

Android Sensor Simulator

&

Developer Tool

Ronan Watkins

B.Eng. (Honours) Computer & Electronic Engineering

Galway-Mayo Institute of Technology

2017/2018

# Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering in Software and Electronic Engineering at Galway-Mayo Institute of Technology.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

# Acknowledgements

I am grateful to my family, friends, and the staff at GMIT for their help, support, and guidance throughout this project. I would like to thank in particular my supervisor Niall O’Keeffe for helping me with my project and Michael Murray for giving me the idea when I thought all hope was lost.

Lastly, I would like to thank everyone in the graduating class of 2018 in Computer and Electronic Engineering. It has been an enjoyable 5 years and I wish everyone the best in their future.

# Summary

This project is an Android sensor simulator and developer tool. It is essentially a Graphical User Interface (GUI) wrapper around the Android Debug Bridge (ADB) command line tool and Telnet interface into the Android emulator.

The project is a desktop application written in JavaFX using the Intellij IDE with an Android application to compliment one of its features written in the Android Studio IDE.

It is intended for use on a Windows machine and requires the Android SDK which includes the Android QEMU (Quick Emulator) and ADB executable to get full advantage of the features provided by this application.

The application takes the form of a tabbed GUI split into 8 tabs. The first 3 tabs contain the sensor simulator feature, location simulation feature and the third tab contains components to allow the user to send calls and text messages to the emulator, change network state and simulate changes in battery health and charging status. These features are a wrapper around the Telnet interface into the Android emulator and allow developers and testers of Android applications a way of easily testing the behavior of Android applications under certain conditions.

The 5 other tabs contain the console, automation, applications, LogCat, and monitor features. These features are a wrapper around the ADB command line tool and can be used on emulators and real devices. The device may be plugged into the computer or connected over Wi-Fi through ADB to use these features. I will describe these features in detail throughout this report.

This project was completed successfully with all features working as expected.

Contents

[Declaration 2](#_Toc513394407)

[Acknowledgements 3](#_Toc513394408)

[Summary 4](#_Toc513394409)

[1. Introduction 8](#_Toc513394410)

[1.1 Project Goals 8](#_Toc513394411)

[1.2 Project Motivation 8](#_Toc513394412)

[1.3 About Android 8](#_Toc513394413)

[1.4 About JavaFx 8](#_Toc513394414)

[1.5 Report Overview 9](#_Toc513394415)

[2. Architectural Block diagram 10](#_Toc513394416)

[3. Android: A high level overview 11](#_Toc513394417)

[3.1 Linux Kernel 12](#_Toc513394418)

[3.2 Dalvik Virtual Machine 12](#_Toc513394419)

[3.3 Applications 13](#_Toc513394420)

[4. The Android debug bridge 14](#_Toc513394421)

[5. Telnet and the Android emulator 15](#_Toc513394422)

[6. Sensor simulation 17](#_Toc513394423)

[6.1 Reuse of Openintents’ code 17](#_Toc513394424)

[6.2 Android application 19](#_Toc513394425)

[6.3 Recording and playback of sensor data 21](#_Toc513394426)

[6.4 How it works 23](#_Toc513394427)

[6.4.1 Sending sensor data 23](#_Toc513394428)

[6.4.2 Recording sensor data 24](#_Toc513394429)

[6.4.3 Playback of Sensor data 25](#_Toc513394430)

[7. Location simulation 26](#_Toc513394431)

[7.1. Keyhole Markup Language 26](#_Toc513394432)

[7.2. Google Maps Integration 28](#_Toc513394433)

[7.3. How it works 29](#_Toc513394434)

[7.3.1 Sending geolocation commands 29](#_Toc513394435)

[7.3.2 Adding to the KML file 30](#_Toc513394436)

[7.3.3 Reading from the KML file 30](#_Toc513394437)

[8. Telephony, network, and power events 32](#_Toc513394438)

[8.1 Telephony 32](#_Toc513394439)

[8.2 Network 34](#_Toc513394440)

[8.3 Power 36](#_Toc513394441)

[9. Console 38](#_Toc513394442)

[9.1 How it works 38](#_Toc513394443)

[10. Automation 40](#_Toc513394444)

[10.1 Creating automation file 41](#_Toc513394445)

[7.1 Importance of automation in development and testing 41](#_Toc513394446)

[7.2 Interpreting Hex data from Linux interface file 41](#_Toc513394447)

[7.3 Research into sendevents.c 41](#_Toc513394448)

[11. Applications 42](#_Toc513394449)

[11.1 What is an Intent? 42](#_Toc513394450)

[11.2 Activities 44](#_Toc513394451)

[11.2 Services 44](#_Toc513394452)

[11.3 Broadcast receivers 44](#_Toc513394453)

[11.4 How it works 45](#_Toc513394454)

[11.4.1 Parsing Intents 47](#_Toc513394455)

[11.4.2 Sending Intents 47](#_Toc513394456)

[12. LogCat 49](#_Toc513394457)

[12.1 How it works 52](#_Toc513394458)

[12.1.1 Reading from the logcat utility 52](#_Toc513394459)

[12.1.2 Creating a filter 53](#_Toc513394460)

[12.1.3 Filtering logs 53](#_Toc513394461)

[12.1.4 Saving logs 55](#_Toc513394462)

[13. Monitor 56](#_Toc513394463)

[13.1 Top (table of processes) 58](#_Toc513394464)

[13.2 PS (process status) 59](#_Toc513394465)

[13.3 /proc 59](#_Toc513394466)

[13.4 How it works 61](#_Toc513394467)

[13.4.1 Monitor 61](#_Toc513394468)

[13.4.2 Controller 61](#_Toc513394469)

[14. Conclusion 64](#_Toc513394470)

[14.2. References 65](#_Toc513394471)

[15. Appendices 66](#_Toc513394472)

# Introduction

## Project Goals

The goals for this project were as follows:

* To create an Android sensor simulator and developer tool that is easy to use and visually appealing.
* To build upon my knowledge of Java and GUI development.
* To learn more about the inner workings of the Android operating system and Android application development.
* To contain all features in an easy to use application

## Project Motivation

I have had an interest in Android and the Java programming language for some time now. This interest was heightened in the Mobile device programming module in semester one of the final year of my studies. This project was carried out in order to further my knowledge of the Java programming knowledge especially in the areas of GUI development in JavaFX as I found it to be a well put together platform for GUI development and in the area of Android development. The Android operating system I found to be architecturally brilliant and the idea of a Java interface sitting atop a Linux kernel designed to run on low power mobile devices was intriguing to me.

## About Android

Android is an operating system designed for mobile devices. Based on a modified version of the Linux kernel and other open source software and designed primarily for touchscreen mobile devices such as smartphones and tablets.

## About JavaFx

JavaFX is a software platform for creating and delivering desktop applications, as well as rich Internet applications (RIAs) that can run across a wide variety of devices. JavaFX is intended to replace Swing as the standard GUI library for Java SE, but both will be included for the foreseeable future. JavaFX has support for desktop computers and web browsers on Microsoft Windows, Linux, and macOS.[1]

## Report Overview

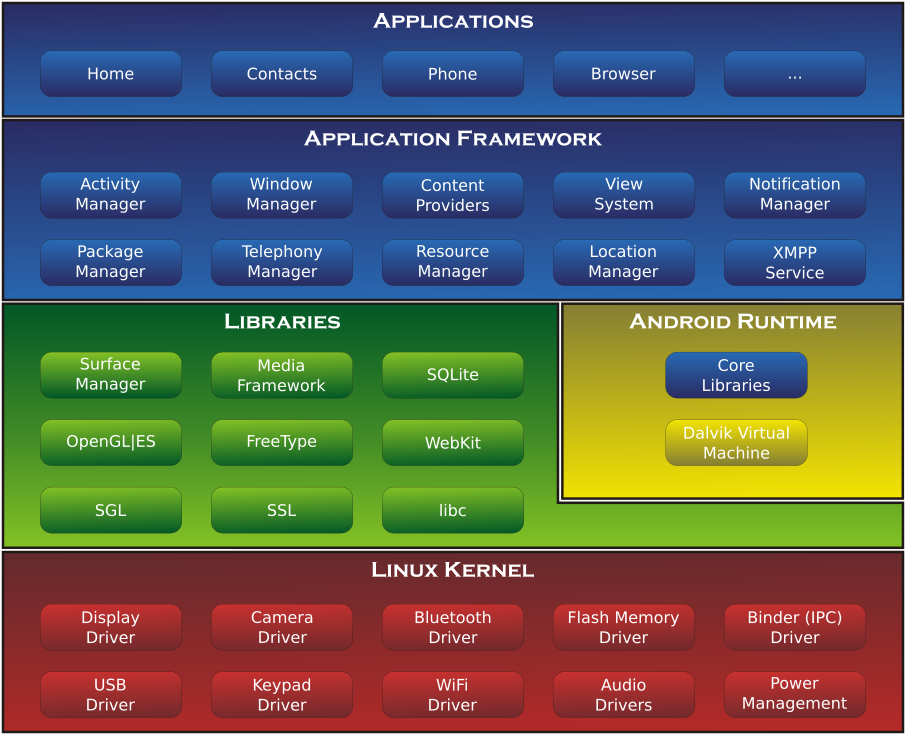
This report contains sections including the architectural diagram and body of report which details the main areas of the project. I have also included UML diagrams throughout this report. Code, programs, and file names and class names are highlighted in blue. All references throughout the body of this report are referenced in section ??, References.

The project poster, links to source code and documentation are attached as an appendix in section??

# A close up of a device Description generated with high confidenceArchitectural Block diagram

# Android: A high level overview

Android is architected in the form of a software stack comprising applications, an operating system, run-time environment, middleware, services, and libraries. This architecture is represented visually in the diagram below. Each layer of the stack, and the corresponding elements within each layer, are tightly integrated and carefully tuned to provide the optimal application development and execution environment for mobile devices.[2]

[3]

## 3.1 Linux Kernel

Android’s kernel is based on one of the Linux Kernel’s long-term support (LTS) branches. Android's variant of the Linux kernel has further architectural changes that are implemented by Google outside the typical Linux kernel development cycle, such as the inclusion of components like device trees, ashmem, ION, and different out of memory (OOM) handling. [4][5]

The kernel provides preemptive multitasking, low-level core system services such as memory, process, and power management in addition to providing a network stack and device drivers for hardware such as the device display, Wi-Fi, and audio.

The original Linux kernel was developed in 1991 by Linus Torvalds and was combined with a set of tools, utilities and compilers developed by Richard Stallman at the Free Software Foundation to create a full operating system referred to as GNU/Linux. Various Linux distributions have been derived from these basic underpinnings such as Ubuntu and Red Hat Enterprise Linux.

It is important to note, however, that Android only uses the Linux kernel. That said, it is worth noting that the Linux kernel was originally developed for use in traditional computers in the form of desktops and servers. In fact, Linux is now most widely deployed in mission critical enterprise server environments. It is a testament to both the power of today’s mobile devices and the efficiency and performance of the Linux kernel that we find this software at the heart of the Android software stack.[6]

## 3.2 Dalvik Virtual Machine

Since Android is not pure Java and is designed to run on a modified version of the Linux kernel, the Android developers decided to make its own runtime for Android applications instead of using the standard Java Virtual Machine (JVM). Unlike in standard Linux distributions, each application does not run as a separate process on the kernel. In fact, each application running on an Android device runs within its own instance of the Dalvik virtual machine. This virtual machine was design by Google to be more memory efficient that the standard JVM in terms of memory usage and be suitable for usage of low power, low memory mobile devices.

Unlike standard Java programs which are compiled into bytecode to be run on the JVM, Android applications are compiled into Dalvik executable (.dex) format, which has a 50% smaller memory footprint than standard Java bytecode.[7]

## 3.3 Applications

Developers write Android applications, normally through the Android derivative of the Java programming language. This may be combined with C/C++. The Go and Kotlin programming languages are now supported.

Android phones come as standard with a set of standard applications such as a web browser, email client, settings application etc. Users can choose to install third party applications, normally through the Play Store or other third-party application marketplaces.

As mentioned previously, applications run inside the Dalvik virtual machine as dex files. Applications are installed onto the device through Android Package (APK) files based on the Java Archive File (JAR). These files are a type of archive file containing the dex files, resources, assets, certificates, and manifest file.[8]

# The Android debug bridge

The Android Debug Bridge (ADB) is a toolkit included in the Android software development kit (SDK) package. It consists of both client and server-side programs that communicate with one another. The ADB is typically accessed through the command-line interface.[9]

The ADB command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device. It is a client-server program that includes three components:

A client, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing an ADB command.

A daemon, which runs commands on a device. The daemon runs as a background process on each device.

A server, which manages communication between the client and the daemon. The server runs as a background process on your development machine.[10]

# Telnet and the Android emulator

The QEMU emulator provided with Android SDK provides a console that lets the developer query the emulated device and environment and environment. Functionality provided by the Telnet console are as follows:

* Sensor simulation by way of providing strings containing sensor data information
* Change network characteristics
* Simulate telephony events
* Change power and battery characteristics
* Manage port redirection

To initiate a connection with the Android emulator through Telnet you enter the following command onto a Telnet console:

telnet localhost console-port

The emulator will then respond with

Connected to localhost.

Escape character is '^]'.

Android Console: Authentication required

Android Console: type 'auth <auth\_token>' to authenticate

Android Console: you can find your <auth\_token> in

'/Users/me/.emulator\_console\_auth\_token'

OK

You must then open the file with a text editor and copy the alphanumeric string into telnet console window to authenticate your session with the emulator. The command looks as follows

auth PuKRUJpgUlET1BmX

After you are connected you can enter commands to controller and query the emulator state.

This application automates this process by creating a ServerSocket and connecting to ‘localhost’ on the port the emulator is listening on, often 5554 for the first emulator running and subsequent port numbers after, 5555, 5556 etc.

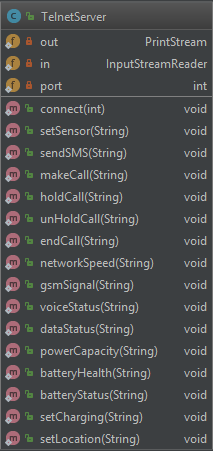
After a connecting is made, my application then creates an InputStreamReader to read the input from the emulator and PrintStream to output commands to the emulator.

The initial emulator response is read by the InputStreamReader and parsed to find the location of the file containing the authentication string ('/Users/me/.emulator\_console\_auth\_token').

The alphanumeric string contained in this file is then sent to the emulator via the PrintStream and the connection is created.

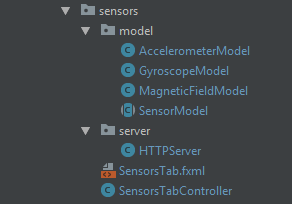
The class in my application with the task of communicating with emulator in my application is called TelnetServer. It contains static methods to send commands to the emulator via the PrintStream. The methods are accessed by the Device class to call the connect method upon connecting to an emulator, by the SensorTabController, LocationTabController and PhoneTabController to send commands to simulate sensor events, location details and telephony, network, and power events respectively.

A UML class diagram of the TelnetServer class is as follows:



# Sensor simulation

The sensor simulation feature has been divided up into 2 packages: one for the model of the orientation sensors and one for the HTTPServer to handle the connection with the Android application. These classes are divided as follows.



The first feature implemented was the Sensor Simulation feature and formed the basis of this project. This feature was inspired by OpenIntents Sensor Simulator [10]. OpenIntents Sensor Simulator was an open source project intended to provide sensor data to early versions of the Android emulator. This was done prior to Android providing the Telnet interface into the emulator and relied on the user modifying their application under development to work with Openintents’ modified version of the Android SensorManager class.

This application was written in Swing; A framework for developing GUIs in Java. This framework is being replaced by JavaFx and for this reason, JavaFX was chosen for this project

## Reuse of Openintents’ code

Three important elements of Openintents’ code were the simulation of magnetometer, gyroscope and accelerometer sensor data using 3 sliders to control the yaw, pitch and roll of the phone. A decision was made to incorporate this code into this project as it was found it to be well written and accurate. The simulated values were compared to real values by writing a simple Android application to print the values from the magnetometer, gyroscope and accelerometer as the phone was rotated. The values printed were compared to Openintents’ simulated values on the GUI and were found to be very similar.

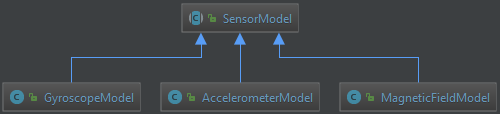
Openintents’ sensor simulator GUI followed a Model View Controller (MVC) design pattern, a common pattern for GUI and web development. This pattern divides the application into three interconnected parts. The controller accepts input and converts it to commands for the model or view. The view is responsible for presenting the output, sliders, labels etc. The model is independent of the user interface. It directly manages the data, logic and rules of the application[11] [Simple Example of MVC (Model–View–Controller) Design Pattern for Abstraction](http://www.codeproject.com/Articles/25057/Simple-Example-of-MVC-Model-View-Controller-Design)

The model element of this program was required for this application as this managed the data and logic responsible for simulating the sensor values based on input it received from pitch, yaw, and roll sliders.

In their application, this code was held in the SensorModel, AccelerometerModel, MagneticFieldModel and GyroscopeModel classes.

The SensorModel was an abstract class which, in object orientated programming terms meant it could not be instantiated and provided a description for how a class representing a sensor’s behaviour should behave.

The SensorModel, AccelerometerModel, MagneticFieldModel are extensions of this class with functionality on how their respective sensors work. A UML diagram of the relationship is below



## Android application

To compliment this feature, an Android application was developed that would send sensor data as a JavaScript Object Notation (JSON) formatted string over Wi-Fi to the host computer. Upon receiving this data, the host computer would relay the data to the emulator. This was found to be a useful feature for the simulation of orientation sensors on the emulator, a feature that would be useful for developers testing applications reliant on these sensors.

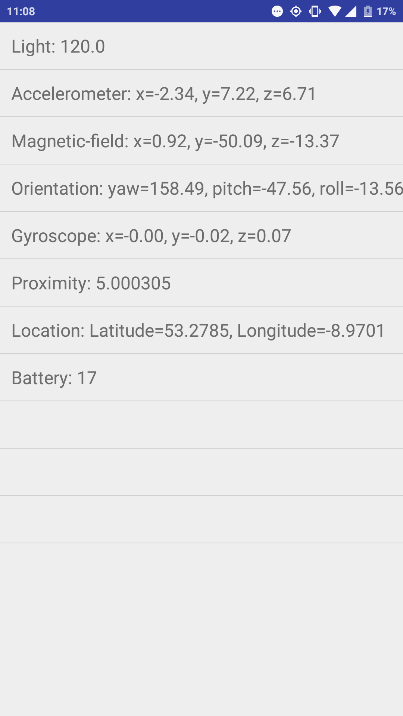
The use of this feature is optional to the user. To make use of this feature, this user simply clicks the ‘Connect’ button on the GUI and enters the IP and port number into their application to connect to the device. As soon as the connection is established, the device will start sending a JSON formatted string of sensor data at periods of 250ms to the host computer. A sample of this string is as follows:

{"light":"120.0","location":[53.2787,-8.97],"accelerometer":["-0.17","0.36","9.90"],

"magnetic-field":["-2.72","-24.59","-40.07"],"proximity":"5.000305",

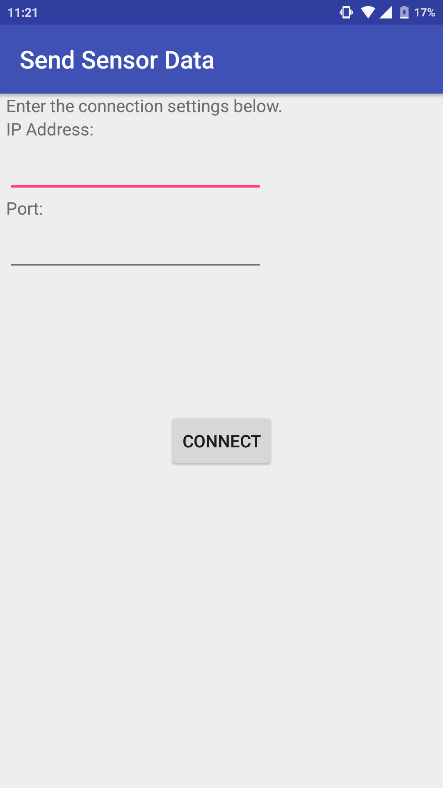
"orientation":["171.46","-2.15","-1.02"],"gyroscope":["-0.00","-0.00","-0.00"],"battery":"18"}

The application provides a read out of the sensor data in a list format like so:



SecondActivity

The Android application is relatively simple and divided into 4 classes: MainActivity, SecondActivity, SendSensorDataService and SQLLiteUtils

The MainActivity is the entry point to the Application. It displays 2 text fields for the user to enter the IP address and port number and a button to connect to the computer. A screen shot of it is below:

MainActivity

Upon pressing the connect button, the application sends a HTTP GET request to the computer. If a HTTP 200 OK response is received within the specified timeout of 5 seconds, the SecondActivity is activated and the SendSensorDataService is invoked. If a connection is failed to be established, an error message is displayed containing the message:

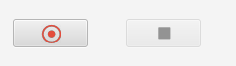
Host unreachable. Make sure the IP Address and port number are correct

SQLite[12], a relational database management system embedded into Android phones was used for persistent storage of the IP address and port number to make for a more pleasant user experience. Upon opening the application, a check is made to see if there is an IP address and port number in the database. If so, they are entered into the respective text fields. The reason being, it would be tedious for a user to enter the same data every time they wish to use the application. In the event the user enters a different IP address and port number to those stored in the database, the application will update the database with the new information.

## Recording and playback of sensor data

Since the idea for developing this feature comes from the need for application developers to test how their application responds to certain sensor data, it was important to allow the user to record the data as they are sending it. One usage for this feature would be to allow the user to playback the dataset as they wish and test how different emulators react under the exact same conditions.

The user interface for this feature is simple; The user presses the record button as shown below to start recording.



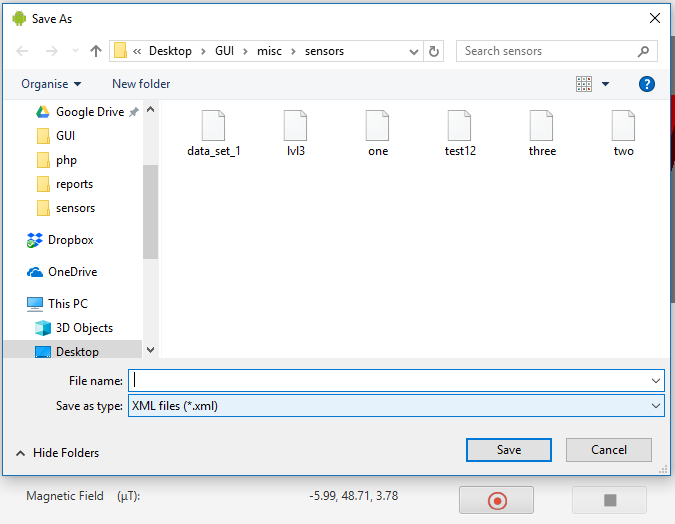
Once the record button has been pressed, the record icon changes to a pause icon and recording begins.



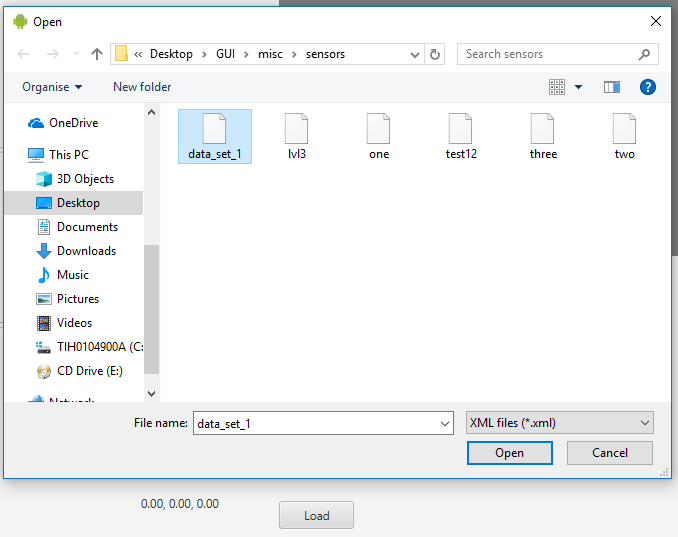
The user can press the pause button to pause the recording, once the recording is paused the stop button is enabled.



The user can pause and play the recording as much as they wish. Upon pressed the stop button, a file chooser dialog is displayed, allowing the user to save the recorded values in an XML with a filename of their choosing.



Once a file has been saved, the user can load a file using the Load button. This will bring up a file chooser dialog allowing the user to select the file.



Upon selecting the file and pressing ‘Open’, the file chooser dialog will close, and controls will appear to control playback of the selected data set.



The user can then play the file back. The sliders and 3D model will move automatically, and sensor data sent to the emulator as the data set is played.

The user can choose for the file to start playing from the beginning once it is finished by selecting the loop check box.

The speed of playback can be altered using the playback speed slider. Playback speeds range between x0.1 and x10 the standard speed.

## How it works

### 6.4.1 Sending sensor data

As mentioned previously in this report, the emulator is designed to take commands over Telnet to control various elements. The format of a message to simulate a sensor event is as follows:

sensor set <name> <value>

e.g. sensor set acceleration 2.23517e-07:9.77631:0.812348

In my application, the action listeners of the sliders (the code that executes when a slider is moved) calls the setSensor method in the TelnetServer class passing a string containing the sensor name and value to it. The setSensor method concatenates the string argument onto the string “set sensor ” and sends it to the emulator via the PrintStream. The code in setSensor is below.

public static void setSensor(String command) {

if(out != null)

out.println("sensor set " + command);

}

Upon receiving this command, the emulator will respond as if a real sensor event had occurred.

### 6.4.2 Recording sensor data

The action listeners on the slider have another functionality, that is to add the sensor name with its corresponding value to a HashMap. A HashMap is a data type provided by the Java collections framework. It stores objects as key value pairs. [13] <https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html>

For example, when an action occurs on the light slider changing the value to 1400lux, the value added to the HashMap would be “light” as the key, and 1400 as the value.

When the recording of the sensor values begins, a TimerTask[14] begins appending the contents of the HashMap to an XML file at a period of 20ms. The short period ensures fine sensor movements are recorded and is important for sensitive movements of the orientation sensors.

A sample of a period in the XML file is as follows:

<stage>

<sensor type="light"><value>35692.30769230769</value></sensor>

<sensor type="proximity"><value>25.0</value></sensor>

<sensor type="temperature"><value>-90.38076923076923</value></sensor>

<sensor type="roll"><value>0.0</value></sensor>

<sensor type="humidity"><value>36.92307692307693</value></sensor>

<sensor type="pressure"><value>143.84615384615387</value></sensor>

<sensor type="pitch"><value>-69.0</value></sensor>

<sensor type="yaw"><value>-147.0</value></sensor>

</stage>

### 6.4.3 Playback of Sensor data

A Thread implemented as a private inner class named playbackThread handles the play back of the sensor values. Once the user loads an XML file of sensor values, an instance of this class is created. This class has a single constructor, taking a multidimensional HashMap as follows:

HashMap<Integer, HashMap<String, Double>>

This HashMap has been populated with the sensor data from the XML file. The Integer represents the number of the period of loaded values.

Inside this thread, a nested for loop loops through the Integer values from the HashMap. For every Integer, it loops through the keyset of Strings (sensor names in this case). For each Double value (sensor values in this case) the slider corresponding to the sensor is set to the value. The action to set the slider value is wrapped in a Platform.runLater() lambda expression. Platform.runLater() is used for thread safety, to ensure that updates to the GUI are done on the JavaFx application thread and not from other background threads.

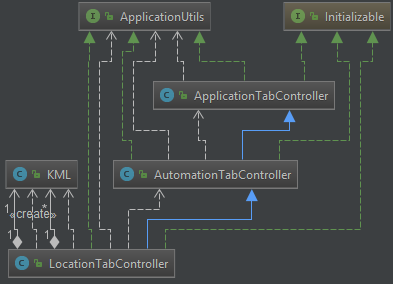
To facilitate pausing the playback of values, a pauseFlag AtomicBoolean is polled on each iteration of the loop. If this flag is set, the thread will wait for the flag to be cleared until continuing.

To facilitate stopping the playback of values, a stopFlag AtomicBoolean is polled on each iteration of the loop. If this flag is set, the program will return from the function.

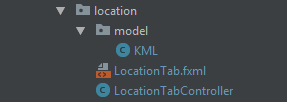
To facilitate looping, upon completion of the playback once the function successfully returns the state of the loop CheckBox is checked, if it is selected the function will begin again.

# Location simulation

The location simulation feature contains a single package. It has the LocationTabController and LocationTab to manage the user interface. The LocationTabController shares some of the functionality of the AutomationTabController (described in the Automation section) so I decided to make it a subclass as to inherit some of the required behaviour. UML diagram of this relationship is shown below, it includes in the relationship with the KML class.



The KML class models the contents of the KML file. These classes are divided as follows.



## Keyhole Markup Language

Keyhole Markup Language (KML) is an XML notation for expressing geographic annotation and visualization within Internet-based, two-dimensional maps and three-dimensional Earth browsers.[15] I made the decision to make use of KML in this project as it is a standard markup language for representing geographical coordinates across the internet.

An example of a KML file is as follows

<kml>

<Document>

<Placemark>

<name>Galway City</name>

<description>Galway City</description>

<Point>

<coordinates>-9.00983,53.27846,0</coordinates>

</Point>

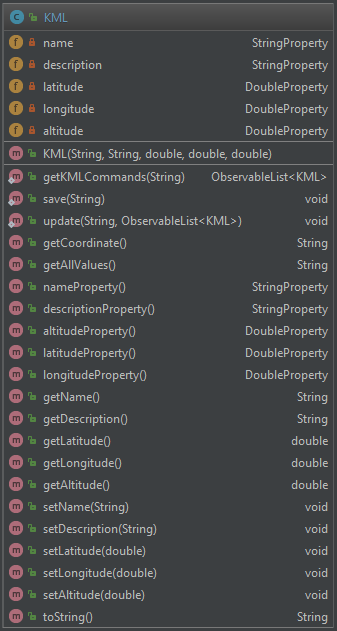
</Placemark>

</Document>

</kml>

KML files can also define other features such as place marks, images, polygons, 3D models, textual descriptions, etc. For my purposes I only need the name, description, latitude, longitude, and altitude.

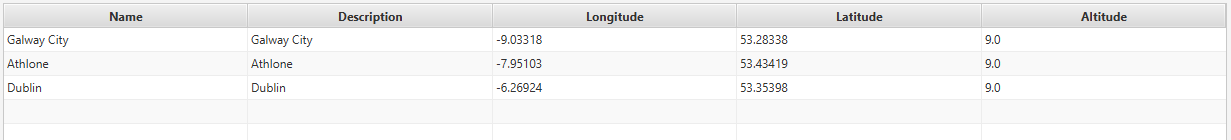
To model the KML file data in my application, I created a JavaFx bean class named KML.

A UML diagram of the class is below.

The members of this class are StringProperties and DoubleProperties instead of Strings and Doubles. Properties are an extension of primitive data types in JavaFX and contain extra functionality allowing them to operate with JavaFX components such as TableViews.

I required these data types as the contents of the KML class are to be contained in a TableView in my application.

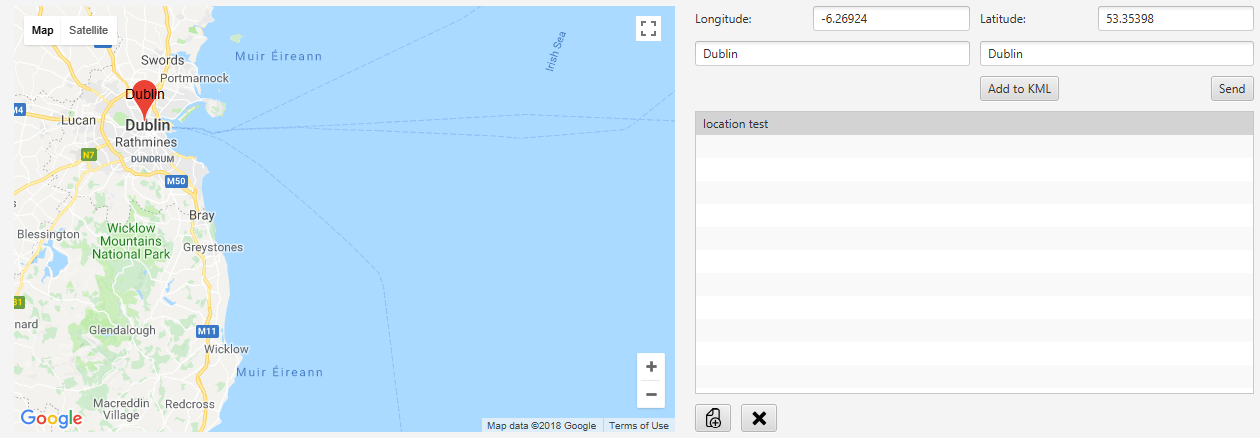
An example of some KML waypoints in the TableView is shown below



## Google Maps Integration

The location feature would not be complete without a method of easily getting the geographical coordinates from a visual map of the Earth’s surface. For this reason, a third party open source Google Map wrapper GMapsFX [16] https://rterp.github.io/GMapsFX/ was integrated into this project.

The Google Map allows the user to pan over the earth’s surface and as they click on the map, the Longitude and Latitude coordinates of the location they click on is placed into their respective text fields. The user can then click the ‘Send’ button to send the location to their emulator or click the ‘Add to KML’ button to add the coordinates, and optionally the name and description to their selected KML file. A screen shot of this layout is below.



# How it works

### 7.3.1 Sending geolocation commands

The main functionality of the AutomationTabController I wished to inherit was the ability to scroll through a list of commands in an editable table (or list) and send commands from the list to the emulator, in this case the commands are coordinates. To send location commands to the emulator, the command looks as follows:

geo fix <longitude> <latitude> <altitude>

e.g. geo fix -6.26924 53.35398 120

In my application, the ways of sending geo commands to the emulator are: Clicking the ‘Send’ button, having the application send the command automatically as it loops through the list of commands, or double clicking on a coordinate in the table. The setLocation method in the TelnetServer class passing a string containing the coordinates and altitude to it. The setLocation method concatenates the string argument onto the string “geo fix ” and sends it to the emulator via the PrintStream. The code in setLocation is below

public static void setLocation(String command) {

if(out != null)

out.println("geo fix " + command);

}

Upon receiving this command, the emulator will respond as if a change of location was detected from the GPS.

### 7.3.2 Adding to the KML file

The KML files are edited through a utility class responsible for editing, saving and opening XML and files named XMLUtil. As the Add to KML button is pressed, a new instance of the KML class is created using the only constructor, passing the Name, Description, Latitude, Longitude and Altitude as parameters. The table is updated with the new KML information and is added to the selected KML file via the XMLUtil.

### 7.3.3 Reading from the KML file

Upon selecting a KML file from the list of files, the table is updated with the contents of the file. This done by passing the name of the file to a method in XMLUtil, openBatchCommands. This method returns an ObservableList[ref] of KML objects. And ObservableList is similar to an ArrayList[ref] from the Java Collections framework but, like the properties has additional functionality allowing it to operate with JavaFx elements.

The table is then populated with all KML commands from this list. The user can edit the table as they wish, deleting, adding, and moving commands up and down the list.

To send the location commands to the emulator, the user can double click on a table item or the play button as shown below.



The application will send the location to emulator at intervals of 3 seconds. The user can pause or stop this playback at any time. The combo box allows the user to choose between running all commands, running from the selected command, and running the selected command.

The code in this feature works similarly to the playback of sensor values and automation commands, which is covered in the next section.

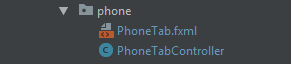
A background thread pulls an ObservableList of KML commands from the table and iterates through them, sending each command to the emulator and panning the Google Map to the coordinates. Upon each iteration of the loop, a pauseFlag is polled. If it is set, the thread pauses until it is cleared. If the stop button is pressed, the function returns.

# Telephony, network, and power events

Developers of Android applications can choose to ‘subscribe’ their application to listen to certain broadcasts[ref] by the system. They might have a need for their application to respond to an event such as an incoming call or SMS message. To a change of network state or act differently under different levels of network coverage. They may require their application to only perform certain processes if the battery level is above a specified percentage or if the device is charging.

For this reason, I decided to include the simulation of telephony, network, and power events.

The code for this feature is contained in a single package as shown below.



## 8.1 Telephony

On the left on the pane, there is a text field for a phone number to send the simulated call or text message from. Underneath this, there is a text area for a text message the user wishes to send to the emulator.

To send location phone calls to the emulator, the command looks as follows:

gsm call <number>

To place the call on hold, the command looks as follows:

gsm hold <number>

To place the call off hold, the command looks as follows:

gsm accept <number>

To end the call, the command looks as follows:

gsm cancel <number>

To send SMS messages to the emulator, the command looks as follows:

sms send <number> <message>

In my application, the ways of sending telephony commands to the emulator are simply by entering the phone number in the number field. Any of the 3 buttons below can be used to manage the phone call (make call, hold/un-hold, end call). The makeCall, holdCall, unHoldCall, endCall methods in the TelnetServer class handle all functionality relating to making phone calls. The methods concatenate the string argument onto the existing string and send it to the emulator via the PrintStream. The code in the methods is below

public static void makeCall(String command) {

if(out != null)

out.println("gsm call " + command);

}

public static void holdCall(String command) {

if(out != null)

out.println("gsm hold " + command);

}

public static void unHoldCall(String command) {

if(out != null)

out.println("gsm accept " + command);

}

public static void endCall(String command) {

if(out != null)

out.println("gsm cancel " + command);

}

To send an SMS message, a number must be entered in the number field and a text message in the message area. The send SMS button is pressed to send the message to the emulator. The sendSMS method in the TelnetServer class provides this functionality. The method concatenates the string argument onto the existing string and sends it to the emulator via the PrintStream. The code in the method is below

public static void sendSMS(String command) {

if(out != null)

out.println("sms send " + command);

}

## 8.2 Network

In the centre of the pane, 4 combo boxes allow the user to simulate network events. The events simulated include: Change of network type, change of signal strength, change of voice status, and change of data status.

To change the network speed of the emulator, the command looks as follows:

network speed <speed>

To change the signal strength of the emulator, the command looks as follows:

gsm signal-profile <number>

To change the voice status of the emulator, the command looks as follows:

gsm voice <state>

To change the data status of the emulator, the command looks as follows:

gsm data <state>

The networkSpeed, gsmSignal, voiceStatus, dataStatus methods in the TelnetServer class handle all functionality relating to network state. The methods concatenate the string argument onto the existing string and send it to the emulator via the PrintStream. The code in the methods is below.

public static void networkSpeed(String command) {

if(out != null)

out.println("network speed " + command);

}

public static void gsmSignal(String command) {

if(out != null)

out.println("gsm signal-profile " + command);

}

public static void voiceStatus(String command) {

if(out != null)

out.println("gsm voice " + command);

}

public static void dataStatus(String command) {

if(out != null)

out.println("gsm data " + command);

}

## 8.3 Power

On the right of the pane, 1 slider and 3 combo boxes allow the user to simulate power events. The events simulated include: Change of battery level, change of battery health, change of battery status, and change of charging status.

To change the battery of the emulator, the command looks as follows:

power capacity <percent>

To change the battery health of the emulator, the command looks as follows:

power health <state>

To change the battery status of the emulator, the command looks as follows:

power status <state>

To change the charging status of the emulator, the command looks as follows:

power present <boolean>

The powerCapacity, batteryHealth, batteryStatus, setCharging methods in the TelnetServer class handle all functionality relating to power state. The methods concatenate the string argument onto the existing string and send it to the emulator via the PrintStream. The code in the methods is below.

public static void powerCapacity(String command) {

if(out != null)

out.println("power capacity " + command);

}

public static void batteryHealth(String command) {

if(out != null)

out.println("power health " + command);

}

public static void batteryStatus(String command) {

if(out != null)

out.println("power status " + command);

}

public static void setCharging(String command) {

if(out != null)

out.println("power ac " + command);

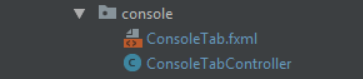
}

# Console

The console is a simple but power feature. This allows the user access to the Android kernel, along with some features of the ADB tool. This feature makes use of the ADB tool and gives access to the connected device or emulator

The console pane consists of a text field to enter the command, a button to send the command to the device and a text area to display the output. A help link is provided to guide the user to the ADB commands reference on the Android developer website.

The code for this feature is contained in a single package as shown below:



## 9.1 How it works

The user enters a command into the text field, the command is executed by pressing ‘Enter’ or clicking the enter button. An instance of a Task[ref] is then created. This task is responsible for calling the ADB tool and passing the command as a parameter to it. It then creates a BufferedReader[ref] to read the output from the ADB tool. As the ADB tool returns output from the Android device, the text is appended onto the text area. The user can cancel any process by pressing CTRL+C, this will cancel the task.

An example of running the ‘top’[ref] command is shown below

adb.exe -s 1a8e261 shell top

The ‘-s’ flag is required to tell ADB which connected device to send the command to, this is only required in the case of multiple devices connected to ADB.

The string 1a8e261 is the name of the device the command is to be sent to.

A sample output from this command is below:

User 11%, System 25%, IOW 0%, IRQ 0%

User 207 + Nice 0 + Sys 470 + Idle 1172 + IOW 0 + IRQ 0 + SIRQ 3 = 1852

PID PR CPU% S #THR VSS RSS PCY UID Name

619 3 3% S 42 42860K 600K fg shell /sbin/adbd

32191 5 1% S 1 6304K 2892K fg shell top

14752 1 1% S 1 6304K 2888K fg shell top

7812 0 1% S 1 6312K 2896K fg shell top

14617 1 1% S 1 6304K 2892K fg shell top

4131 4 1% S 1 6312K 2896K fg shell top

1811 0 1% S 1 6304K 2888K fg shell top

13135 0 1% S 1 6328K 2912K fg shell top

23716 5 1% S 1 6304K 2888K fg shell top

14519 1 1% S 1 6304K 2888K fg shell top

15268 5 1% S 1 6308K 2912K fg shell top

3146 5 1% S 1 6312K 2896K fg shell top

18873 4 1% S 1 6304K 2880K fg shell top

# Automation

Digging deeper into the power of the ADB tool, this feature provides an interface for the user to create a batch of commands they wish to automate on their device or emulator. Some of the automated actions by this feature include:

- Text input

- Button presses

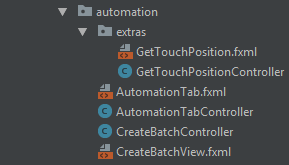
- Touch gestures (tap and swipe)

- Opening, closing, and uninstalling applications

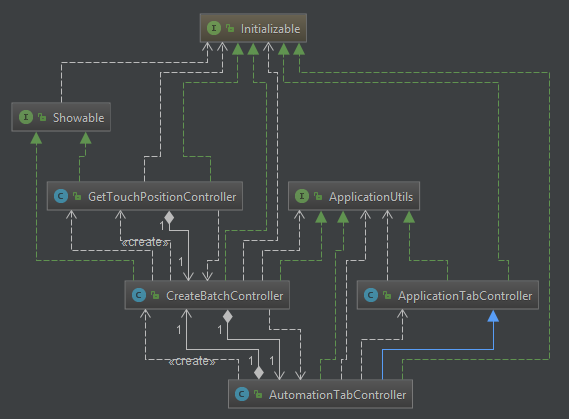
- Copying files to and from the device

- Sending intents to start activities, services and send broadcasts

This feature consists of a pane and 2 modal dialogs and is packaged as follows



A UML diagram describing the relationship is shown below.



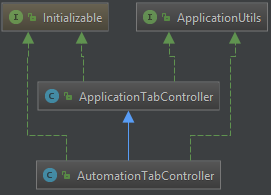
The Automation pane consists of 3 list views. The leftmost list view displays a list of all automation files. These are XML files containing a list of commands.

The top right list view displays all commands in the selected file.

The bottom right list view displays the output from each command as they are run.

As mentioned previously in this report, The LocationTabController shares some of the functionality of the AutomationTabController as the AutomationTabController is a super class of the LocationTabController.

The AutomationTabController is a sub class of the ApplicationTabController, explained in the next section. This was done to inherit the ability to launch an instance of the LogCatTabController (explained in a later section). A UML diagram of this relationship is shown below.



Controls along the top allow the user to play through the list of commands. The same combo box as the LocationTabController allows the user to play all commands, play from selected or just play selected. A check box can be selected to stop playback if a command fails for whatever reason.

Controls along the bottom allow the user to create a new automation file, edit the selected file and delete the selected file.

## 10.1 Creating automation file

The CreateBatchController is responsible for the creation of the batch file.

# Applications

Since applications form such a key part of the Android operating system, it was decided to provide a feature whereby the user could get an insight into all applications installed on their device and carry out important actions relating to the development and testing of applications.

The Application feature provides a list on the left from which the user can select an application and have its details appear in the two table views on the right. A search bar is provided on the top to enable the user to search for their required application.

Along the top row, buttons provide the user with actions to carry out on the selected application including opening, closing, uninstalling, and copying over the APK file to their computer. The LogCat monitor described in the next section can be displayed in another window detailing logs from the selected application.

The top most table view contains columns containing the name of the APK file, path to the APK file, version information, application ID (process ID) and location of the application’s data directory.

The bottom table view contains a list of all intents[ref] that the device is capable of receiving as specified in the manifest file[ref].

## 11.1 What is an Intent?

An intent is an abstract description of an operation to be performed. It can be used to start an activity, send a broadcast to any subscribing broadcast receivers or to communicate with a background service.

Intents are made up of different attributes.

Action: The action to performed by the intent. E.g.

ACTION\_VIEW

ACTION\_SEARCH

ACTION\_SEND

ACTION\_EDIT

Category: Additional information about the action to execute. E.g.

CATEGORY\_LAUNCHER

CATEGORY\_DEFAULT

CATEGORY\_ALTERNATIVE

Component: Specifies an explicit name of a component class to use for the intent.

The four different types of component are Activities, Services, Broadcasts, Content providers.

Mime Type: Specifies the type of data passed with the intent, this is only used in the case of an intent that takes data.

Data: The data to operate on. Examples of data are contact, file, http.

Many applications define their own data types specific to their needs.

There are two main types of intents: Those that take data and those who don’t. When data is sent with an intent, it is done with the intention of the receiving object acting upon it in some way.

An example of this is sending the data ‘content://contacts/people/1’ with an intent. Based on the action of this intent, the data can be acted upon in a certain way e.g. Viewing the associated contact, editing the contact, or dialling the contact.

## 11.2 Activities

An activity is a single, focused thing that the user can do. Almost all activities interact with the user. An activity is the entry point for interacting with the user. It represents a single screen with a user interface.[ref]

Each activity is independent of the others. Activities can launch other activities by way of intents. This can be done within applications or across different applications. For example, the gallery application can start the email application by the user selecting to send a picture in an email. In this case the data sent in the intent is the photo and the mime type is image/jpg or image/png for example.

## 11.2 Services

A service is a way for applications to perform long running processes independent of user interaction. A service can run in the background while the user is not interacting with the application.[ref]

Applications can start services using intents. An intent used to start a service may or may not take data.

## 11.3 Broadcast receivers

A broadcast receiver is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements. [ref]

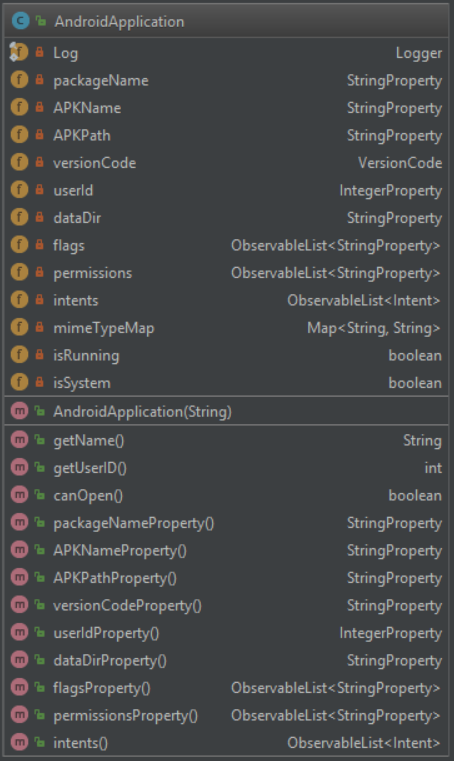
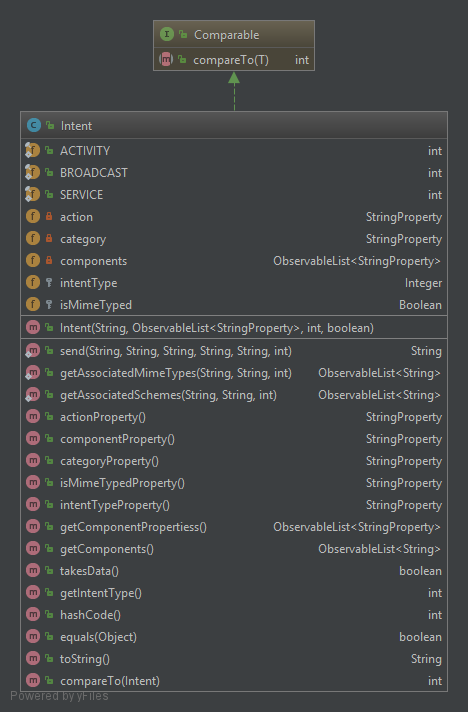
Application developers normally subscribe to system wide announcements such as change of network state, power state or telephony events as describe in section 8.

Developers can create broadcast receivers to subscribe to broadcasts sent by their own services. These broadcasts normally indicate a service completing or having completed a certain task that the activity can then respond to.

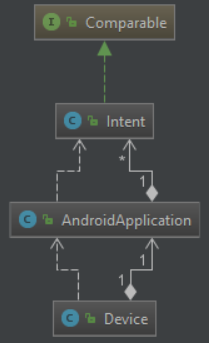
## 11.4 How it works

This feature relies heavily on two classes to model the information relating to Android applications and intents. These classes are the AndroidApplication class and the Intent class.

UML class diagrams of the AndroidApplication and Intent class are shown below.

Since the contents of these classes are to be displayed in table views, the fields are mostly JavaFx properties. A class following the singleton pattern [ref] named Device is widely used in this application to model the information of the connected Android device and provide functionality to it. This Device class holds a reference to an AndroidApplication, this AndroidApplication is instantiated when the user selects an AndroidApplication from the list view on the left. Each AndroidApplication holds an ObservableList of Intents. A UML diagram outlining the relationship is shown below.



Upon selecting an Android application from the list view on the left, a new Android application is instantiated. This involves parsing output dumpsys [ref] tool to get the necessary data relating to the application, it is a relatively long running task (up to 5 seconds) and is performed by a background thread. Upon instantiation, the data relating to the Android application and list of its intents are display in their respective table views.

The dumpsys tool is a tool that runs on Android devices and provides information about system services. For this application, dumpsys is invoked the with the following arguments:

adb.exe shell dumpsys package <package name>

To make for a tidier output and faster parsing time, the output is frequently piped into grep[ref] to show only the required data.

### 11.4.1 Parsing Intents

The dumpsys tool gives output pertaining to all intents as declared in the manifest file. These intents can be used to start activities, services or send broadcasts. In this application, the intents used to start activities are gathered first, followed by the intents used to start services and lastly the intents used to send broadcasts.

Regex[ref] is used widely when parsing the dumpsys output for intents.

In the case of intents that take data, further information is gathered into the type of data the intent takes, this is specified in the Mime type of the intent.

### 11.4.2 Sending Intents

A key aspect of this feature is providing the user with the ability to send intents to their device. This is useful in the case of deep linking in an application [ref].

The am tool, or activity manager[ref] https://developer.android.com/studio/command-line/adb#am is provided as part of the Android kernel and accessible through ADB. This tool is intended to provide developers with a way to resolve correct activity and test deep linking.

The developer can call this tool, passing the necessary arguments to send an intent do their device.

An example of invoking this tool with an intent to start the Chrome browser and link to the google.com website is shown below

adb.exe shell am start

-a android.intent.action.VIEW

-c android.intent.category.DEFAULT

-n com.android.chrome/com.google.android.apps.chrome.IntentDispatcher

-t text/html

-d http://google.com

The -a flag specifies the action to be completed, in this case it is ACTION\_VIEW

The -c flag specifies the category of the intent, in this case it is CATEGORY\_DEFAULT

The -n flag specifies the component class for the intent, in this case it is

com.android.chrome/com.google.android.apps.chrome.IntentDispatcher

The -t flag specifies the type of data provided with the intent, in this case it is text/html

The -d flag specifies the data passed along with the intent, in this case it is the URL http://google.com

This application provides 5 text fields, from which the user can build an intent and a combo box to select what to do with the intent (start activity, service or send broadcast).

The user can choose the manually enter the attributes of the intent, or by clicking on an intent in the table, the application will automatically fill the fields. In the case where there are multiple components capable of receiving the intent, a combo box beside the component field can be used to select the required component.

In the case of intents that take data, combo boxes are provided beside the data and mime type fields. These can be used to select the data and mime type to send with the intent.

When the user wishes to send the intent, they do so by simply clicking the send button. This will invoke the am tool and pass it the necessary flags taken from the text fields.

The output of the am tool is displayed in the text field below. This will indicate the success, or failure of sending the intent. In the case of a failure, it will print an error message and code describing the type of error received.

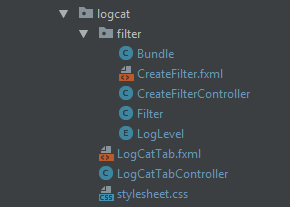
# LogCat

Logcat is a command-line tool that dumps a log of system messages, including stack traces when the device throws an error and messages that you have written from your app with the Log class.[ref] <https://developer.android.com/studio/command-line/logcat>

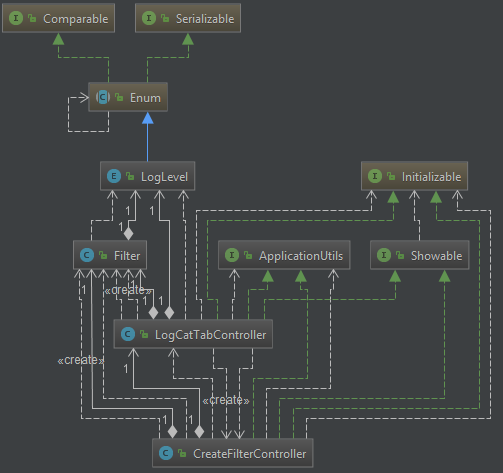
This feature is intended to provide the user with an easy way to view, search, save and create filters to see the logs coming from their device.

To create and save filters, a java bean named Filter was created to model the filter contents, a modal dialog was also created to allow the user to create and edit filters. Filters are saved in an XML file.

The layout of the logcat package is shown below.



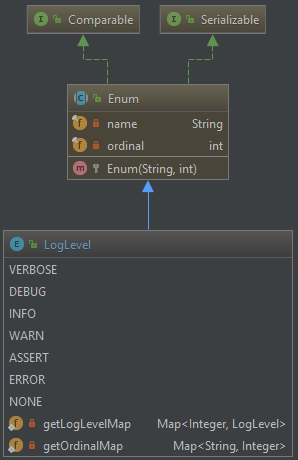
The UML diagram below explains the relationship between these classes.



The LogLevel is an Enum representing the logging levels provided by the logcat utility:

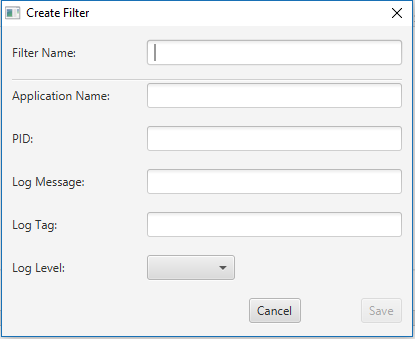
Verbose, Debug, Info, Warn, Assert and Error.

A UML diagram describes the LogLevel Enum below.

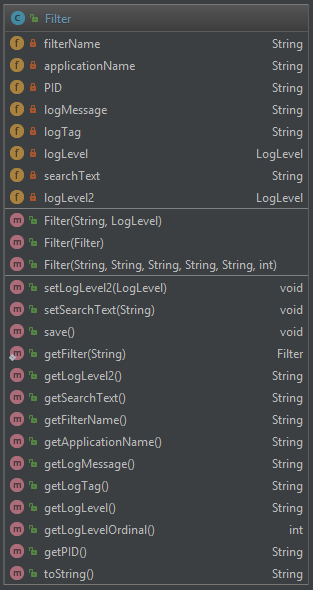


The Filter class contains fields representing the name the user wishes to call the filter, as well as items in the log that are to be filtered for.

When a filter is applied, the contents of the filter will be included, as well as the contents of the search field and the LogLevel combo box at the top of the pane.

A screen shot of the CreateFilterController is below.

Below is a UML diagram of the Filter class. The searchText and logLevel2 fields contain the search text and log level from the GUI. The other fields contain the items stored in the filter.



## 12.1 How it works

### 12.1.1 Reading from the logcat utility

The logcat utility can be invoked with the command

adb.exe logcat -v threadtime

The threadtime flag defines the output to print along with the log messages. In this case, the timestamp of the log was desired to provide for a more user-friendly experience.

A BufferedReader is created to read from the logcat utility as the logs are received. These logs are filtered using the filter mentioned previously and if the content of the log matches the user’s criteria it is added to the list view and displayed on the GUI.

CSS [ref] is applied dynamically as logs are received before they are displayed on the GUI. The CSS applies colour to the log according to its level.

The CSS is applied as follows.

Verbose: green

Debug: blue

Info: brown

Warn: orange

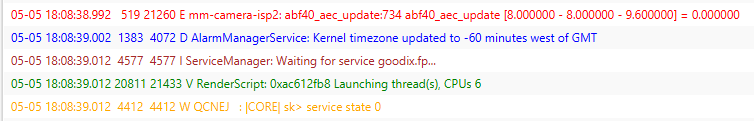
Assert: black

Error: red

The CSS for these styles is contained in the stylesheet.css file. This file is attached only to the LogCatTabController and not to the rest of the application.

The log level of a log is defined by a single letter in the fifth column of the log.

An example of the styled logcat output is below.



### 12.1.2 Creating a filter

As mentioned previously, the Filter class (essentially a bean) holds all data related to the filter. The contents of the saved filter are stored in XML. The XMLUtil class provides methods for creating and opening XML filter files.

The user can press the ‘Plus’ button to display the CreateFilterController as a modal dialog. This pane has fields and a combo box to add the filter details.

When the user saves the filter, an instance of the Filter class is created and passed as an argument to a method in the XMLUtil class. The method creates and saves an XML file with the name provided by the user in the ‘Filter name’ field.

To edit a filter, the user can select a filter from the filter combo box and press the Edit button. This will bring up the CreateFilterController. The text fields will be populated with the contents of the filter criteria.

### 12.1.3 Filtering logs

The LogCatTabController holds an instance of the Filter class. When a filter is selected, this filter class holds the contents of the selected filter. It also holds the contents of the search field and the LogLevel combo box.

As logs are read by the BufferedReader, they are checked against the contents of the Filter class instance, if they fit the criteria they are added to the list view.

To parse the log, it is divided up using the split[ref] method of the String class and the individual columns of the log are checked against the contents of the Filter instance.

The contents of the method responsible for this is shown below.

private boolean matchesFilter(Filter filter, String s) {

if (s.startsWith("-") || s.isEmpty())

return false;

s = s.replaceAll(" {2,3}", " ");

String[] split = s.split(" ");

String pid = split[2];

String level = split[4];

String tag = split[5];

String message = s.substring(s.indexOf(tag));

message = message.substring(message.indexOf(":")+1);

String logLevel1 = filter.getLogLevel().substring(0, 1).trim();

String logLevel2 = filter.getLogLevel2().substring(0, 1).trim();

if (!logLevel1.equals("N"))

if (!level.equals(logLevel1))

return false;

if (!logLevel2.equals("N"))

if (!level.equals(logLevel2))

return false;

return s.contains(filter.getSearchText()) &&

s.contains(filter.getApplicationName()) &&

(filter.getPID().isEmpty() || pid.contains(filter.getPID())) &&

(filter.getLogTag().isEmpty() || tag.contains(filter.getLogTag())) &&

(filter.getLogMessage().isEmpty() || message.contains(filter.getLogMessage()));

}

### 12.1.4 Saving logs

To provide the user with the ability to save logs, a save button is provided along the top row of the LogCatTabController pane.

Once invoked, a Task is created and saves the contents of the logcat list view to a file. The file name is created by concatenating the name of the connected device with the current timestamp.

The file is saved in the logcat directory of the application directory. To access this directory, along with the other directories used by this application the user can click the File menu on the top-left of the GUI and select ‘Open Application Directory’ from the list.

The logs are stored in the ‘logcat’ directory as .log files. These can be opened by the user in any text editor they please for analysis at a later time.

# Monitor

The monitor feature was the final feature to be implemented into this application. It is a feature that makes use of tools found across most Linux kernels. These tools include ps, top, dumpsys (as was mentioned previously), and the proc directory (short for process). These are explained in more detail in the next sections.

3 graphs in this pane provide graphical output of the CPU, Memory and Network usage of the system and selected Android application. Labels to the right of each graph provide further insight into each section.

The MonitorTabController is the controller of this pane and is a subclass of the ApplicationTabController. This decision was made in order to inherit some of the behaviour of the ApplicationTabController, including a nested split pane to hold a list of installed applications and buttons to provide input to these applications.

A class runs as a thread in the background with many Tasks running to update the graph and labels every second. This class is named MonitorService and implements the Runnable interface. This class updates the fields of 3 JavaFx bean classes with the responsibility of modelling the information relating to the CPU, Memory and Network usage respectively.

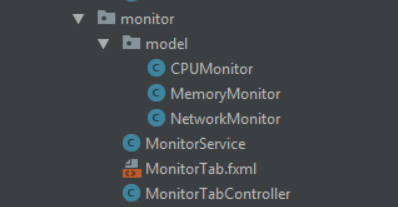
The 3 classes modelling information are named CPUMonitor, MemoryMonitor and NetworkMonitor. These classes long with the Device class as mentioned previously and MonitorService follow the singleton design pattern as to prevent instantiation of more than one object of each class. Instead, the two classes that rely on the CPUMonitor, MemoryMonitor and NetworkMonitor (MonitorTabController and MonitorService) hold the same instance of each class. This instance is retrieved through a static getter method named getInstance in each case.

The MonitorTabController uses a JavaFx feature called binding[ref] to bind its labels to properties in the model classes. This is a powerful feature and reduces the overhead of running a background thread in a loop to update the labels on a constant basis.

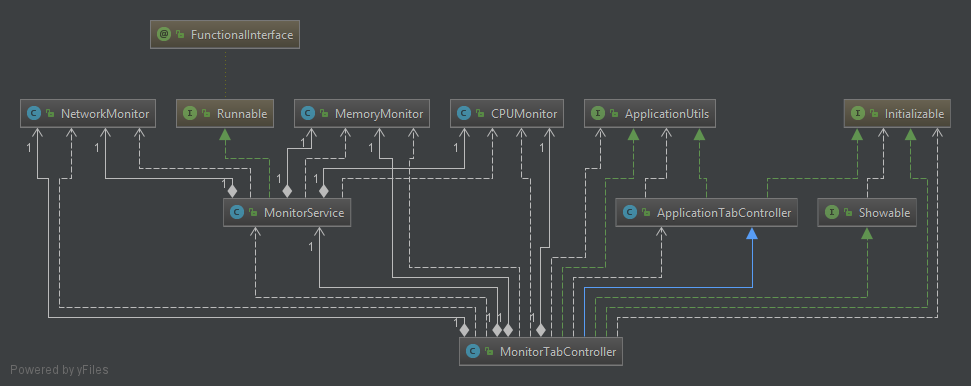
The Y axis of the 3 graphs gets its data from properties in each respective class and is updated every second.

When the monitor tab is not selected, the Tasks in the MonitorService are paused, this is done to reduce CPU usage of the computer and connected device.

The classes involved in this feature are packaged as follows.



A UML diagram describing the relationship of these classes is shown below.



## 13.1 Top (table of processes)

The top program provides a dynamic real-time view of a running system. It can display system summary information as well as a list of tasks currently being managed by the Linux kernel. It produces an ordered list of running processes selected by user-specified criteria, and updates it periodically. [ref] <https://linux.die.net/man/1/top>

In this application, top is used to gather information about the CPU usage of the system and selected application. To get system CPU usage, top is called with the following parameters

adb.exe -s <device name> shell top | grep System

To get CPU usage of the selected application, top is called with the following parameters

adb.exe -s <device name> shell top | grep <application name>

For System, the output looks as follows.

User 0%, System 0%, IOW 0%, IRQ 0%

User 0%, System 1%, IOW 0%, IRQ 0%

User 0%, System 0%, IOW 0%, IRQ 0%

The output is received every second while the task is running. The output is parsed to get the percentage usage beside the System word.

The top utility can also be called with the -t flag. This is used to get information about threads running on the device. Every second, it outputs a list of running threads along with details about each one. The output looks as follows.

User 17%, System 8%, IOW 0%, IRQ 0%

User 357 + Nice 1 + Sys 169 + Idle 1502 + IOW 1 + IRQ 0 + SIRQ 0 = 2030

PID TID PR CPU% S VSS RSS PCY UID Thread Proc

3776 4086 0 8% R 1555040K 135212K fg system RenderThread com.android.systemui

3776 3776 1 6% S 1555040K 135212K fg system ndroid.systemui com.android.systemui

429 429 1 1% S 164832K 10036K fg system surfaceflinger /system/bin/surfaceflinger

26898 26914 0 1% S 1201324K 106720K bg u0\_a178 HeapTrimmerDaem com.snapchat.andro

As the output is received every second, the number of lines in the output is counted (minus the header and whitespace) this gives the number of running threads on the system.

## 13.2 PS (process status)

PS is a command line utility that displays the current running processes [ref] <https://linux.die.net/man/1/ps>

In this application, ps is used to get the number of running processes. The output is piped into wc[ref] with the -l flag to count the number of lines (minus the header and whitespace).

adb.exe -s <device name> shell ps | wc -l

An output of ps without piping it into wc is below.

USER PID PPID VSIZE RSS WCHAN PC NAME

root 7 2 0 0 ffffffff 00000000 S rcu\_preempt

root 8 2 0 0 ffffffff 00000000 S rcu\_bh

root 9 2 0 0 ffffffff 00000000 S rcu\_sched

root 10 2 0 0 ffffffff 00000000 S migration/0

root 11 2 0 0 ffffffff 00000000 S migration/1

root 12 2 0 0 ffffffff 00000000 S ksoftirqd/1

## 13.3 /proc

The proc directory in Linux is a virtual file system. It doesn't contain real files but runtime system information (e.g. system memory, devices mounted, hardware configuration, etc). For this reason, it can be regarded as a control and information centre for the kernel. [ref] <https://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html>

In this application, the contents of the proc directory are used to get an insight into all facets of the monitor feature (CPU, Memory and Network).

The /proc file system is used to provide insight into the following information:

CPU

CPU manufacturer: adb.exe -s <device name> shell cat /proc/cpuinfo | grep Hardware

Number of cores: adb.exe -s <device name> shell cat /proc/cpuinfo | grep ‘CPU Architecture’

CPU uptime: adb.exe -s <device name> shell cat /proc/uptime

Memory

Total memory: adb.exe -s <device name> shell cat /proc/meminfo | grep MemTotal

Percentage memory used:

adb.exe -s <device name> shell cat /proc/meminfo | grep -E 'MemFree:|Buffers:|Cached:|SwapFree:'

Network

Received, Sent KBps (system):

adb.exe -s <device name> shell cat /proc/net/dev | grep wlan0

Received, Sent KBps (application):

adb.exe -s <device name> shell cat /proc/net/xt\_qtaguid/stats | grep <application name> | grep wlan0

The output of these commands is parsed to get the relevant information.

## 13.4 How it works

### 13.4.1 Monitor

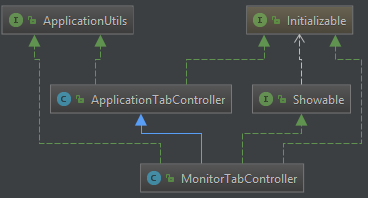
The MonitorService is responsible for gathering all the information in this feature. It is started when a device is connected to this application. When the user clicks on an application in the list view it starts again, in order to start monitoring the selected application. The service is paused while the monitor tab isn’t selected as it would be wasteful of CPU resources to keep it running.

Inside the run method of the MonitorService, several tasks are started, each responsible for gathering information relating to CPU, Memory or Network usage. Each of these tasks are called passing the name of the command to execute as a parameter in the constructor. The tasks all run in a while loop. At the start of the iteration of every loop, a check is made if a request was made to cancel the task, if so the loop exits. A check is also made to see if the task is paused, if so it waits until it can resume.

The tasks update fields of their respective Monitor class. This update is performed in a Platform.runLater() lamba expression. This is done because the properties are bound to labels and axes of the charts and thus, any modification of these variables must be done on the JavaFX application thread.

### 13.4.2 Controller

As mentioned previously, the MonitorTabController is a subclass of the ApplicationTabController. This relationship is described in the below diagram



The Showable interface is implemented by this class to define the ability to be launched in a new window. This behaviour was given to the MonitorTabController to allow the user to monitor the behaviour of their device while under automated testing.

ApplicationUtils is an interface implemented by most controllers in this application. It defines a method to initialize buttons. This is done to set buttons enabled/disabled or hidden initially and to provide an optional graphic and tool tip text to it. The ApplicationUtils interface makes use of ‘default’ interface methods. This is a new feature to Java 8 and allows developers to provide methods in interfaces with implementation. Unlike the standard abstract methods normally defines by interfaces, the implementing class does not need to provide an implementation for these methods. They can be used to provide useful utility methods to classes.

The 3 default methods defined in ApplicationUtils are:

void browse(String URL)

This method will display the default web browser and browse to the specified URL.

void setImage(String URL, String text, Button button)

This method will apply an icon and tool tip text to the specified button.

ObservableList<String> filter(String searchText, ObservableList<String> list)

This method returns an ObservableList<String> of items that contain the specified search text.

The charts in MonitorTabController are updated every second. The Y axis of the charts is linked to properties in each of the respective monitors. This property in each case is an IntegerProperty.

In the case of the CPU chart, it represents the percentage CPU utilization.

In the case of the Memory chart, it represents percentage Memory utilization.

In the case of the Network chart, it represents the Sent and Received KiloBytes per second. To calculate the KiloBytes per second in the MonitorService the total Sent/Received KiloBytes is retrieved every second and the KiloBytes sent/received in the previous second is subtracted from it.

As mentioned previously, the labels are found to properties in the monitors. This removes a lot of overhead which otherwise be needed to constantly update the labels in a background thread. All labels are bound in a method called bindLabels(). An example of binding a label is shown below.

CPUSpeedLabel.textProperty().bind(cpuMonitor.currentFrequencyProperty().asString("%.1f").concat(" GHz"));

In this case the CPUSpeedLabel label is bound to the currentFrequencyProperty in the CPUMonitor class. The currentFrequencyProperty is a DoubleProperty and therefore, the asString() method must be called to convert it to a string for binding with the textProperty. The format specifier “%.1f” ensures the value will be printed to only one decimal place. The concat() method is used to concatenate “ GHZ” onto the end of the label.

# Conclusion

Since the very beginning, this project set out to be a demonstration of my abilities and what I have learned from this course. It is my opinion that the finished product provides an honest realisation of these skills.

From building this application, I have gained experience with innovation. The technical skills I have gained from this project are immense and I now have a new-found interest in the Java programming language and object orientated programming, an interest and skill that I hope to nurture and improve upon as my career as a software engineer develops. The insight this project has given me into the Android operating system is profound and I find it to be a very impressive piece of technology. No surprise most of us use it to enhance our lives on a daily basis.

The project increased in scope and surpassed initial objectives as I gained skills and Java, GUI development and learned of the power of the ADB tool. I hope to continue the development of this application, adding additional features as I see fit and releasing it to the open source community under the GNU license.

It was a challenging and interesting project and has though this author a lot about GUI development and the Java programming language as well as a deeper insight into the workings of the Android operating system and a new-found appreciation for the technology we all use in our day to day lives.

# References

[1] javafx wiki

[2] <https://www.techotopia.com/index.php/An_Overview_of_the_Android_Architecture>

[3] By Smieh - Anatomy Physiology of an Android, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=20067152>

# Appendices

## 15.1 Poster



## 15.2 Source code

Source code can be found at the following link: <https://github.com/ronanwatkins/FYP-GUI>

## 15.3 Documentation

The documentation can be found at the following link: <https://ronanwatkins.github.io>