argument. Again, this would be set to a value higher than one to prevent excessive battery drain.

It's worth noting that two location providers (for example NETWORK_PROVIDER and GPS_PROVIDER) can be used at the same time, which would require two Location Manager objects and two listeners. The *makeUseOfNewLocation* function (the function which actually uses the location data), would have to use the location.getAccuracy method to find out which services offers the highest accuracy (according to an estimate provided by the Android Framework). The network provider location is expected to be more accurate at first, until GPS manages to obtain a fix.

### 4.2.3 Shell commands that are of interest to location services testing (injecting fix co-ordinates)

Since the application is expected to be tested on a Desktop or Laptop system, a way to "inject" artificial locations is provided by the Android Emulator, so the tester doesn't have to move around in order to test the application.

Artificial locations can be injected to the device using the Android Debug Bridge (adb).

```
adb emu "geo fix $longitude $latitude"
```

For example:

```
adb emu "geo fix 23.419002314038046 67.21900231403805"
```

With 23.419002314038046 and 67.21900231403805 being the latitude and the longitude the location service will fix to accordingly. In addition, the quotes are part of the command that must be executed.

### 4.2.4 Automation of location services testing using software

The method described above is not automated. However, as with the GUI testing application, it can be automated with custom software written for that purpose, which will take a list of location co-ordinates from the tester, and even a possible "jitter" that will be applied to those co-ordinates, and then send an appropriate series of commands to the android debug bridge.

### 4.2.5 The GPSTestAutomation software package

The GPS Test Automation software package is a UI-driven software package whose main purpose is to test location-service applications by simulating location service input. Like GUITestAutomation, it was developed in Java, uses Swing for its GUI and was developed in Netbeans.

The features provided by the software are

- Ability to simulate a precise fix, with user-specified latitude and longitude
- Ability to simulate a non-precise ("jittery") fix, with user-specified latitude and longitude and user-specified jitter.
- Ability to simulate a sequence of precise fixes, all of them with user-specified latitudes and longitudes and with a user-specified duration between the fixes. This feature is intended primarily for testing navigation-based applications

Much like GUITestAutomation, the technique of creating a batch file and executing it in a shell is used in this software too.

### 4.2.6 Design of the software

The software has been designed according to the Model-View-Controller (MVC) design pattern.

#### 4.2.6.1 The Model of GPSTestAutomation

The Model of the software is relatively simple, and is meant to hold the sequence of location fixes requires by the third feature. It consists of a class called *sequenceHolder*, which is a list holding items of class *sequenceElement*. Each object of *sequenceElement* class holds the longitude and latitude of one location fix, plus the duration of this fix (the duration of the fix is meant to represent how long the user stays in that location). A string called nameToShow is meant to hold a description of that element.

#### 4.2.6.2 The UI of GPSTestAutomation

The UI is implemented in the GPSTestAutomationUI class, and it is consisted of a single window (and hence, a single jFrame). For the first two features, the tester just inserts the necessary information and executes the command, while for the third, the tester prepares a list of location fixes and can then press execute to sequence them.

#### 4.2.6.3 The Controller of GPSTestAutomation

The Contoller is implemented in the GPSTestAutomation class. The first feature (simulate precise fix) is implemented in the *simulateFix* method, and it simply creates the batch file (called *GPSTEST.BAT*) and executes it. It is worth mentioning that there is no need for Java to wait for the batch file to finish (as there was in GUITestAutomation) as there is no risk of a second batch file being launched by the application while the first is running, as there is no for loop.

The second feature (simulate jittery fix) is implemented in the *simulateJitteryFix* controller method. In this method a series of commands which contain fix locations slightly different among them is generated, as to simulate the fix of a “jittery” GPS receiver which provides a series of slightly inaccurate readings. This phenomenon is simulated by adding an error to the longitude and latitude, with the error being a random number between zero and the “maximum jitter” value as defined by the user. 100 such commands are produced.

For the third feature, the user populates a sequence (stored in the model) with fix co-ordinates, and the duration (in miliseconds) between those fix co-ordinates. That sequence is stored in the model by the *addSequenceElement* controller method, everytime the tester clicks “add” in the UI. Finally, the *executeSequenceOfFixes* controller function executes this sequence. Since batch files do not support pausing with millisecond accuracy (only with second accuracy), the approach that was taken was to generate a batch file for a single element of the sequence, then pausing the thread for the appropriate amount of miliseconds, and then generating a new batch file for the next element.

#### 4.2.6.4 Final comment on the GPSTestAutomation software

This small piece of software plugs an important gap in the functionality of the Android SDK, as no way to automate testing of location-based apps has been provided. Also, with the software requiring no programming knowledge, it allows non-programmers users to act as testers and alleviate much of the testing burden from programmers and testers with programming skills when it comes to location-services based applications

### 4.2.7 Identity of experiment

The experiment involves repetitions of a test which involves injecting a list of 10 locations to the device, while a mapping application is running. In the real-world, injecting a list of locations is very usefull when testing navigation applications or any other type of application that needs to poll the location services more than once. For the perponses of this experiment, the co-ordinates of the locations have been chosen randomly.

How the experiment will be conducted:

The co-ordinates will be injected both manually and using the GPSTestAutomation tool. In manual testing, the built-in tools of the Android emulator for injecting GPS co-ordinates will be used, and the co-ordinates will be copied and pasted manually. In automated testing, the feature which allows for repeated injection of a list of co-ordinates will be used, after the list of co-ordinates has been prepared using the “Add” feature of GPSTestAutomation. The time needed for each of the two methods for various RoP numbers will be recorded.

This experiment was performed on the Android emulator, with an image of Android 6 “Marshmallow” (latest stable version as of writing) having Google Maps already installed.

Measuring efficiency.

As with the automation of GUI testing above, manual testing is expected to be faster for a small number of repetitions, but automated testing will be faster as RoP increases. The “break even” point is the RoP value for which automated testing surpasses manual testing.

It is worth noting that injecting “jittery” location fixes is impossible with manual testing, as the official tools don’t provide the feature and there is no way a human tester could simulate this by hand, so the automated tool developed for the purposes of this dissertation is considered to be the only way to do this and no comparison with manual testing can exist.

### 4.2.8 Results of experiment

Note: All times are in seconds

**Injecting 10 random locations**

| Task: | NoR=1 | NoR=2 | NoR=3 | NoR=4 | NoR=5 | NoR=6 |
|---|---|---|---|---|---|---|
| **Manual testing** | | | | | | |
| Time spent for this repetition (use fastest manual time for all reps): | 177.98 | 177.98 | 177.98 | 177.98 | 177.98 | 177.98 |
| Total time spent for manual testing of this use case: | 177.98 | 355.96 | 533.94 | 711.92 | 889.90 | 1067.88 |
| **Automated black-box testing** | | | | | | |
| Tool setup time (preparing list): | 110.96 | --- | --- | --- | --- | --- |
| Script execution time: | 10.69 | 10.69 | 10.69 | 10.69 | 10.69 | 10.69 |
| Time spent for this repetition (setup plus script execution time): | 121.65 | 10.69 | 10.69 | 10.69 | 10.69 | 10.69 |
| Total time spent for automated testing of this use case: | 121.65 | 132.34 | 143.03 | 153.72 | 164.41 | 175.51 |

Table 7: Location services testing, injecting 10 random locations

### 4.2.9 Comment on results

The results of the experiment exceed expectations, as the automated method “broke even” with NoR=1. This was caused by the fact that, when co-ordinates are inserted manually, the tester has to look at the map and confirm the fix every time he inserts a co-ordinate, something which doesn’t have to be done when preparing the list. In overall, the tool is expected to result in immense time savings when multiple location co-ordinates must be inserted into an application, and hence the experiment is considered a success.

## 4.3 Results of attempt to automate security testing

### 4.3.1 Introduction

As more and more applications handle sensitive user data (for example banking and online payment applications) and as the Android operating system itself stores sensitive user data and a multitude of user account details, the topic of security becomes more and more important.

There are two main kinds of security vulnerabilities that may affect an Android application:

1) Vulnerabilities in the application itself: These include common application vulnerabilities, where bugs in the code can result in exploitable vulnerabilities when a “crafted” input consisted of an unexpected sequence of bytes or a single unexpected byte (character) is given to the application. Testing for such vulnerabilities is challenging and is very dependent on the particular application being examined. However, it is possible to automate part of this procedure with a fuzz testing tool, which tests the different inputs of an application by giving them as input random bytes of data of random length. This will uncover any bugs which are related to improper handling of specific input characters or related to improper handling of strings that exceed a certain length.

2) Vulnerabilities of the operating system: Older versions of the Android operating system, as well as components of such versions like WebView (a component for rendering webpages inside an application), have documented vulnerabilities. Since an application is expected to run on older versions of the OS, and since It is impossible for the application to patch the operating system, it is important to have an automated method which will allow the application to “discover” any vulnerabilities that may be present in the environment it is running on. This would possibly allow the application to employ mitigation measures, such as avoiding using some components of the operating system and instead use alternatives, for example using another webpage renderer instead of WebView or using another multimedia framework instead of Stagefright, which also has critical vulnerabilities in some versions of the OS. Another countermeasure would be to disable certain sensitive features entirely.

### 4.3.2 Testing against vulnerabilities in the application (fuzz testing)

As mentioned above, testing for all kinds of vulnerabilities an app may have is impossible. This holds true because vulnerabilities in applications stem from bugs in the application logic and from special input cases being improperly handled, and it is not possible to create a program that will identify all of the application’s logic bugs. However, it is possible to create a program that exposes some of the (security) bugs of another program via the method of fuzz testing. Fuzz testing involves testing various types of input on an application of variable length and of a variety of characters. In general, it is wise to test input which is of significant length, as some applications tend to expose bugs like buffer overflows on long inputs, which in turn can expose significant security vulnerabilities. It is also wise to test input which contains special characters, be it special Unicode characters, non-printable characters, or non-printable control characters.

However, testing for those kinds of input manually can be a tedious process. Indeed, manual testing using long strings is a mundane affair, which is the reason most testing done manually by human testers is generally done with short-length inputs. In addition, typing special Unicode characters or non-printable (control) characters is difficult, and when it comes to mobile applications it is almost impossible, as the default emulator doesn't offer a documented way of inserting special Unicode characters and non-printable (control) characters.

### 4.3.3 Shell commands that are of interest to fuzz testing

An interesting feature of the command used to input virtual keyboard input, the *adb shell input text 'text_to_be_typed'* command, is that it cannot simulate the input of all special characters, even if those characters are properly escaped, because the default Android virtual keyboard doesn't allow it. Also, the command that helps simulate keypresses also can't simulate the input of all special characters, because it doesn't provide keypress codes for all special characters/

In order to sidestep this problem, a third-party keyboard (called ADBkeyboard) designed specifically in order to accept input of special characters from an adb server has to be installed on the device which the application will be fuzz-tested on. After this third-party keyboard has been installed, it must be set as the device's default keyboard from System Settings.

The adb command to send a string with special characters to that third-party keyboard is:

```
adb shell am broadcast -a ADB_INPUT_TEXT --es msg 'text_to_be_typed'
```

Where *am* refers to Activity Manager and *broadcast -a* means that an Intent message of type *ADB_INPUT_TEX*T will be sent to the device. The activity which has being registered to listen to such kinds of messages (the default keyboard) will be started. The rest of the command is the payload of the message (essentially the text to be typed).

### 4.3.4 Automation of fuzz testing

A tool which automates the process of fuzz testing for mobile application would be of great help to testers. A fuzz testing tool should include the following features:

- Quick fuzz testing of an input field, by inserting a random string of random length. It is crucial that the random string contains all possible Unicode characters allowed by the device.
- Quick fuzz testing of an input field, by inserting a random string of specific length. This feature will allow the tester to test random strings which are arbitrarily large or small, to test the application's robustness against inputs of various lengths.
- Fuzz testing for a specific character or characters. This feature will allow the tester to test against a specific character. In order to allow for the insertion of any possible character, the Unicode codepoint of the character can be inserted by the tester, and the fuzz testing tool will produce the appropriate Unicode character

### 4.3.5 Design of the fuzz testing tool

In order to implement the above features, the GUITestAutomation tool (documented above) was modified to include Fuzz Testing functionality. With this approach, fuzz testing can be smoothly integrated into the GUI testing routine of the tester. In order to accomplish this, one more child class of the GUI command class was added, which contains the fuzz text to be typed into the input field. The *adbcommand* field contains the command to be sent to the adb.

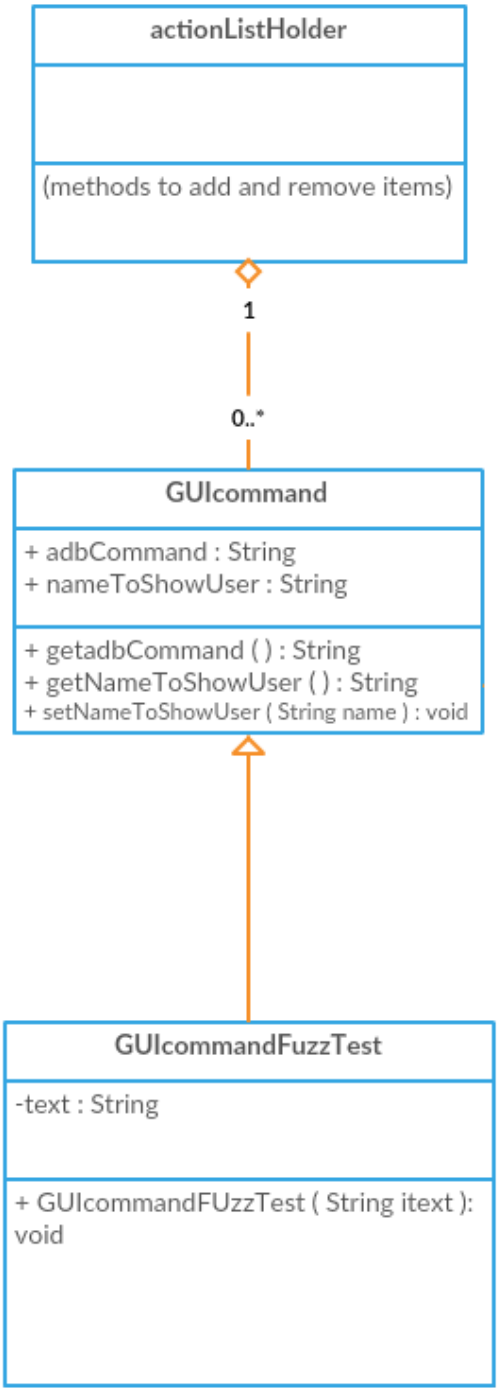



Figure 8: The parts of the GUITestApplication model that are relevant to fuzz testing (own work)

In addition, a small modification had to be made to the controller of GUITestAutomation tool, so the resulting batch file, which will execute the fuzz command, has its "codepage" set to Unicode. Without that modification, the command processor will attempt to read the 16-bit Unicode characters as 8-bit text (in the windows-1252 or windows-1253 encoding or something similar. The command that sets the "codepage" to Unicode is *chcp 65001*.

### 4.3.6 Results of testing the insertion of fuzz strings using the fuzz testing tool

A wide variety of tests were done with the fuzz testing tool, to validate its ability to insert every character in the Unicode spectrum into a text field of an application running on Android OS, via the adb. The tool can handle all the Unicode spectrum except the single quote (apostrophe) character. Due to the fact the string has to go through 2 different OSes (Windows and Android) which use two different kinds of quotes for escaping (Windows uses double and Android uses single), it was not possible to properly escape the single quote, despite the fact a number of escape characters and multiple quotes were attempted. However, the inability to insert a single quote is not expected to be a major problem for the tester, as he can test the application's robustness against single quotes by sending a virtual keypress with the code 75. The fuzz testing tool filters out the single quote, in order to not break any random strings that might contain it. A further development of the fuzz testing tool (should it enter production) would be to create the appropriate keypress itself.

So, while not an absolute success, the fuzz testing tool proves itself useful for testing applications for potential security vulnerabilities.

### 4.3.7 Testing against vulnerabilities in the operating system and its components

It is well known fact that older versions of all operating system have documented vulnerabilities, which have been documented on "CVE vulnerability repositories" on the internet. In the case of Android OS, older devices do not receive security patches, and hence an application developer should not expect every device to run the latest (patched) version of the operating system. The official SDK of the Android OS offers no way to generate code that will check for vulnerabilities in the environment it is running on, and each application developer has to develop such a capability as a custom-made solution.

### 4.3.8 Automating vulnerability detection in the operating system and its components

A library which checks the version of the OS and OS components (such as stagefright and WebView) can be developed and released in source-code form, which can be (re)used by many developers. This library can contain a database of all known vulnerabilities so far and with their severity (what aspects of security they affect and how) and this database can be built upon by other developers.

An Android application called VersionReport was developed which does exactly that, and can be integrated into other applications if converted into a library.

## 4.4 Results of attempt to automate testing for network changes

### 4.4.1 Introduction

Mobile devices are expected to be move during operation (in contrast to laptops which are generally mostly stationary or move very little when in operation) and hence are expected to change networks often. A mobile device is expected to switch from WiFi to WiFi network or from WiFi network to cellular and back to WiFi. While the transition from one network to another happens, transmission of data is expected to momentarily stop. Mobile applications are expected to handle this time period of no network connectivity "gracefully", in a way that doesn't negatively impact the user experience significantly. For example, web page downloads must not be interrupted while this transition happens, and if they are interrupted (because the transition took too long), the webpage must be reloaded automatically instead of requiring from the user to do it. Indeed, the "Chrome for Android" application behaves in this way. For that reason, a way to test applications against network changes must exist, preferably in an automated manner.

### 4.4.2 Shell commands that are of interest to testing of network changes

Since achieving actual loss of connectivity at the network adapter driver level requires root access to the device, which is unavailable in most of the devices, a better approach is to simulate loss of connectivity by disabling the network connection. The connection can then be re-enabled. The adb commands to accomplish this are:

For WiFi:

```
adb shell svc wifi enable

adb shell svc wifi disable
```

For Cellular:

```
adb shell svc data enable

adb shell svc data disable
```

Where data (short for "mobile data") refers to cellular network connection.

### 4.4.3 Automation of network changes in mobile applications

The official emulator provides little in the way of testing against this scenario. Although adb commands for disabling connectivity (WiFi and Cellular) are provided, this must be done manually. Ideally, this would happen in a "scripted" manner, which can better simulate the

momentary loss of connectivity during network changes. A tool for testing an application against network changes should provide the following features to the tester:

- Simulate loss of WiFi connectivity and re-enable WiFi connectivity
- Simulate loss of cellular connectivity (GSM connectivity) and re-enable WiFi connectivity
- Simulate transition from WiFi to Cellular and back to WiFi

The above can happen in a sequence.

### 4.4.4 Design of a tool to test against network changes

The GUITestAutomation software was extended to also test against network changes. The existing "sequencer" of GUITestAutomation can be used to issue the commands described above to the adb. Another advantage of this approach is that (much like the case of the fuzz testing), testing against network changes can be smoothly integrated into the GUI testing routine for an application. In order to accomplish this, one more child class of the *GUIcommand* class was added (called *GUIcommandNetwork*). The two fields of the class store the type of connection to be manipulated (WiFi or Cellular) and whether the connection should be enabled or disabled. The constructor of the class creates the appropriate adb command for each case.

The value of the testing tool lies mainly in the fact the commands it generates can be integrated into the UI testing sequence of commands, making for a more complete black-box testing of the application. However, an attempt will be made to experimentally evaluate the tool by itself, comparing it against manual testing.

### 4.4.5 Identity of the experiment

The experiment involves repetitions of simulating the transition from WiFi to Cellular and back to WiFi. Since the Android emulator cannot emulate a WiFi connection (even if the host machine is connected to the internet via WiFi, the Android Emulator will make it appear to Android OS it is running as a cellular connection. For that reason, those tests had to be done on a physical device connected via USB. The simulation was performed manually (pasting the adb commands to the cmd by hand) and with the use of the tool, for a number of NoR values, starting from 1.

### 4.4.6 Results of experiment

**Simulating transition from WiFi to Cellular to WiFi:**

| **Task:** | **NoR=1** | **NoR=2** | **NoR=3** | **NoR=4** | **NoR=5** | **NoR=6** |
|---|---|---|---|---|---|---|
| **Manual testing** | | | | | | |
| Time spent for this repetition (use fastest manual time for all reps): | 15.45 | 15.45 | 15.45 | 15.45 | 15.45 | 15.45 |
| Total time spent for manual testing | 15.45 | 30.90 | 46.35 | 61.80 | 77.25 | 92.70 |

| | | | | | | |
|---|---|---|---|---|---|---|
| of this use case: | | | | | | |
| **Automated black-box testing** | | | | | | |
| Tool setup time (create command): | 4.88 | --- | --- | --- | --- | --- |
| Script execution time: | 5.03 | 5.03 | 5.03 | 5.03 | 5.03 | 5.03 |
| Time spent for this repetition (setup plus script execution time): | 9.91 | 5.03 | 5.03 | 5.03 | 5.03 | 5.03 |
| Total time spent for automated testing of this use case: | 9.91 | 14.94 | 19.97 | 25 | 30.03 | 35.06 |

Table 8: Network changes testing, Simulating transition from WiFi to Cellular to WiFi

### 4.4.7 Comment on results

It is obvious from the table above that time-savings are achieved from the first repetition. This is to be expected, as any kind of testing that involves WiFi will have to be done on a physical device, which means the tester has to forgo the controls offered by the Android Emulator and instead manually issue adb commands. The automated tool relieves the tester from this, and for that reason the experiment is considered successful.

## 4.5 Results of attempt to automate energy consumption measurement

### 4.5.1 Introduction

In early cellphones, the major contributor to power consumption was the GSM module, the module responsible for transmitting and receiving data to and from the cell tower. However, in the case of modern smartphones, tablets, and most modern cellphones, there are lots of other factors are contributing to power consumption. Those devices are essentially handheld computers with a GSM module attached, which means that there are more factors contributing to power draw. Some of the factors contributing to power consumption in a modern device are:

1) **The screen:** The screen consumes over 50% of the power on any given moment (except when the device is on standy/sleep and the screen is disabled), on an average brightness setting. The complexity of software running does not play a factor in the power consumption of the screen. In fact, on LCD screens, power consumption is constant (or practically constant) regardless of the content shown, because the vast amount of power draw on an LCD screen is caused by the backlight, which is turned on at a constant brightness regardless of the content displayed in the TFT film in front of it. On AMOLED screens, the complexity of animation presented doesn't play a factor, but its appearance (what is displayed on screen) does. AMOLED screens are consisted of tiny LED elements (one LED for each subpixel), which are turned off when black needs to be produced, are turned on when 100% intensity needs to be produced, and are rapidly turned on-and-off when some intermediate intensity needs to be produced, using pulse-width modulation. The direct effect of this is that significant savings in battery life can be achieved if dark-colour themes are used for the UI of the application. Beyond that however, there is no real reason to measure the battery consumption of the screen when designing mobile applications.

2) **The CPU:** A major contributor to power consumption is the CPU. Mobile CPUs have a wide range of frequencies in which they operate, with the ability to scale down to a couple of MHz when not in use in order to conserve power and the ability to scale their frequencies to even higher than their nominal fequency when computing power is required and when the "thermal headroom" of the device allows it (in other words, when the device is cool enough), which however results in a dramatic increase in power consumption. For that reason, it is essential that CPU usage is taken into account when designing a mobile application. For example, content updates and checks for messages must be carefully designed to achieve a balance between content "freshness" and CPU usage. Also, activities like animations should not happen when the particular window or tab is not in view. It is also worth mentioning that CPU time can be used (even at full frequency) even when the device is on standby, using a feature of the Android OS called "wakelocks". As a result, a reliable way to measure CPU usage needs to exist when developing mobile applications.

3) **The GPU:** Mobile GPUs can sometimes exceed the die-size of the CPU cores on the Soc (System-on-a-chip) of the mobile device, so it naturally follows that their power

consumption is as significant, if not more, as the power consumption of the CPU. However, a peculiarity of GPUs is that they almost always work at either peak power or minimal power. This is because a GPU is expected to draw as many frames per second as possible, in order to achieve a good framerate. As a result, special care needs to be taken when designing an application with lots of elements in permanent motion (even when the user is not actively interacting with the app), such as for example animated backgrounds and animated buttons. In fact, the Android OS used to provide for the ability to have animated backgrounds in the home screen, but the feature was seldomly used by manufacturers precisely because of its high power consumption. For those reasons, the ability to measure when the GPU is "active" would be a major help for the developer in optimizing the power consumption of the application.

4) **The network:** The network, either via the GSM module or via WiFi, is another major contributor to power consumption. Only applications that have requested and received the "create internet sockets" connection can use the network. The permission makes no distinction between GSM module and WiFi (this means that once the application has been granted the permission, it can use both types of network connectivity). The user however can selectively turn off the GSM module or WiFi, or both. The OS always places priority on using the WiFi instead of the GSM module. In both cases however, power consumption can be considerable, and most importantly, can happen when the device is on standby. For that reason, a way to measure the additional power draw caused by both types of network connectivity needs to exist.

5) **The GPS receiver:** Applications which have been granted access to the "detailed location" permission can use the GPS receiver of the device to access a more fine-grained location, and contribute to a device's power consumption that way. It is of interest that this kind of power consumption can also happen when the device is on standby, so measuring GPS usage is important. The "coarse location", which is acquired by the cell tower ID the device is connected to, is always available to the device (as long as it is not in flight mode) and hence an application requesting for the "coarse location" does not cause an increase in power draw. Hence, only the increase in power draw caused by "detailed location" (GPS receiver location) should be measured.

6) There are other ways in which the power consumption of a mobile device can increase, such as the speakerphone, the vibration motor and illumination of hardware buttons (where such features exist). However, such features contribute minimally to power consumption, and the increase happens for only a short amount of time in most cases, so typically, most developers don't feel the need to measure this kind of power consumption and optimize for it.

### 4.5.2 Measuring power consumption on Android

The Android OS SDK provides the Android Monitor, a comprehensive tool for measuring CPU, GPU, GPS usage, and Network Usage of a particular running application. Combined with the automated GUI testing tools (those provided by the Android official tools and the tools designed

for the purposes of this dissertation), we have a set of tools which can provide a starting point to measuring the power consumption of a running application, by running the application repeatedly a specific number of times, for a specific usage script (scenario) each time, and taking an average CPU, GPU, Network Usages and GPS usages of the application.

However, such measurements, which measure the amount of CPU used, or the duration of time that the GPU, GPS or network was used, do not immediately translate into consumed milliwatts. In fact, since Android devices are powered by widely different kinds of SoCs, manufactured on different processes and with different types of cores as well as different number of cores, both when it comes to the CPU and the GPU, a precise formula that would fit all devices is impossible, even in the case where only a rough approximation is required.

### 4.5.3 Proposed method for measuring power consumption

One particular approach to measuring power consumption is to collect experimental usage data per SoC. However, the challenge of this method would be to isolate each measurement (CPU, GPU, GPS, and network) from each other. In the case of CPU usage measurement, isolation is easy to achieve as all it takes is an application that uses maximum CPU cycles without any graphics animations, and having no network or GPS permissions. By measuring the power consumption of maximum cycles on a finite amount of time, an experimental result of power consumption for that device can be derived.

When it comes to measuring GPU usage however, isolation between CPU and GPU time is not as easy to achieve, as the OpenGL or OpenGL ES APIs used to issue drawing commands to the GPU will, by definition, also use some amount of CPU time. The solution to this problem is to use Android monitor to measure the CPU time used, and substract the mah/s that correspond to that usage.

By using the methods described above, CPU-only power draw data and GPU-only power draw data can be collected, so that a “profile” of the SoC can be constructed, and a formula that would take as input the CPU usage percentage per second and the GPU usage per second, and outputs the power consumption per second can be devised.

Network usage can be derived in a similar way, by stressing the network (for example downloading a very large file) an subtracting the mah/s corresponding to the CPU time used during that period of time. Of course, a separate series of experiments have to be performed for when the device is operating on GSM network and on WiFi network. Only WiFi power usage was measured for this study.

Measuring GPS power draw is more involved, as there is no such thing as using “maximum GPS transfer rate” (as is the case with network) or “maximum CPU/GPU cycles” (as is the case with the CPU and GPU). The device used the GPS receiver at its maximum potential (power draw) for as long as a reliable lock (fix) is achieved, and then power consumption is reduced, all this managed by the SoC’s drivers. So, even if the application is constantly polling the GPS receiver for location, there is no guarantee that the maximum power draw figure for that device will be achieved. A possible way to remedy this is to perform the test in a location where the GPS receiver is very unlikely to obtain a lock, for example inside a room with no windows.

The above measurements could give a good “profile” of the power consumption of a device. Of course, as part of this dissertation, only a small amount of devices can be studied, as a proof of concept of the method.

### 4.5.4 **Implementation**

The implementation of the method is consisted of two parts:

1) A set of applications which are meant to be run on a physical device. The (nominal) maximum capacity of the device (in mAh) must be known beforehand. A set of applications designed to stress a particular resource (CPU, GPU, Network, GPS) to its maximum (100%) are installed in the devices. The applications themselves are compatible with Android 4.0.3 (ICS) and higher. The applications are meant to “stress” each component of the device as much as possible, while at the same time stressing the other components of the devices to a minimal degree. In total, 3 applications where developed:
   - SpinThread: A basic application which is meant to increase the workload of all the CPU cores of the SoC to 100%. It achieves this by creating one thread per CPU core and running the CPU core at maximum by running a while(1). It is worth nothing that this test is optimally run with the screen dimmed completely, in order to not have the power draw from the screen itself interfere with the test.
   - Graphics3D: A basic 3D application which stresses the GPU of the SoC, by rotating a 3D cube (with texture-mapped) images at a high rate. The application makes use of OpenGL ES, in order to utilitize the hardware-accelerated 3D capabilities of the SoC. The OpenGL ES functions are called from within the Java language. It is worth noting that it is not possible to use the GPU without using some of the CPU too. For that reason, an estimated 12% CPU usage exists when running Graphics 3D (as measured by Android Monitor for all devices participating in the experiment), which must be accounted for.
   - Gpsbasics: This is a simple application which attempts to “lock” into the GPS as many times as possible, as a way to “stress” the GPS receiver of the device as much as possible. It is worth noting that this test must also be run with the screen turned-off or dimmed to a maximum.

   The user is expected to run the applications on a mobile device for a finite amount of seconds. After that, a mah/second usage number can be derived for that device, which is an energy measurement that corresponds to the amount of energy consumed when using that resource (CPU, GPU, GPS or network) to 100%. After that, energy measurements for smaller values of usage (50%, 30% etc) can be derived, by assuming (mostly) linear scaling.

2) The second part of the implementation is consisted of a desktop application, written in Java and using the Swing framework for UI creation. The purpose of the application is to take raw data from the measured by the user and calculate an estimate of the device’s consumption (in mAh) per second, for that particular resource. The application asks the user to input one of the 4 main types of power-intensive resources he wishes to measure: CPU, GPU, GPS, WiFi (note: athough no purpose-built application to “stress” the WiFi has

been created, this can be done simply by downloading a large file.) and select the device the experiment was made on.
After that, the user must input the total capacity of the device, the reduction (in percentage points) of the battery, and the duration of the test (in seconds). The application then calculates the mAh consumed per second for that resource. In the case of GPU power consumption, the 12% consumed by the CPU is not taken into account (it is not “charged” to the GPU).

The application itself was written in Java using Netbeans. It is consisted of a main window which is launched by the main method of the application. The main window collects user input and then calls the methods askCPUcalc, askGPUcalc, askGPScalc and askWiFicalc to calculate CPU, GPU, GPS and WiFi power consumption accordingly (one of the four). After that, a second window is launched, which informs the user about the mah per second consumed.

### 4.5.5 Results of experiment

By applying the method above, a set of values can be derived, which can be considered as the “profile” of the particular device. Those values, combined with Android monitor, can help a developer predict the power consumption his application will have. As a proof-of-concept, experiements for CPU and GPU usage were made.

Device 1: HTC EVO 3D (low-end device as of 2016) (battery capacity: 1730Mah)

| **Resource:** | **Seconds:** | **Battery before:** | **Battery after:** | **Reduction:** | **Mah/second:** |
|---|---|---|---|---|---|
| CPU | 600 | 100 | 96 | 4 | 0.1153 |
| CPU | 600 | 96 | 92 | 4 | 0.1153 |
| CPU | 600 | 92 | 89 | 3 | 0.0865 |
| CPU | 600 | 89 | 86 | 3 | 0.0865 |
| CPU | 600 | 86 | 83 | 3 | 0.0865 |
| CPU | 600 | 83 | 80 | 3 | 0.0865 |
| CPU | 600 | 80 | 77 | 3 | 0.0865 |
| CPU | 600 | 77 | 70 | 3 | 0.0865 |
| CPU | 600 | 70 | 67 | 3 | 0.0865 |
| CPU | 600 | 67 | 64 | 3 | 0.0865 |
| CPU | 600 | 64 | 61 | 3 | 0.0865 |
| CPU | 600 | 61 | 58 | 3 | 0.0865 |

| | | | | | |
|---|---|---|---|---|---|
| CPU | 600 | 58 | 55 | 3 | 0.0865 |
| CPU | 600 | 55 | 52 | 3 | 0.0865 |
| CPU | 600 | 52 | 49 | 3 | 0.0865 |

| **Resource:** | **Seconds:** | **Battery before (%):** | **Battery after (%):** | **Reduction (%):** | **Mah/second:** |
|---|---|---|---|---|---|
| GPU | 600 | 100 | 94 | 6 | 0.1729 |
| GPU | 600 | 94 | 89 | 5 | 0.1441 |
| GPU | 600 | 89 | 84 | 5 | 0.1441 |
| GPU | 600 | 84 | 79 | 5 | 0.1441 |
| GPU | 600 | 79 | 74 | 5 | 0.1441 |
| GPU | 600 | 74 | 70 | 4 | 0.1153 |
| GPU | 600 | 70 | 66 | 4 | 0.1153 |
| GPU | 600 | 66 | 62 | 4 | 0.1153 |
| GPU | 600 | 62 | 58 | 4 | 0.1153 |
| GPU | 600 | 58 | 44 | 4 | 0.1153 |
| GPU | 600 | 54 | 50 | 4 | 0.1153 |

Device 2: Samsung Galaxy S3 (midrange device as of 2016) (battery capacity: 2100Mah)

| **Resource:** | **Seconds:** | **Battery before:** | **Battery after:** | **Reduction:** | **Mah/second:** |
|---|---|---|---|---|---|
| CPU | 600 | 84 | 78 | 6 | 0.2100 |
| CPU | 600 | 78 | 74 | 4 | 0.1400 |
| CPU | 600 | 74 | 70 | 4 | 0.1400 |
| CPU | 600 | 70 | 66 | 4 | 0.1400 |
| CPU | 600 | 66 | 62 | 4 | 0.1400 |
| CPU | 600 | 62 | 58 | 4 | 0.1400 |
| CPU | 600 | 58 | 54 | 4 | 0.1400 |
| CPU | 600 | 54 | 50 | 4 | 0.1400 |

| **Resource:** | **Seconds:** | **Battery before:** | **Battery after:** | **Reduction:** | **Mah/second:** |
|---|---|---|---|---|---|
| GPU | 600 | 100 | 91 | 9 | 0.3149 |
| GPU | 600 | 91 | 84 | 7 | 0.2449 |
| GPU | 600 | 84 | 77 | 7 | 0.2449 |
| GPU | 600 | 77 | 70 | 7 | 0.2449 |
| GPU | 600 | 70 | 63 | 7 | 0.2449 |
| GPU | 600 | 63 | 56 | 7 | 0.2449 |
| GPU | 600 | 56 | 50 | 6 | 0.2099 |

#### 4.5.6 Comment on results

While a very early attempt, it was proven that amount of mah/sec a given application will draw per second can be precalculated, given the profile of a device. Hence, the mah an application will draw on popular devices can be pre-calculated by developers (as long as a profile has been made for them), and power draw can be optimised.

This is possible because consistency of power draw is relatively high regardless of the initial condition (charge) of the battery.

# Chapter 5: Legal and commercial aspects

## 5.1 Legal Aspects

When using an operating system produced by a commercial entity, certain legal aspects regarding the copyright status of the OS and its tools arise. This happens because most commercial entities adopt a “mixed-model” of licensing where part of the OS is open source and part is proprietary software. In order to avoid this issue, only the open-source parts of the Android OS were used, and any dependence on APIs implemented by proprietary software (such as the Google APIs implemented by the Play Services background services) was avoided. This means that the product of this research applies to any Android variant derived from the AOSP (Android Open Source Project) code, and is not dependent on the offerings of a commercial company.

One piece of proprietary software (Google Maps) was used as an application to be tested, but it is not a software dependence.

In addition, all code that implements the tools that were created as part of this dissertation is custom-built, and hence can be licensed under any license and for any use.

## 5.2 Commercial aspects

With mobile applications getting more complex and with mobile application UIs becoming more elaborate, the need for better testing tools that save testers time is increasing. Hence, there is a clear potential that the prototype software created as part of this dissertation can evolve into a commercial mobile app testing package that will plug gaps in functionality and testing features not offered by the official tools, especially for the case of Native apps.

In addition, the research into the measuring of energy consumption is a mostly untouched field by academic research which can evolve into a promising commercial tool if developed as a commercial solution.

# Chapter 6: Conclusion

The goals of this study were:

1) To review the existing tools and testing methods for mobile applications and identify any gaps, by examining the ones currently used for application development on the most popular operating system, the Android operating system. In particular, areas of testing that are unique to mobile applications were studied. More specifically: GUI testing, Location services testing, Security of mobile applications, Testing against network changes and measuring energy consumption. The study succeeded in that aim, as a comprehensive analysis of the available testing methods was done, both in the field of white-box testing and black-box testing, and several gaps were identified, particularly in the fields of black-box GUI testing, location-services testing for navigation apps, fuzz-testing of mobile applications and checking for vulnerabilities, and measuring energy consumption in mah/sec in a physical device.

2) Propose technical solutions and software to plug those gaps, in order to increase the effectiveness of testing and decrease the time required. During this study, software that helps decrease testing time and increase testing effectiveness was created. For example, the GUITestAutomation tool and the tool for testing against network changes helps developers and testers achieve significant time savings when testing their mobile applications. The security-testing tools as well as the method and software proposed to measure energy consumption in mobile devices helps developers and testers increase the quality of their testing by testing for things like robustness against unexpected input and energy consumption accordingly, which are things which the official tools and methods do not provide a method to test against. In addition, the GPSTestAutomation tool helps developers increase both the quality of their testing, by providing a better way to test apps which need location services input and helps developers reduce testing time significantly. Hence, the study succeeded in this goal to a very large degree.

3) A measurement of the savings in testing time and gains in effectiveness (testing coverage) achieved with the new software was made. All new software has its testing capabilities experimentally validated against real mobile applications, via a series of experiments. The study provided a comprehensive overview of those experiments and comments on their results, while the experiments themselves were successful.

Overall, the study managed to accomplish its goals and aims, which is to further the effectiveness and efficiency of testing for mobile applications.

Additionally, a series of further research opportunities were uncovered, which can provide further gains in effectiveness and efficiency of mobile application testing in the future.

# Appendices

# Appendix A

**Codes for keypresses:**

0 --> "KEYCODE_UNKNOWN"

1 --> "KEYCODE_MENU"

2 --> "KEYCODE_SOFT_RIGHT"

3 --> "KEYCODE_HOME"

4 --> "KEYCODE_BACK"

5 --> "KEYCODE_CALL"

6 --> "KEYCODE_ENDCALL"

7 --> "KEYCODE_0"

8 --> "KEYCODE_1"

9 --> "KEYCODE_2"

10 --> "KEYCODE_3"

11 --> "KEYCODE_4"

12 --> "KEYCODE_5"

13 --> "KEYCODE_6"

14 --> "KEYCODE_7"

15 --> "KEYCODE_8"

16 --> "KEYCODE_9"

17 --> "KEYCODE_STAR"

18 --> "KEYCODE_POUND"

19 --> "KEYCODE_DPAD_UP"

20 --> "KEYCODE_DPAD_DOWN"

21 --> "KEYCODE_DPAD_LEFT"

22 --> "KEYCODE_DPAD_RIGHT"

23 --> "KEYCODE_DPAD_CENTER"

24 --> "KEYCODE_VOLUME_UP"

25 --> "KEYCODE_VOLUME_DOWN"

26 --> "KEYCODE_POWER"

27 --> "KEYCODE_CAMERA"

28 --> "KEYCODE_CLEAR"

29 --> "KEYCODE_A"

30 --> "KEYCODE_B"

31 --> "KEYCODE_C"

32 --> "KEYCODE_D"

33 --> "KEYCODE_E"

34 --> "KEYCODE_F"

35 --> "KEYCODE_G"

36 --> "KEYCODE_H"

37 --> "KEYCODE_I"

38 --> "KEYCODE_J"

39 --> "KEYCODE_K"

40 --> "KEYCODE_L"

41 --> "KEYCODE_M"

42 --> "KEYCODE_N"

43 --> "KEYCODE_O"

44 --> "KEYCODE_P"

45 --> "KEYCODE_Q"

46 --> "KEYCODE_R"

47 --> "KEYCODE_S"

48 --> "KEYCODE_T"

49 --> "KEYCODE_U"

50 --> "KEYCODE_V"

51 --> "KEYCODE_W"

52 --> "KEYCODE_X"

53 --> "KEYCODE_Y"

54 --> "KEYCODE_Z"

55 --> "KEYCODE_COMMA"

56 --> "KEYCODE_PERIOD"

57 --> "KEYCODE_ALT_LEFT"

58 --> "KEYCODE_ALT_RIGHT"

59 --> "KEYCODE_SHIFT_LEFT"

60 --> "KEYCODE_SHIFT_RIGHT"

61 --> "KEYCODE_TAB"

62 --> "KEYCODE_SPACE"

63 --> "KEYCODE_SYM"

64 --> "KEYCODE_EXPLORER"

65 --> "KEYCODE_ENVELOPE"

66 --> "KEYCODE_ENTER"

67 --> "KEYCODE_DEL"

68 --> "KEYCODE_GRAVE"

69 --> "KEYCODE_MINUS"

70 --> "KEYCODE_EQUALS"

71 --> "KEYCODE_LEFT_BRACKET"

72 --> "KEYCODE_RIGHT_BRACKET"

73 --> "KEYCODE_BACKSLASH"

74 --> "KEYCODE_SEMICOLON"

75 --> "KEYCODE_APOSTROPHE"

76 --> "KEYCODE_SLASH"

77 --> "KEYCODE_AT"

78 --> "KEYCODE_NUM"

79 --> "KEYCODE_HEADSETHOOK"

80 --> "KEYCODE_FOCUS"

81 --> "KEYCODE_PLUS"

82 --> "KEYCODE_MENU"

83 --> "KEYCODE_NOTIFICATION"

84 --> "KEYCODE_SEARCH"

85 --> "TAG_LAST_KEYCODE

# Glossary of Terms

**Adb:** A collection of executable binaries which send debug commands to the Android emulator, or to a physical device connected to the development machine via USB and which has "USB debugging" mode enabled. "Sending commands to the adb" means launching an adb client with the appropriate command-line arguments, which describe the debug command to be sent. The debug command will then be sent to the server, and then be sent to the daemon running on Android OS in order to be executed.

**Android Emulator:** An emulator which allows the execution of an Android OS image inside a hosted OS, most commonly Windows, OS X or some variant of Desktop Linux

**Android OS:** Used to refer either to the Android OS itself, or an installation image of Android running inside a physical machine or inside the Android Emulator.

**AOSP browser:** A relatively simple browser application that comes with the Android OS source code and doesn't depend on any Google APIs or Google proprietary components.

**Cellular data:** Refers to network connectivity using the tower of a cellular operator.

**Fuzz Testing:** Inserting random characters covering the full spectrum of characters an OS allows (in the case of Android, the full Unicode 16-bit spectrum) in order to test the robustness of applications and OS utilities against special characters, non-printable characters and control characters.

**SoC:** System on a chip. A chip inside the device which includes most of the logic functions of a device, including the CPU, GPU, memory manager, networking (on some devices, on some it is a separate chip), GPS receiver, DAC (Digital to Analog Converter) and ADC (Analog to Digital Converter), and the DSP for the camera. It is the chip which consumes most of the battery power of a device (compared to other chips) during normal operation.

# List of References

References are sorted by the order they are found on the main text.

JOORABCHI: M. E. Joorabchi, A Mesbah, P Krunchten, "Real Challenges In Mobile App Development". *ACM/ IEEE International Symposium on Empirical Software Engineering and Measurement* 1.1: p. 19-21.

MUCCINI: H. Muccini, A. D. Francesco, P. Esposito, "Software Testing Of Mobile Applications: Challenges And Future Research Directions". *IEEE* 1.1: p. 33.

KAIKKONEN: A. Kaikkonen, A. Kekäläinen, M. Cankar, T. Kallio, A. Kankainen, "Usability Testing Of Mobile Applications: A Comparison Between Laboratory And Field Testing". *Journal of Usability Studies* 1.1

MAHMOOD: R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, A. Stavrou, "Whitebox Approach For Automated Security Testing Of Android Applications On The Cloud". *IEEE* 1.1: p. 22-27.

ANDROID: "Application Fundamentals | Android Developers". Developer.android.com. Web (https://developer.android.com/guide/components/fundamentals.html)

NATIVE: "Native App  Concepts, Android Developers". Developer.android.com. Web (https://developer.android.com/ndk/guides/concepts.html)

FRAMEWORK: "Android Framework - App Components, Android Developers". Developer.android.com. Web (https://developer.android.com/guide/components/index.html)

APK: "APK Files And App Install Location, Android Developers". Developer.android.com. Web (https://developer.android.com/guide/topics/data/install-location.html)

ADB: "Android Debug Bridge". Developer.android.com. Web (https://developer.android.com/studio/command-line/adb.html)

ADB SHELL:  "ADB Shell Commands". *Developer.android.com*. Web (https://developer.android.com/studio/command-line/shell.html)

UNIT TESTING: "Building Local Unit Tests, Android Developers". Developer.android.com. Web (https://developer.android.com/training/testing/unit-testing/local-unit-tests.html)

MOCKITO: "The Mockito Framework". *Site.mockito.org*. Web (http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html)

ESPRESSO:  "Espresso Testing, Android Developers". Developer.android.com. Web (https://developer.android.com/training/testing/ui-testing/espresso-testing.html)

GOOGLE TEST: "A Quick Introduction To The Google C++ Testing Framework". Ibm.com. Web (http://www.ibm.com/developerworks/aix/library/au-googletestingframework.html)

UI AUTOMATOR"UI Automator, Android Developers". Developer.android.com. Web (https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html)

# Bibliography


- *Software Test Automation,* Mark Fewster and Dorothy Gragam (Addison-Wesley press)

- *Testing Computer Software* textbook by Cem Kaner, Jack Falk and Hung Quoc Nguyen (Wiley press)