Wearable App for Students

Aidan Grabe

Final Year Project
Computer Science

Academic Mentor: Dr. Sabin Tabirca
Department of Computer Science
University College Cork
Co Cork
Ireland
April 14, 2015

Declaration of Originality

In signing this declaration, you are confirming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way towards an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Name: Aidan Grabe

Signed: _____

Date: 08/04/15

Abstract

Android Wear is a version of the Android operating system, currently tailored for smartwatches, announced on March 18, 2014. This project explores the underlying technology powering Android Wear and demonstrates some of it's capabilities with a smartwatch and mobile application targeting students and their day-to-day activities.

The project consists of three main components. A research project demonstrating the low level APIs of Android Wear and displaying the raw data acquired from the different sensors. The second, is a student application using some of the sensors and information acquired by the research project and using it in a practical way to create a usable application for college students. The third component is a custom watch face designed specifically for Android Wear that demonstrates how the Android Wear watch face APIs work.

Contents

1 Introduction **6**
- 1.1 Aims and Objectives . . . 7
- 1.2 Document Structure . . . 8
- 1.3 Personal Contributions . . . 9

2 Background **10**
- 2.1 Introduction to Android Wear . . . 10
- 2.2 Current Technology . . . 11
  - 2.2.1 Apple Watch . . . 11
  - 2.2.2 Pebble Time . . . 11
- 2.3 Communication . . . 12
- 2.4 Development Environment . . . 13

3 Analysis & Design **15**
- 3.1 Overview . . . 15
- 3.2 Application Architecture . . . 16
  - 3.2.1 Module Structure . . . 18
  - 3.2.2 Package Overview . . . 18
  - 3.2.3 API Overview . . . 20
- 3.3 Problem Analysis . . . 20
  - 3.3.1 Communication . . . 20
  - 3.3.2 Synchronizing Data . . . 22
  - 3.3.3 Availability . . . 22
- 3.4 User Interface . . . 23
  - 3.4.1 UI Compatibility . . . 23

4 Implementation **25**
- 4.1 Language Choice . . . 25

4.2 Development Tools . . . 26
4.2.1 IDE . . . 26
4.2.2 Version Control . . . 26
4.3 Android Wear Communication . . . 26
4.3.1 Pre-requisites . . . 27
4.3.2 Communication Types . . . 27
4.3.3 Message API . . . 29
4.3.4 Data API . . . 32
4.3.5 WearableListenerService . . . 36
4.3.6 Syncing Assets . . . 38
4.3.7 Object Serialization . . . 40
4.3.8 Communication Threads . . . 43
4.4 Sensors . . . 45
4.5 Gestures . . . 50
4.6 Notifications . . . 52
4.7 Student Application . . . 53
4.7.1 Modules . . . 56
4.7.2 To-Do List . . . 56
4.7.3 Timetable . . . 59
4.7.4 Results . . . 62
4.7.5 News . . . 66
4.8 Campus Map . . . 72
4.9 Games . . . 76
4.9.1 Lights Out . . . 76
4.9.2 Minesweeper . . . 78
4.9.3 Multiplayer Snake . . . 83
4.10 Custom Watchface . . . 84
4.10.1 Overview . . . 84
4.10.2 Watchface API . . . 85
4.10.3 Drawing on the Canvas . . . 88
4.10.4 Calendar Events . . . 90
4.10.5 Calendar WatchFace . . . 91

5 Testing **92**
5.1 Overview . . . 92
5.2 Unit Testing . . . 93
5.3 Performance Testing . . . 94
5.4 Usability Testing . . . 95

6 Conclusion **96**
6.1 Overview 96
6.2 Android Wear Conclusion 97
6.3 Future of Android Wear 98
6.4 Future Work 99

Chapter 1

Introduction

As computer components become smaller, cheaper to manufacture and more powerful, embedded systems are more accessible than ever before. They can be designed to perform specific functions with a subset of the hardware required for a typical personal computer. Thus consuming a fraction of the power, and costing a fraction of the price.

As more and more devices become interconnected with the "Internet of Things", embedded systems are playing an ever more important role in people's lives, keeping them connected and allowing them to control different aspects of their daily activities from one device, such as a smartphone.
Today, you can control the lights in your house, have a live home security stream, control your television or laptop or get alerts about and alter the temperature in your home all from the screen of your smartphone.

Since these systems are getting smaller and cheaper to produce, the trend is moving towards wearable technology. Components such as heart rate monitors, step counters, proximity sensors etc. are all components that can be worn on the body and the information can be fed to the user's smartphone via Bluetooth, WiFi or standard wiring.

With these sensors worn on the body or embedded in the user's clothing, the smartphone now has a constant feed of updating sensor data with which it can make important decisions. For example, if a user is jogging while wearing a heart rate sensor, the smartphone could play a different tone into the headphones based on the level of the heart rate. The user could configure this to encourage them to run faster

when the heart rate is below a certain threshold or encourage them to slow down if it is too high.

These types of wearable sensors can be hugely beneficial to users with medical conditions. For example, a sensor that automatically detects a user's blood sugar levels and alerting them via their smartphone when the levels are too low could help avoid potentially dangerous scenarios.

Wearable sensors would also benefit athletes by providing them with data about their performance and perhaps predict certain injuries before they occur.

Android Wear, as the title may suggest is Google's attempt to create a set of APIs for dealing with these wearable sensors. Currently it is tailored specifically for smart watches, but one could assume that future updates might support a wider range of wearables such as Google Glass or devices similar to Nike's Nike+ range of shoes.

Android Wear runs a very similar version of the Android Operating system to that running on a mobile phone or tablet, but with certain parts stripped away (eg. Camera, WiFi) or tailored for the smaller screen size. This means developers can use most of the regular Android APIs and can even send serialized objects over Bluetooth and have them reconstructed on the other side of the connection.

1.1 Aims and Objectives

The purpose of this project is to outline some of the capabilities and API usage for the Android Wear operating system and demonstrate some of these capabilities in a practical application for college students. The project consists of three main components:

1. Research project demonstrating different sensors and their data along with some of the more important APIs of Android Wear.
2. Mobile/tablet and smartwatch applications for college students which allows users to schedule lectures, track results keep a to-do list etc.
3. A Custom Watchface that displays a user's calendar events along the circumference of the watch face.

1.2 Document Structure

This document contains 6 chapters:

Chapter 2

This chapter gives a brief overview of the Android Wear technology in it's current state and how to start development of an application.

Chapter 3

This chapter presents the architecture and design of the applications. It gives an overview of the solution and describes initial problems faced by development.

Chapter 4

This chapter goes into the details of the implementation of the project. It describes in depth, how the application's different sections work and problems encoutered along the way.

Chapter 5

This chapter gives an overview of the testing process and how testing was approached and undertaken for the project.

Chapter 6

This chapter outlines the final conclusions of the project. The successes and failures and gives a brief outline of the future work that can be undertaken on the project.

1.3 Personal Contributions

Both the Student Application (wearable and handheld) and custom watchface are open source and available on Github. There are also some utility classes within the projects that may be useful to other developers when sending and receiving data over Bluetooth using the Android Wear APIs.

The repository for the custom watchface has seen a bit of activity from other developers and may hopefully provide others with a base template when creating a custom watchface of their own.

Chapter 2

Background

2.1 Introduction to Android Wear

Android Wear is a version of Google's Android operating system tailored for wearables. At the time of writing, Android Wear is only available for smartwatches, but one can assume that it will be extended to other wearable devices in the foreseeable future.
Android Wear was announced on March 18, 2014 and the first publicly available devices were released on June 25, 2014 at Google I/O. These devices were the "LG G Watch" and the "Samsung Gear Live" (not to be confused with the "Samsung Galaxy Gear" which is not an Android Wear device.)

Upon release, Android Wear shipped running "Android 4.4W" (API 20) or "Android KitKat" for wearables and could communicate with mobile devices running "Android 4.3" (API 18) or higher. At the time of writing, the latest Android Wear revision is Android 5.0.2 (API 21), more commonly known as "Android Lollipop".

Android Wear is relatively restricted in what it can do on it's own. It can run applications and services but may need a connection to a mobile device to carry out certain tasks. For example, smartwatches have no internet connection and so will need to use the mobile device as a proxy if web-related content is needed. GPS is only available on select smartwatches such as the "Moto 360" and so the mobile device may again be used as proxy if necessary. It is also recommended to use the mobile device for long running, or computationally intensive tasks so as not to drain the wearable device's battery.

2.2 Current Technology

At the time of writing three main players exist in the smartwatch market.

1. Apple
   Soon to be released Apple Watch which comes in different variants and sizes.
2. Google
   Multiple smartwatches running Android Wear.
3. Pebble
   Pebble Watch and soon to be released Pebble Time smartwatch.

2.2.1 Apple Watch

Apple's first attempt at a smart watch is being released to the general public in the summer of 2015. The watch comes in 3 different styles: Watch Sport, Watch, and Watch Edition. The price for the watch ranges from $349 all the way up to $17,000 for models made of 18-Karat gold.

Apple Watch can communicate exclusively with devices running iOS. Android or Windows users will not be able to use the watch with their devices. The Apple Watch's stand out feature is probably Apple Pay. Apple Pay allows the wearer to pay for goods by simply tapping the watch to a payment device and clicking a button on the watch to complete the transaction.

Apple Watch also implements a unique feature called "Force Touch" which allows the device to detect touches of different pressure. The watch's screen can differentiate between a light tap and a deep press.

Apple Watch applications are run on the handheld device exclusively using the watch's display only as a view. Code is executed on the handheld device and the application sends the content to display back to the watch. These watch applications are called "Extensions" and are bundled into the mobile application.

2.2.2 Pebble Time

The Pebble Time is a soon to be released watch by Pebble that comes in two variants. A regular, plastic model and the Pebble Steel which boasts similar specs and a better

battery.

Pebble's two stand out features are its cross-platform capabilities and it's battery life. The Pebble uses a very different display technology to other devices currently on the market. Android watches and the Apple Watch use OLED technology which allows the display to light up only the pixels it needs to, leaving black pixels off to save battery. The Pebble watches however use e-displays. These displays use "electronic paper" which do not use back-lighting to light the display, instead they are lit by external light sources which make them easier on the eyes when reading and allows them to consume less battery power.

The Pebble watches can be used by Apple, Android and Windows Phone (through third-party apps) users.

2.3 Communication

Android Wear uses Bluetooth LE (Low Energy) for communication between smartwatch and mobile device. Communications are secure and reliable using the official Android Wear API. Manual communications over Bluetooth is still possible, but not recommended[1].

When communicating between mobile and watch, every communication has a URI attached to it. Each URI consists of a base which is specific to Android Wear and the current application running, and a path that allows the developer to differentiate between different communications.

The communications API consists of three different methods:

1. Messages
   Messages use the `MessageApi` and contain small amounts of data. Messages are especially useful for Remote Procedure Calls (RPC) or for implementing a request-response model in an application. Once a message is sent/received, it is destroyed.

2. Data Items
   Data items are used to synchronize data between the wearable and the mobile device. Creating a `DataItem` automatically replicates the data on the other side of the connection allowing for easy sharing of Java objects or data structures.

[1]Google Developers: http://goo.gl/Ke7MAu

`DataItem`s are limited to a payload of 100KB. If a bigger payload is required, an `Asset` can be used.

3. Assets
   An `Asset` object can be attached to a `DataItem` and can contain a payload greater than 100KB. `Asset`s can be any blob data, and will be automatically replicated on the wearable or mobile by the system. `Asset`s are slightly different to `DataItem`s in that when replicated they get a file system handler allocated to them which must be used when reading the data back from the store.

2.4 Development Environment

Android Studio is the recommended Integrated Development Environment (IDE) to use when developing for Android Wear, although with the recent switch to the Gradle build system, it is still possible to build in other IDE's such as Eclipse, or even use no IDE at all and use the command-line Gradle wrapper to build the application.

For Android Wear to be able to communicate with a mobile device, the mobile device needs to have the official Android Wear companion app [2] installed and running. The companion app is necessary for pairing the two devices together in a secure way.

Android Wear system images can be downloaded using Android's SDK Manager and run in the emulator just like the regular versions of Android. Debugging using the emulator is swift and easy, but has it's limitations. If the application needs to communicate with the mobile device, the emulator must be set up to recognize it. To allow the emulator to communicate with another emulator or physical device, the ports on the machine must be setup using the command:

```
$ adb -d forward tcp:5601 tcp:5601
```

When debugging with physical devices, there are three ways in which the Android Debugging Bridge (ADB) can access the device:

1. USB
   This is the most straight-forward method for debugging and is as simple as connecting the device to the development computer and ensuring the device is visible to ADB and has the correct permissions to communicate with it.

[2] Google Play Store: http://goo.gl/MvmLsV

2. Bluetooth
   Debugging over Bluetooth is a bit trickier to set up, and much slower in terms of transferring data. The following commands are used to setup the necessary ports:

```
$ adb forward tcp:4444 localabstract:/adb-hub
$ adb connect localhost:4444
```

3. WiFi
   Debugging over WiFi is only possible with mobile devices as the wearables do not have WiFi capabilities. However to setup a mobile device for debugging over WiFi, the following commands are necessary:

```
$ adb tcpip 5555
$ adb connect #.#.#.#:5555
```

   where `#.#.#.#` is the ip address of the mobile device

Chapter 3

Analysis & Design

This chapter aims to give an overview of the architecture and structure of the main components of the project.

3.1 Overview

As previously mentioned, the project is made up of three main parts:

1. Research Project
   This project aims to explore the Android Wear APIs and display real world data obtained from the smartwatch's sensors. Since this project's sole purpose is to display experimental data, it's user interface is kept minimal and straightforward.

2. Wearable Student Application
   This is the main project of the three and is an application for both smartwatches and mobile/tablet devices. It's goal is to provide students with a simple and easy to use app to help them keep track of their lectures and results whilst also demonstrating some of the newer APIs available in Android "Lollipop"

3. Calendar Watchface
   This project provides an example of how to use the new Watchface APIs released by Google in January 2014. Previously, the APIs were undocumented but a working prototype was built in the Research Project and was converted to the new documented APIs for this project.

3.2 Application Architecture

An Android Wear application is bundled with a mobile application. Each application has it's own `AndroidManifest.xml` file and `res/` directory, but must use the same application id such as `com.example.app`. Additionally, the wearable application must declare the following in it's manifest:

```
<manifest ...>
    ...

    <!-- declare that this is a smartwatch application -->
    <uses-feature
        android:name="android.hardware.type.watch" />

    <application ...>
        ...
    </application>
</manifest>
```

The two applications are kept completely separated. As a result, the smartwatch app does not have access to resources or code of the mobile application and vise-versa. This can however be achieved using the Gradle build system and compiling one project as a module of the other project's `build.gradle` file. This can be especially useful when serializing objects and sending them from one application to the other at runtime and not having to duplicate the code.

When publishing the applications to the Google Play Store, a single APK is permitted. The Gradle build system will generate a single APK for the handheld application with the wearable application embedded inside the `raw` directory. When installing this APK onto a device, it should automatically transfer and install the wearable APK on the wearable device.

In order for the two applications to communicate, they use the DataLayer provided by Android Wear. The DataLayer is accessed via a `GoogleApiClient` which can be used to connect Google's various APIs, wearable being one of them. The structure of the application is shown in Figure 3.1

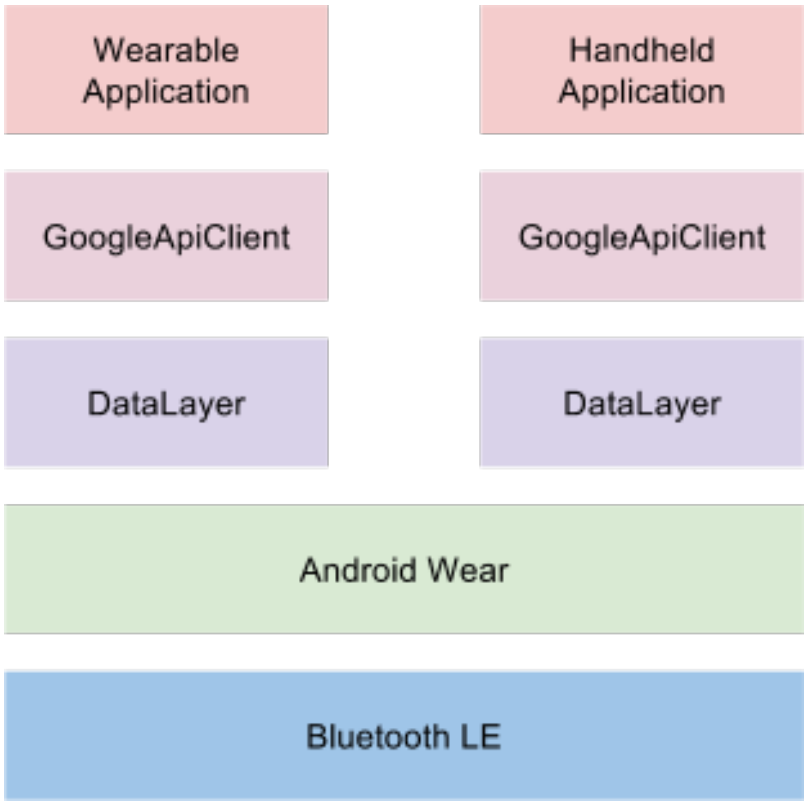

Figure 3.1: Application Architecture

Figure 3.2: Modules as shown in Android Studio

3.2.1 Module Structure

An Android application is comprised of multiple modules. Most mobile applications would only contain a mobile module, but with the introduction of Android TV, Auto and Wear, modules may be added to support each type of device. This project only caters for mobile and Android Wear and so consists of two main modules: one for the mobile application and one for the wearable application. Modules can also be added to share code between the other modules. In the case of this project a "common" module was created and added as a dependency of the "mobile" and "wear" modules. This allows code as well as resources such as images and layouts to be shared between the two modules.

The resulting structure is illustrated by Figure 3.2. Other libraries could also be included here as modules in Android Studio and get compiled and linked as part of the build process using Gradle.

Each module is an Android library project. This means that they each follow the Android project architecture shown in figure 3.3.

3.2.2 Package Overview

The package structure of the Student Application is quite straight-forward. Each module follows the basic package structure as shown below:

adapters - Contains `Adapter` classes for dealing with the data used by `ListViews` and `RecyclerViews`.

activities - Contains the applications Activity classes.

fragments - Contains the applications Fragments which are used by the Activities.

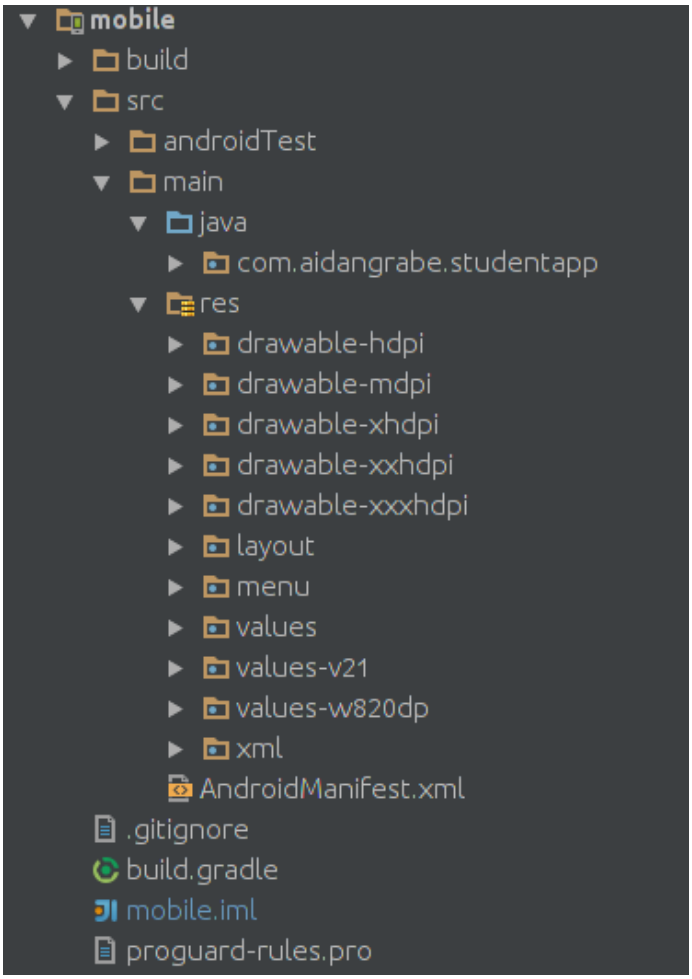

Figure 3.3: Android project structure

models - Contains simple classes that contain mostly attributes eg. Module, ToDoItem etc.

tasks - Contains `AsyncTask`s that run tasks in the background.

util - Contains classes that provide utility methods such as `ArrayUtil` ot `ColorUtil`.

views - Contains any custom view classes.

If a class is only ever used in a single module, it resides in that module. If a class needs to be used in both modules, it resides in the common module. This allows objects to be serialized across the connection and de-serialized on the other side.

3.2.3 API Overview

The Android Wear APIs are mostly focused around communication between the wearable and handheld devices. The structure of the main classes are shown in the class diagram depicted in Figure 3.4.

3.3 Problem Analysis

This section analyzes the problems faced by building an application for a handheld device and a wearable, along with the proposed solutions.

3.3.1 Communication

A fundamental part of the application is to communicate from the handheld application to the wearable application and vice-versa. Reliability, integrity and confidentiality are all key factors when discussing communication. Bluetooth is used as the communication layer for Android Wear and by default is not reliable or secure. Thankfully the Android Wear APIs build these features on top of the standard Bluetooth protocol. All communications over Bluetooth using the Android Wear APIs for communication are secured. At the time of writing the documentation is not clear on how it is secured.

In order to keep communications reliable, we have two options. Use Android Wear's DataApi or to use the MessageApi. The DataApi is slightly less efficient as it has

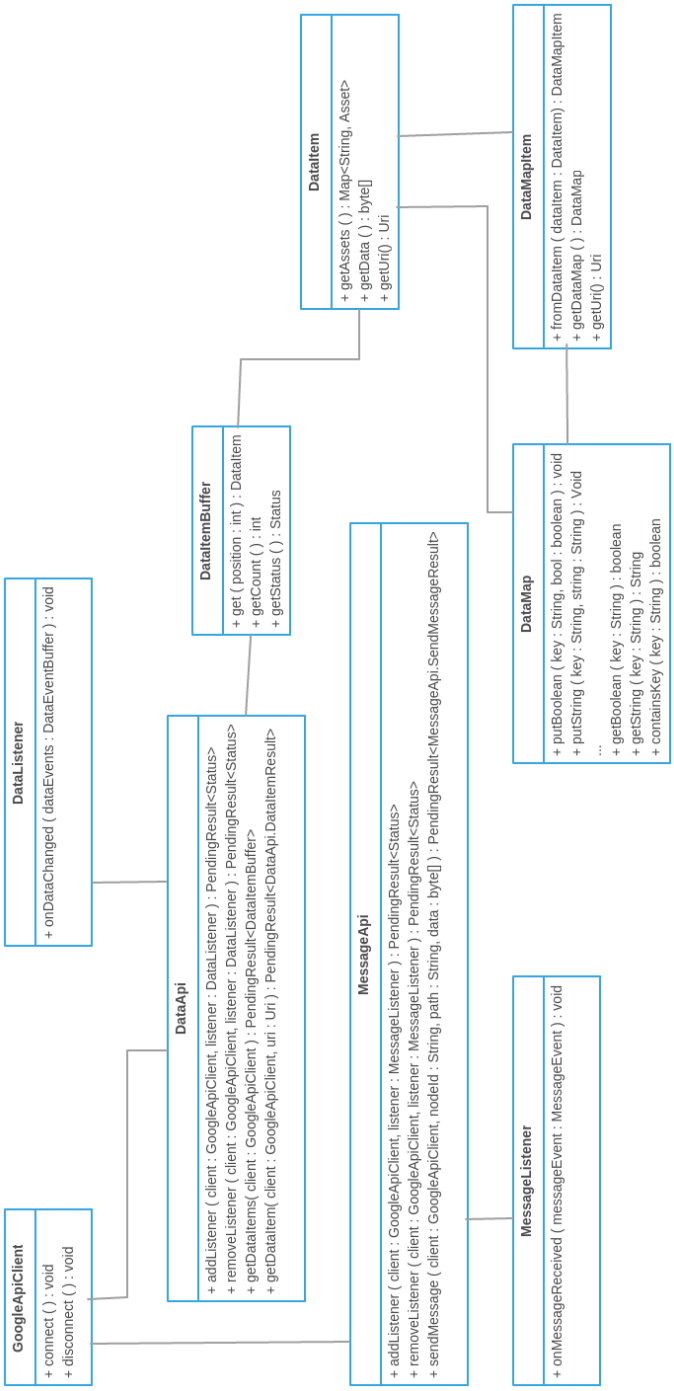

Figure 3.4: Class Diagram of the Main Android Wear classes

more overheads for syncing the data and making sure only objects that need to be sent are transferred via Bluetooth, but it comes with reliability built-in. If the wearable is not in range when the handheld application stores objects using the DataApi, the system buffers the messages until it can reliably transfer the objects to the wearable when it comes back in range.

The MessageApi does not guarantee reliability. In this case it is up to the programmer to handle the cases where messages are not delivered and resend them if necessary. As a result, if a message needs to be sent reliably in this project, it is sent using the DataApi. The MessageApi is kept for messages where reliability is not critical.

3.3.2 Synchronizing Data

Another problem that arises when two applications need to display the same data set is keeping the data sets on either side in sync. Luckily, the Android Wear developers have built this straight into the APIs with the DataApi. The DataApi will automatically replicate changes to a data set on both sides of the connection. It is up to the programmer to then handle the changes and store them somewhere on the device.

In this project, the handheld device will act as a server and the wearable as a client. The wearable will request resources using the MessageApi whenever it needs them. This prevents the wearable from having to keep the data in sync with the handheld everytime a change is made to the data set. This also means the wearable does not have to store any information on the device itself. Since the wearable has less memory and slower disk access times, it makes sense to leave the storage to the handheld device.

3.3.3 Availability

With this server-client architecture, the wearable will not be able to display much information to the user without being connected to the handheld. This is a problem faced by all client-server models. There is no solution to this problem, but measures can be taken to lessen it's impact such as caching information to display when there is no connection. However, this raises further problems of how and when to refresh this cached data. Cached data might also give the user a false impression that a connection is available, and frustrate them by not displaying the latest

information.

Alternatively, simply showing a message to the user that a connection is not available or that the handheld is not in range might be the cleanest solution. Many features of the wearable are only available when in range of the handheld, so this project's application would be no different. It also might prompt the user to rectify the problem, if they can just go pick up their handheld, not having realized it was too far away to communicate with, it would solve the problem entirely.

3.4 User Interface

The user interface for this application uses the standard Android SDK, along with some custom views implemented by drawing on a `Canvas` object. The main menu for the handheld is a `ListView` with custom layouts for each of the items.

The User Interface for the wearable application uses a `WearableListView` which is very similar to a regular `ListView` but also ensures that the list's items are not truncated when displayed on a circular screen. The input is also slightly different in that one item always snaps to the center of the screen making it easier to select on a small screen.

The watch face application displays an analog clock using the Canvas, but cannot take input as it is the watch's homescreen and so input is handled by the system for launching apps and starting voice commands etc.

Due to the amount of screens and different user interfaces throughout the applications, they will not all be discussed. Most of the handheld UIs are based on the `ListView` however, and the wearable's screens are mostly based on the `WearableListView` or just the regular `ListView` in some places. These screens are also discussed in more detail in next section.

3.4.1 UI Compatibility

Since Android Wear uses the same base version of Android as phones and tablets, applications are written in Java and configured using XML just like on mobile. Most layouts and views work without any changes, however some input fields such as `EditTexts` are useless since the screen is too small to comfortably fit a keyboard. Instead of requesting input from a user in the form of text fields, it is favourable

to use speech recognition to allow the user to input textual data. Some layouts and views are specific to Android Wear such as `WatchViewStub`, which allows the developer to select different layouts for square and circular screens.

At the time of writing there are multiple circular screen devices on the market such as the first circular smartwatch, the "Moto 360" from Motorola and the "G Watch R" from LG. By default, layouts will work on both rectangular and circular screens, but text/parts of the User Interface (UI) may be clipped depending on the radius of the screen and the pixel density.

When creating layouts and user interfaces for a wearable, special care must be taken to ensure buttons/targets are large enough to be easy to use without accidentally touching the wrong place, but also to ensure that all elements fit on the screen for both rectangular and circular screens. Often times this means using a different layout for rectangular screens and circular ones.

Chapter 4

Implementation

4.1 Language Choice

Java was the language used in implementing the applications outlined in the previous chapters. Android applications can be created using Java, C or C++. Since each application running on an Android device is executed in a Java Virtual Machine (JVM), Java is the primary development language used on the platform.
C or C++ could also be used thanks to the Android NDK, but as the official documentation suggests[1], should only be used for very specific, CPU-intensive tasks and one should not use the NDK because they prefer writing applications in C/C++.

In addition to these lower-level languages, many other languages can be used thanks to third-party transpilers such as PhoneGap (JavaScript), Apportable (Objective-C/Swift), Titanium (JavaScript) etc.
These tools allow developers to write code in different languages and either run them in a `WebView` using JavaScript and HTML5 or transpiles the source code into Java and compiles as if the developer had used Java in the first place.

[1]NDK documentation - http://goo.gl/OykZpe

4.2 Development Tools

4.2.1 IDE

When it comes to Android Development, two main IDEs stand out:

1. Eclipse
   Eclipse was the original official Android development environment and is still used by many developers. It uses a plugin called ADT (Android Development Tools) to interact with ADB and manage SDKs etc.

2. Android Studio
   Android Studio is currently the official IDE for Android and comes with many improvements over Eclipse. It is built on top of the popular IntelliJ IDE, and is tailored specifically for Android development. This allows it to have specific Android tools and functionality 'baked in', thus not requiring any plugins to begin development.
   Android Studio also comes with built-in functionality for the Gradle build system, allowing for easier dependency management and a more sophisticated build environment.

With Android's new Gradle build system, it is possible to use no more than a text editor and the Android build tools from the command line to compile and run an application. However, for building interfaces and the editing tools an IDE can offer, Android Studio was used.

4.2.2 Version Control

Android Studio has built-in support for the most popular version control systems. Git was chosen due to the sheer volume of libraries available on Github and Google's sample projects from the Android documentation site also use it.

4.3 Android Wear Communication

This section details how an Android device communicates with its Android Wear counterpart using Bluetooth and the Android Wear APIs. Communication was a

fundamental part of the project and is used to send messages, commands or entire objects from handheld to smartwatch and vice-versa.

4.3.1 Pre-requisites

In order for a smartwatch application to communicate with its counterpart handheld application, a few conditions must be met.

1. Android Wear Companion App
   As mentioned in previous chapters, the official Android Wear Companion App must be installed, and the wearable must connected and paired with the handheld through this app.

2. Package name
   Both the wearable and the handheld applications must use the same package name in their `AndroidManifest.xml` files. The APIs will not work correctly and communication will not occur between the applications if one of the package names is different.

4.3.2 Communication Types

With Android Wear there are two main types of communication:

1. Messages
   Messages are blobs of data limited in size to 100KB, which are sent from one device to another. The destination of the message must be known when sending.
   Messages are useful for RPC or a client-server messaging protocol.

2. DataItems
   DataItems are blobs of data which when created are automatically synced across the Android Wear network and replicated on all connected nodes. DataItems are useful for ensuring data is kept the same on both handheld and wearable.

All communication is handled by the system. The API for doing this is all dealt with by the `GoogleApiClient`. This API client connects to Google Play Services and allows for interaction with Google services, such as Android Wear, Play Games, Google Drive etc.
As a result, multiple APIs can be accessed via a single `GoogleApiClient`, however,

Google recommend using a single `GoogleApiClient` instance for dealing solely with Android Wear communications. This prevents failure callbacks for users that don't have a wearable device if they are trying to access another service.

```
GoogleApiClient mGoogleApiClient = new GoogleApiClient.Builder()
        .addApi(Wearable.API)
        .addConnectionCallbacks(mConnectionCallbacks)
        .addOnConnectionFailedListener(
            mOnConnectionFailedListener)
        .build();
```

Here we create a GoogleApiClient and tell it which API to connect to, which in this case is the Wearable API. We also set its connection callbacks and listeners. This allows us to get notified of connection events such as when the device is connected, suspended etc.

`mConnectionCallbacks` and `mOnConnectionFailedListener` might look something like the following:

```
mConnectionCallbacks = new ConnectionCallbacks() {
    @Override
    public void onConnected(Bundle connectionHint) {

        // device has been connected succefully to the
        // Wearable/Handheld

    }

    @Override
    public void onConnectionSuspended(int cause) {

        // the connection has been suspended. Device out
        // of range, or unavailable

    }
}

mOnConnectionFailedListener =
        new OnConnectionFailedListener() {
    @Override
    public void onConnectionFailed(ConnectionResult result) {
```

```
        // the connection failed to initialize

    }
}
```

In order to use the `GoogleApiClient`, it must be connected to Google Services. Here is how we achieve this with the Android Activity lifecycle:

```
public class MyActivity extends Activity {

    ...

    @Override
    public void onResume() {
        super.onResume();
        mGoogleApiClient.connect();
    }

    @Override
    public void onPause() {
        super.onPause();
        mGoogleApiClient.disconnect();
    }

    ...

}
```

4.3.3 Message API

Once the `GoogleApiClient` is connected, it can be used to send messages and/or `DataItems`. In order to send a message we need the following information:

1. A connected `GoogleApiClient`.
2. A `Node` to send the message to.

3. A path for the message so the receiving application can decide which action to apply.
4. A payload

All paths on Android Wear communications are relative to a base URI which Wear uses to communicate. These URIs are of the form: `wear://<node-id>/<path>`

```
Wearable.MessageApi.sendMessage(
        mGoogleApiClient,
        nodeId,        // String : the id of the receiving Node
        "/start/music", // String : the path of the message

        // byte[] : the payload of the message
        "song-123".getBytes()
);
```

On the receiving end, when a message is received, we can check whether it's the message we are interested in by checking the path:

```
...
if (path.equals("/start/music") {
    // open music player
}
```

Using this technique we can send multiple different messages and perform different actions on the handheld based on the path of the messages.

Receiving Messages

Sending messages from the handheld to the watch or vice-versa is only useful if the other side is listening for these messages. In order to perform actions or respond to received messages, the device must register itself with the Android Wear API and implement a callback method for when a message is received.
In order to start receiving messages from the counter-part application, the application must register itself as a listener to the `MessageApi`:

```
@Override
public void onResume() {
```

```
    super.onResume();
    // add the listener
    Wearable.MessageApi.addListener(mGoogleApiClient, mListener);
}

...

@Override
public void onPause() {
    super.onPause();
    // remove the listener
    Wearable.MessageApi.removeListener(mGoogleApiClient, mListener);
}
```

The listener for receiving message events looks like this:

```
@Override
public void onMessageReceived(MessageEvent messageEvent) {

    // get the path of this message's URI
    String path = messageEvent.getPath();

    // perform an action based on the path
    if (path.equals("/start/music")) {
        // start the music application
        Intent intent = new Intent(this, MusicActivity.class);
        startActivity(intent);
    }

}
```

Using this send-respond architecture, a client-server model can be created whereby the wearable can request data from the handheld using a custom protocol, and the handheld can reply with the requested data.
It also enables using the wearable as a remote to start activities or trigger actions on the handlheld, such as launching a compose new email activity or starting/stopping music playback.

4.3.4 Data API

Previously we discussed sending messages and performing actions upon receiving them. Android Wear provides another form of communication allowing developers to synchronize data across both the handheld and wearable. Whenever the data is updated on one side, an event is triggered on the other notifying it of the change.

This is useful for sending images, or keeping data consistent across both devices. For example, the handheld could retrieve contact data from it's database using SQLite and store it in the wearable `DataApi`. Provided the wearable has registered itself for receiving data changes, the data would then present itself to the wearable where it could be displayed or stored in it's own SQLite database.

Inserting DataItems to the Data Layer

In order to insert something into the data layer, we must first create a `PutDataRequest` which is very similar to a java `Map` but that deals with byte arrays. To make things simpler, we can use a `PutDataMapRequest`, which behaves more like a traditional `Map` where we can store key-value pairs of primitive types.

As with all communication on Android Wear, we must also specify a path for this data's URI. We can then hand this `PutDataMapRequest` off to the system for synchronization.

```
// create the PutDataMapRequest with a given path
PutDataMapRequest putDataMapRequest =
       PutDataMapRequest.create("/settings");

// get a more traditional map from the PutDataMapRequest
DataMap dataMap = putDataMapRequest.getDataMap();

// fill the map with some primitive data
dataMap.putString("color", "#FFAAFF");
dataMap.putFloat("rating", 4.5f);
dataMap.putInt("refresh_interval", 20);

// convert our traditional map back to a byte[] map
PutDataRequest putDataRequest = putDataMapRequest
       .asPutDataRequest();
```

```
// hand the request off to the system to put the map
// in the data layer
Wearable.DataApi.putDataItem(mGoogleApiClient, dataMap);
```

In the code above, we created a map containing some basic settings and put the map into the data layer. This means the data will be accessible from either device. If the handheld or wearable are not connected or out of range, the system will queue the request and perform when possible. This does not have to be handled by the developer.

Retrieving DataItems from the Data Layer

Retrieving `DataItems` from the data layer is slightly different. If we know the path of the data we want to retrieve, we can use

```
Wearable.DataApi.getDataItem(mGoogleApiClient, uri);
```

where `uri` is the `URI` of the item we want to retrieve. This involves building the URI manually which can be cumbersome and erroneous. A simpler way of retrieving `DataItems` is by fetching all items and filtering them by path as shown below.

```
DataItemBuffer dataItemBuffer = Wearable.DataApi
        .getDataItems(mGoogleApiClient)
        .await();

for (DataItem dataItem : dataItemBuffer) {
    String path = dataItem.getUri().getPath();

    // check we are using the correct
    if (path.equals("/settings")) {
        // create a DataMapItem from the given DataItem
        // so we can access our map
        DataMapItem dataMapItem = DataMapItem
                .fromDataItem(item);
        DataMap dataMap = dataMapItem.getDataMap();

        // get the settings back out of the map
        String color = dataMap.getString("color");
        Float rating = dataMap.getFloat("rating");
```

```
        int refreshInterval= dataMap.getInt("refresh_interval");
    } else if (path.equals("/other-data")) {
        ...
    }
}

// make sure to release the buffer to prevent leaking memory
dataItemBuffer.release();
```

Data Layer Events

In order to be able to synchronize data across the handheld and wearable devices, listeners must be created and added to the `DataApi`. Then, whenever `putDataItem()` is called, the listeners on both sides of the communications will be notified.

For example, when the wearable application first starts, it may call `getDataItems()` to get all the `DataItems` currently stored in the data layer and store the relevant information in memory. It might then add a listener to the `DataApi` in order to watch changes to this data and update its internal data structures for that data.

```
@Override
public void onResume() {
    super.onResume();
    Wearable.DataApi.addListener( mGoogleApiClient,
            mListener);
}

@Override
public void onPause() {
    super.onPause();
    Wearable.DataApi.removeListener( mGoogleApiClient,
            mListener);
}
```

The above code demonstrates how to add a listener for data layer events to an `Activity`. When the `Activity` is suspended to the background we genereally don't want to interact with the events. A `Service` can be used to interact with data layer

events when the application is not in the foreground. This will be explained in the next section.
Below is what `mListener` might look like for handling data layer events

```
mListener = new DataApi.DataListener() {

    @Override
    public void onDataChanged(DataEventBuffer dataEventBuffer) {

        // there may be multiple events handled at once
        for (DataEvent dataEvent : dataEventBuffer) {

            // get the DataItem from the event
            DataItem dataItem = dataEvent.getDataItem();
            String path = dataItem.getUri().getPath();

            // ignore items that we are not interested in
            if (! path.equals("/settings") {
                continue;
            }

            // we only want to perform actions on data that
            // has changed
            if (event.getType() == DataEvent.TYPE_CHANGED) {
                DataMap dataMap = DataMapItem
                        .fromDataItem(dataItem)
                        .getDataMap();

                // get the new settings from the map
                String newColor = dataMap.getString("color");
                Float newRating = dataMap.getFloat("rating");
                int newRefresh = dataMap
                        .getInt("refresh_interval");
            }
        }

    }

};
```

4.3.5 WearableListenerService

In this section communicating with wearables from a `Service` will be discussed. Android Wear APIs come with a special `Service` designed specifically for communicating with wearables: `WearableListenerService`.

In order to use `WearableListenerService`, it must be extended by a custom `Service`. Just like any regular Android `Service`, it must first be declared in the `AndroidManifest.xml` file but with a special `intent-filter`:

```
<service android:name=".DataLayerListenerService">
    <intent-filter>
        <action android:name=
            "com.google.android.gms.wearable.BIND_LISTENER" />
    </intent-filter>
</service>
```

Unlike a normal Android `Service`, it must not be started or stopped. The system handles this.
This `Service` must extend `WearableListenerService` and override it's communication methods. In order to listen for communication events, a `GoogleApiClient` must be created and the listeners must be attached as described in previous sections.

```
public class DataLayerListenerService extends
        WearableListenerService {

    @Override
    public void onCreate() {
        super.onCreate();

        mGoogleApiClient = new GoogleApiClient.Builder()
                ...
                .build();

        // since we are running in a Service, we can call
        // a blocking method. Calling this on the main thread
        // of an Activity would cause an error and the async
        // mGoogleApiClient.connect() would have to be used
```

```
        // instead
        mGoogleApiClient.blockingConnect();

        // attach the listeners
        Wearable.MessageApi.addListener(mGoogleApiClient, this);
        Wearable.DataApi.addListener(mGoogleApiClient, this);

    }

    @Override
    public void onDestroy() {
        super.onDestroy();

        // remove the listeners
        Wearable.MessageApi.removeListener(mGoogleApiClient, this);
        Wearable.DataApi.removeListener(mGoogleApiClient, this);

        // ensure we disconnect from the Wearable API
        mGoogleApiClient.disconnect();
    }

    @Override
    public void onPeerConnected(Node peer) {
        super.onPeerConnected(peer);

        // called when the wearable and handheld have been
        // connected
    }

    @Override
    public void onDataChanged(DataEventBuffer dataEvents) {
        // code to deal with changed or delete data events
    }

    @Override
    public void onMessageReceived(MessageEvent messageEvent) {
        // code to run when message received
    }

}
```

This type of `Service` allows us to listen for communication events even when the application is not in the foreground. This is quite a common case as it is quite rare that a user would have the application open on both the wearable and the handheld at the same time. In the Student Application discussed in this report, a `WearableListenerService` is created on handheld and acts as a server to the wearable. The `Service` could just as easily have been created on the wearable, or on both.

4.3.6 Syncing Assets

Up to this point, syncing data and sending messages have been limited to 100KB in payload. In order to send data of larger size such as images or audio files, `Assets` must be used. `Assets` are attached to `DataItems` and are stored and retrieved in a similar way to any other data stored in a `DataItem`.

```
// get the Bitmap to send from the device's resources
Bitmap bitmap = BitmapFactory.decodeResource(
        getResources(), R.drawable.ic_launcher);

// here we need to convert the Bitmap into an Asset
// an Asset can be created from a byte array, so we must
// get the byte array from the Bitmap
ByteArrayOutputStream byteStream =
        new ByteArrayOutputStream();
bitmap.compress(Bitmap.CompressFormat.PNG, 100, byteStream);
Asset asset = Asset.createFromBytes(byteStream.toByteArray());


// here we create a PutDataMapRequest just like we did
// for putting primitive values into the data layer
PutDataMapRequest putDataMapRequest =
        PutDataMapRequest.create("/image");

// get the DataMap for this PutDataMapRequest
DataMap dataMap = putDataMapRequest.getDataMap();

// add the Asset to the DataMap
dataMap.putAsset("icon", asset);
```

```
// sync the DataMap containing the Asset into the data layer
Wearable.DataApi.putDataItem(mGoogleApiClient,
        dataMap.asPutDataRequest());
```

In order to receive the Asset on the other side of the communication, we must implement the `DataApi.DataListener` and check for the event in `onDataChanged()`.

```
...

@Override
public void onDataChanged(DataEventBuffer dataEvents) {

    for (DataEvent dataEvent : dataEvents) {
        DataItem dataItem = dataEvent.getDataItem();
        String path = dataItem.getUri().getPath();

        if (path.equals("/image")) {
            DataMap dataMap = DataMapItem
                    .fromDataItem(dataItem)
                    .getDataMap();

            // retrieve the Asset from the DataMap
            Asset asset = dataMap.getAsset("icon");
            onAssetReceived(asset);

        }
    }

}

private void onAssetReceived(Asset asset) {

    // Assets have unique file descriptors on the device
    // get the file descriptor for the given Asset
    // this method is called asynchronously, to get an
    // immediate repsonse, await() could be used, but only
    // in a background thread
    Wearable.DataApi.getFdForAsset(mGoogleApiClient, asset)
```

```
            .setResultCallback(new
                ResultCallback<DataApi.GetFdForAssetResult>()
                    {

        @Override
        public void onResult(DataApi.GetFdForAssetResult
                getFdForAssetResult) {

            // re-construct the Bitmap from the file
            InputStream assetInputStream =
                    getFdForAssetResult.getInputStream();
            Bitmap mapBitmap = BitmapFactory
                        .decodeStream(assetInputStream);

            // do something with the received Bitmap

        }

    });
}

...
```

4.3.7 Object Serialization

When communicating between handheld and wearable, a common use case might require a Java object to be used on both devices, for example a calendar event. In order to achieve this, the object must be broken down into it's primitive data, transfered as an array of bytes and reconstructed on the other side of the connection. Android has built-in interfaces for this, namely the `Parcelable` interface.

When an object implements `Parcelable` it describes itself in terms of it's primitive data such as `Integer`s, `Float`s and `String`s. This data can then be written to a `Parcel` object and "marshalled" into a byte array. `Parcelable` objects can be nested within other `Parcelable` objects. The `Parcelable.Creator<T>` interface specifies how an object of type `T` should be recreated from a `Parcel`.

For Android Wear, `DataMap`s could also be used instead of `Parcel`s, but since `Parcel`s are used frequently in Android development, such as passing an object between

Activities, `Parcel`s may be more versatile for re-use.

Below is an example of a simple object which implements `Parcelable`

```
public class CalendarEvent implements Parcelable {

    private String title;
    private int numAttendees;
    private Date date;

    // nested Parcelable oject
    private Parcelable myObject;

    ...

    // Describe the kinds of special objects contained in
    // this Parcelable's marshalled representation
    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel out, int flags) {
        out.writeString(title);
        out.writeInt(numAttendees);
        out.writeLong(date.getTime());
        out.writeParcelable(myObject);
    }

    // the object that will handle reconstructing the event
    public static final Creator<CalendarEvent> CREATOR =
            new Creator<CalendarEvent>() {

        @Override
        public CalendarEvent createFromParcel(Parcel source) {
            // reconstruct the CalendarEvent from the given
            // Parcel
            CalendarEvent event = new CalendarEvent();
            event.setTitle(source.readString());
```

```
            event.setNumAttendees(source.readInt());
            event.setDate(new Date(source.readLong()));

            // reconstruct the nested Parcelable object
            event.setMyObject(source.readParcelable(
                MyObject.class.getClassLoader()));
            return event;
        }

        @Override
        public CalendarEvent[] newArray(int size) {
            // create a new CalendarEvent array
            return new CalendarEvent[size];
        }

    };

}
```

Now when an instance of the `CalendarEvent` class is created, it can be put into a parcel and marshalled as follows:

```
// get a free Parcel instance
Parcel parcel = Parcel.obtain();
CalendarEvent event = new CalendarEvent();

// write the event into the Parcel using the methods
// implemented above
event.writeToParcel(parcel, 0);

// set the data pointer back to the start ready for reading
parcel.setDataPosition(0);

// to then be able to transfer the Parcel with Android Wear
DataMap dataMap = ...
String key = "cal-event";
dataMap.putByteArray(key, parcel.marshall());

// the DataMap can then be synced using the data api or
```

```
// message apis

// recycle the Parcel when it is no longer needed
parcel.recycle();
```

In order to read the serialized `Parcel` on the other side of the connection, a `Parcel` must be obtained, and it must unmarshall the byte array from the `DataMap`. Using the `CREATOR` instance created above, the `CalendarEvent` can be reconstructed from the `Parcel`.

```
DataMap dataMap = ... // get the DataMap from Android Wear

byte[] byteArray = dataMap.getByteArray(key);

Parcel parcel = Parcel.obtain();

// unmarshall the byte array into the new Parcel
parcel.unmarshall(byteArray, 0, byteArray.length);

// reset the data pointer to the start for reading
parcel.setDataPosition(0);

// read the parcel data, and create a CalendarEvent
CalendarEvent event = creator.createFromParcel(parcel);

parcel.recycle();
```

Of course if the code permits, a `DataMap` could be used directly instead of a `Parcel`, but using a `Parcel` may allow the developer to serialize the object for saving to disk or passing the object between processes or activities.

4.3.8 Communication Threads

The Android Wear communication APIs described in previous sections are all carried out asynchronously with callback methods called in background threads. The APIs return a `PendingResult<T>` instance with the relevant results from the API call.

Android does not allow updating of the user interface from a thread other than the main UI thread. In order to update the UI on a result from an Android Wear API call, a `Handler` must be used to post the UI code to the main UI thread for execution. This can be done as follows:

```
PendingResult<MessageApi.SendMessageResult> result;
result = Wearable.MessageApi.sendMessage(
        mGoogleApiClient, nodeId, path, messageBytes);

result.setResultCallback(new ResultCallback
        <MessageApi.SendMessageResult>() {
    @Override
    public void onResult(MessageApi.SendMessageResult result) {
        // called on a background thread when the message
        // has been sent
        onMessageResult(result);
    }
});

private void onMessageResult(final String result) {

    // perform non-UI logic here
    doSomething();

    // perform UI logic here
    (new Handler(Looper.getMainLooper())).post(new Runnable() {
        @Override
        public void run() {
            mTextView.setText("Result: "
                + result.getStatus().toString());
        }
    });

}
```

In order to run the UI logic on the main thread, a `Runnable` is posted to the main `Handler` and gets scheduled for processing.

If an Android Wear comunications API method is called from a thread that is already

running in the background, there is no need to use callbacks at all. `PendingResult`s have an `await()` method which blocks execution until the result is ready for processing. If multiple API calls need to be made in immediate succession, it might make sense to run them all on a background thread and just use one callback for the overall result. This can be achieved using an `AsyncTask`. The `await()` method cannot be run on the main thread, if it is, the application will crash and throw an exception.

```
(new AsyncTask<Void, Void, Void>() {

    @Override
    protected Void doInBackground(Void... params) {
        // make API calls
        MessageApi.SendMessageResult result;
        result = Wearable.MessageApi.sendMessage(
            mGoogleApiClient, nodeId, path, messageBytes).await();
        if (result.getStatus().isSuccess() {
            // make another API call
        }
    }

    @Override
    protected void onPostExecute(Void aVoid) {
        // execute callback method here
        // this method runs on the UI thread
    }

}).execute();
```

4.4 Sensors

Sensors allow devices to read data from the outside world. Applications can use sensor data to monitor real-time data and perform actions on it, or present it to the user in a more digestible way.

Android supports measuring raw data from many different types of sensors. The availability of sensors depends on the device and version of Android it is running.

Android Wear is no different. Android Wear allows for the same APIs to be used in detecting and measuring data from built-in sensors.

Android Wear provides the user with access to sensors a regular Android device might not have access to. This project was carried out and tested on a Samsung Gear Live device. On this device, there is a heart-rate monitor which can measure the rate at which the heart beats by using a bright LED on the back of the watch to illuminate the top of your wrist and measure the rate of the pulsing of the blood. This method requires the wearer to be completely still and that the LED be completely clean to give an accurate reading. Even when these conditions are met, there is still a certain skepticism as to how accurate this type of monitor really is.

In any case, Android Wear provides the same APIs for accessing sensors on a wearable as those used to achieve the same on a handset. The `SensorManager` is used to query the availability of different `Sensor`s at runtime. A `Sensor` object then provides methods allowing you to discover the capabilities and retrieve information regarding the underlying sensor such as resolution, maximum range.
The `SensorManager` also allows us to attach listeners to each sensor to listen for events related to the sensor, such as when the accuracy of the sensor changes or when the values of the sensor are updated or changed by the underlying hardware for the sensor.

The below code listing demonstrates how one might sense heart-rate data from a wearable device. The code could also be run on a handset device with no errors, but would not return any data unless it was equipped with a heart-rate monitor.

```
public class HeartRateActivity extends Activity
       implements SensorEventListener {

    // the SensorManager is used to retrieve the heart-rate
    // sensor
    private SensorManager mSensorManager;

    // this is the heart-rate sensor itself which will be
    // used to measure the raw data
    private Sensor mHeartSensor;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
    mSensorManager = (SensorManager)
            getSystemService(SENSOR_SERVICE);

    // ask the SensorManager for a heart-rate sensor
    // if there is no heart rate sensor present, this
    // will return null
    mHeartSensor = mSensorManager
            .getDefaultSensor(Sensor.TYPE_HEART_RATE);

}

@Override
public void onResume() {
    // start listening for events on the heart sensor
    mSensorManager.registerListener(this, mHeartSensor,
            SensorManager.SENSOR_DELAY_NORMAL);
}

@Override
public void onPause() {
    mSensorManager.unregisterListener(this,
        mHeartSensor);
}



@Override
public final void onAccuracyChanged(Sensor sensor,
        int accuracy) {
    // called when the accuracy of the sensor has changed
}

@Override
public final void onSensorChanged(
        SensorEvent event) {
    // event.values can be more than one
    // eg. accelerometer returns 3 values
    float heartRate = event.values[0];

    Log.d("HeartRateActivity", String.format(
```

```
            "Received heart rate %.0f", heartRate));
    }

}
```

Figure 4.1: Activity monitoring heart-rate

If a higher degree of accuracy is needed, a higher sample rate can be specified for measuring the raw sensor data. It should be noted however, that selecting a higher sampling rate can consume more power than a slower sampling rate. As a result it is recommended to use a high sampling rate only when necessary.
In the above example the default sampling rate of `SensorManager.SENSOR_DELAY_NORMAL` was used. The available sampling rates are listed below:

1. SENSOR_DELAY_NORMAL (default) - 200ms
2. SENSOR_DELAY_UI - 60ms
3. SENSOR_DELAY_GAME - 20ms
4. SENSOR_DELAY_FASTEST 0ms

When attempting to listen on a sensor that is not available, the system should continue running. When an application requires the use of a specific sensor, that is to say the application needs it to function, it should declare so in it's `AndroidManifest.xml` file. eg.

```
<uses-feature
    android:name="android.hardware.sensor.accelerometer"
    android:required="true" />
```

Note that this is not necessary, but would help filter capable devices on the Google Play Store if submitted.

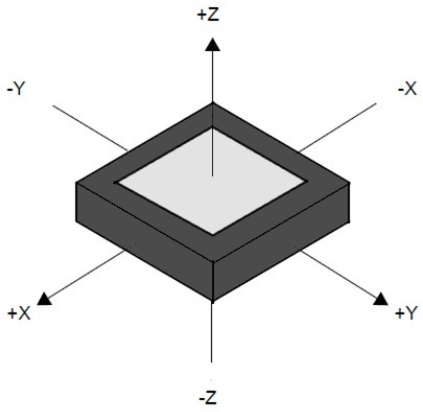

Figure 4.2: Three axes of the accelerometer

4.5 Gestures

Android Wear smartwatches come with accelerometers and gyroscopes, this means the devices can distinguish their orientation in 3d space. Using these sensors we can determine movement or gestures by the user, just like on a smartphone. Smartwatches have the added benefit of being located on a user's wrist, the perfect spot for detecting gestures created by the user's arm. One of the goals of this project was to be able to detect simple linear gestures ie. when the user thrusts their arm up, down, left, right, forwards or backwards.

Listening to the accelerometer data is the same as listening to any other sensor on the device which is explained in the previous section. However, detecting 3d gestures is not straight forward, and very few libraries exist for detecting such gestures. An accelerometer measures the acceleration on 3 different axes: x, y and z as shown in figure 4.2. Gestures in this project are detected by waiting for a value of x, y or z from the accelerometer to exceed a certain threshold. Since the values of x, y and z can be negative or positive, some calculations must be done first in order to determine on which axis the gesture occurred.

```
double, absX, absY, absZ, biggest;
absX = Math.abs(x);
absY = Math.abs(y);
absZ = Math.abs(z);
```

```
// get the largest value
biggest = Math.max(Math.max(absX, absY), absZ);

// see if the largest value is above the minimum threshold
if (biggest > GESTURE_THRESHOLD) {
    // do gesture detection
}
```

The above method works fairly well for ignoring small movements and accidentally triggering a gesture, but if the user does a fast movement to trigger a gesture in one direction, an equal but opposite gesture will be triggered almost immediately after as the user's hand comes to a stop. In order to ignore this second stopping gesture, a delay is used after registering the first gesture, so no gestures can be detected until after this delay. This prevents the pair of gestures from being detected together. Care must be taken to use a threshold high enough so that the second gesture is ignored, but low enough so that a second unique gesture is not accidentally missed as well. In tests, 700ms to 1 second seemed to be an adequate delay.

Once the threshold as been exceeded, we know a gesture has occurred, the next step is to figure out which axis the gesture occurred on. This can be checked using a simple multi-arm conditional statement.

```
String gestureLessThan, gestureGreaterThan;
double val = x;

// check which axis the gesture occurred on
if (absY == biggest) {
    gestureLessThan    = "DOWN";
    gestureGreaterThan = "UP";
    val = y;
} else if (absZ == biggest) {
    gestureLessThan    = "FORWARDS";
    gestureGreaterThan = "BACKWARDS";
    val = z;
} else {
    gestureLessThan    = "LEFT";
    gestureGreaterThan = "RIGHT";
}
```

```
// decide whether the movement was positive or negative
String gesture = val > 0 ? gestureGreaterThan : gestureLessThan;
Log.d("DEBUG", "Gesture: " + gesture);
```

4.6 Notifications

Notifications are a key part of Android Wear as they are not applications which need to be launched by the user. They are a way of displaying short glanceable information to the user that can have certain quick-actions such as replying to a text message, or opening an application on the handheld in order to complete a larger action. Notifications are at the heart of what Android Wear is all about and are easily created and displayed using the Android APIs.

By default, a notification built for a handheld application will display on Android Wear without any modifications. They can however be "extended" to show information that would be displayed specifically for wearable devices.

```
// the intent to start when the notification is clicked
Intent intent = new Intent(this, TimetableActivity.class);

// build the notification
NotificationCompat.Builder notificationBuilder =
        new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.ic_notif)
        .setContentTitle("Sample Notification")
        .setContentText("This is a sample notification")
        .setContentIntent(viewPendingIntent);

// get an instance of the NotificationManager service
NotificationManagerCompat notificationManager =
        NotificationManagerCompat.from(this);

notificationManager.notify(notificationId, notificationBuilder.build());
```

The code listing above describes how to build a notification that can be displayed on a handheld device. By default, this notification will also get displayed on the

wearable. In order to add wearable specific-features to the notification, we use a `WearableExtender`:

```
// Create a WearableExtender to add functionality for wearables
NotificationCompat.WearableExtender wearableExtender =
        new NotificationCompat.WearableExtender()
        .setBackground(mBitmap);

// extend the notification with the wearable-specific features
notificationBuilder.extend(wearableExtender);

// display the notification
notificationManager.notify(notificationId, notificationBuilder.build());
```

Here we can set the background of the notification which will be displayed on the wearable. This background will not get displayed on any handheld devices. By default, adding actions to the notification will also work on the wearable, but sometimes it is desirable to show certain actions only on the wearable, such as "Open on phone". This action should not be displayed in the notification of the phone itself. This can be accomplished by using the `extend()` method of the `NotificationBuilder`

```
new NotificationCompat.Builder(mContext)
        .setSmallIcon(R.drawable.ic_message)
        .setContentTitle(getString(R.string.title))
        .setContentText(getString(R.string.content))

        // add wearable only actions
        .extend(new WearableExtender().addAction(action));
```

4.7 Student Application

The following section will describe the implementation of the Student Application for handheld devices and it's wearable counterpart app. The handheld app consists of a main menu (shown in Figure 4.3) which displays all the available sections of the app.

The handheld application uses SQLite for it's database, and uses the open source Sugar ORM[2] for it's Object-Relational Mapping capabilities. This allows much easier storage of Java objects in a relational database and Sugar ORM provides much of the database boilerplate code through the use of Reflection.

The sections of the app are as follows:

1. Modules
   This screen allows the user to manage their modules or subjects for college. They can add or remove different modules which are then used by other screens within the app.

2. To-do list
   The user can create reminders and to-do items in this screen. To-do items can be created, removed or marked as completed. These to-do items are also synced across to the wearable where they can be viewed and marked as complete.

3. Timetable
   In this screen, the user can view their timetable. Lectures are created on this screen and contain a title, room, day and time.

4. Results
   Results for each module are displayed on this screen and new results can be added or removed in sub screens.

5. News
   This screen displays the recent news from the UCC website [3] in the form of a native Android RecyclerView, the articles are shown when the article image or headline is clicked by the user.

6. Campus Map
   This section is only contained on the wearable app and is designed to have the coordinates of the UCC campus preloaded onto the handheld, which then downloads images of the map from Google Maps and displays them on the wearable in case the user is lost on campus.

7. Games
   This section contains only one game on the handheld, and that is a game of Snake. The Snake game is played using wearables or other mobile devices as controllers and is played over Bluetooth.

[2]Sugar ORM: http://satyan.github.io/sugar/
[3]UCC News page: http://www.ucc.ie/en/about/uccnews/

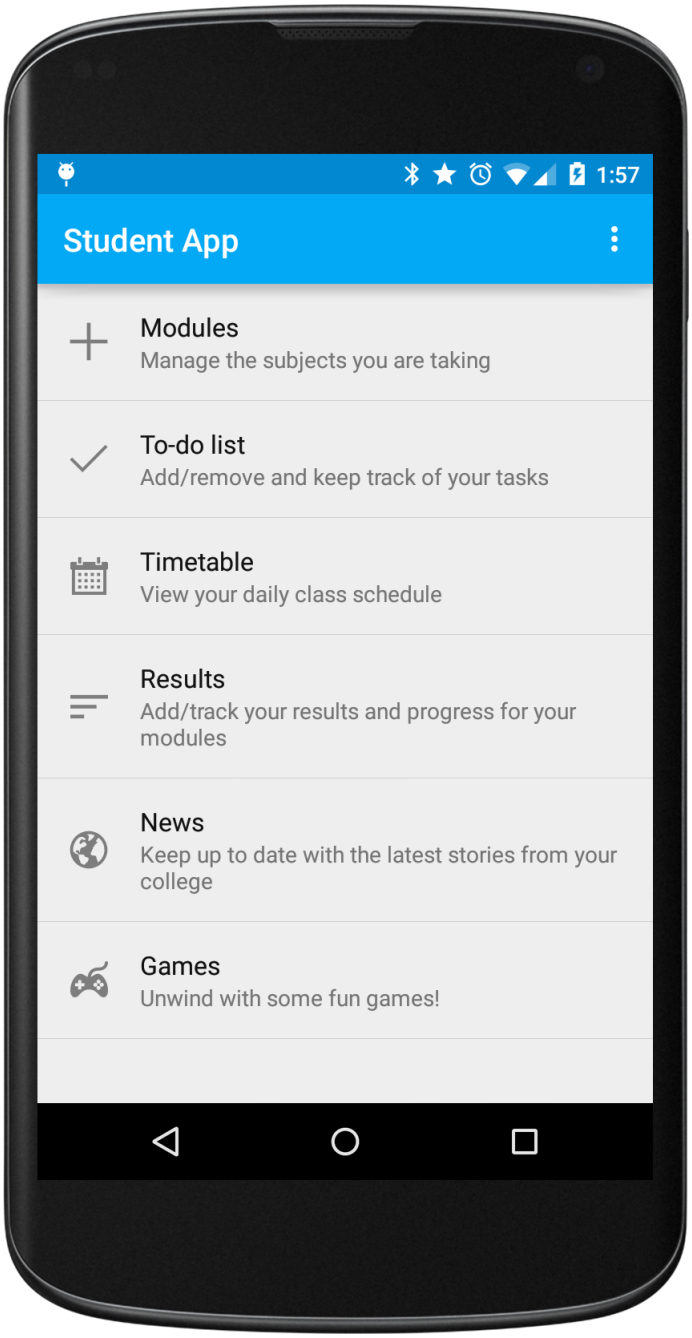

Figure 4.3: Handheld main menu

4.7.1 Modules

When the user creates a new module, a `Module` object is created. This `Module` instance is also `Parcelable`. The new instance is stored in the handheld device's SQLite database. All modules in this database are then read into a `List`, parceled up, and synced to the wearable device.

In order to sync the whole list over to the weearable, the objects in the list must implement `Parcelable`. Once the list is put into the data layer of Android Wear, the wearable can read and display the list of modules by calling `Wearable.DataApi.getDataItems` and filtering by the path of the modules which in this case is `"/modules/get"`.

Although using this method to synchronize the modules to the wearable works, and works quite quickly as the module objects are small in size and quick to transfer. It is not the most optimal method for syncing the modules. A better approach would be to implement a database on the wearable device as well as the handheld and only put the new module object into the data layer when it is created. This means that only one module object would be transfered when a new module is created rather than the entire list.

However, using this method would require a database on both sides of the connection, and would also require the wearable application to be running a `WearableListenerService` to be able to spot the changes to the data layer when the wearable app is not in the foreground. This is a lot of extra complexity for a list of modules, that realistically should never reach over 10-15 modules. If the dataset was much larger, the more efficient solution would be a far better approach and would provide a noticeably better user experience, in terms of a more responsive UI.

4.7.2 To-Do List

The to-do list is implemented rather similarly to the Modules screen. The major difference between the screens is that to-do list items can be created on the wearable device as well as the handheld. This means that while the application is in the foreground on the handheld, the dataset might change after it has been retrieved from the database.

Figure 4.4: SpeechRecognizer Activity

Creating items on the wearable

Creating to-do items on the handheld is straight-forward. An `EditText` is used to get the title of the to-do item and it is stored in the database. However, since the wearable's screen is too small to house a full-blown keyboard, another form of input must be used.

The only alternate way of getting textual input from the user on a wearable is to use the user's voice as input. Android Wear comes with APIs for doing exactly that. The wearable uses a `SpeechRecognizer` activity which listens for the user's voice and returns the speech data to the calling Activity.

```
// Create an intent that can start the Speech Recognizer activity
private void displaySpeechRecognizer() {
    Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
    intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
            RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
    // Start the activity, the intent will be populated with the speech
        text
    startActivityForResult(intent, SPEECH_REQUEST_CODE);
}
```

Calling `displaySpeechRecognizer()` above will start a system Activity (Figure 4.4)

which will display a screen which reads "Speak now". This screen will wait for the user to speak, display the text it has heard, or "Sorry, didn't catch that". This text is then returned to the calling Activity for processing.

In order to create a module from this text, we must override the `onActivityResult()` method of the Activity. This is achieved as follows:

```
// This callback is invoked when the Speech Recognizer returns.
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
    if (requestCode == SPEECH_REQUEST_CODE && resultCode == RESULT_OK) {
        List<String> results = data.getStringArrayListExtra(
                RecognizerIntent.EXTRA_RESULTS);
        String spokenText = results.get(0);

        // only create a new item if the string is non-empty
        if (spokenText.length() > 0) {
            ToDoItem item = new ToDoItem(spokenText);
            syncToDoItem(item);
        }
    }
    super.onActivityResult(requestCode, resultCode, data);
}
```

Syncing the items

When a new to-do list item is created on the handheld, it is stored in the database and synced to the wearable in the same way as a module is synced in the previous section. This means that the wearable can access the entire list of to-do items from the Android Wear data layer. However, if the user creates a new to-do item on the wearable, the wearable stores the new to-do item in the data layer at a different path. The handheld needs to be able to see this change and store the new to-do item in it's database. This is made possible by the `WearableListenerService` implemented on the handheld. When data is changed in the data layer by the wearable, its `onDataChanged()` method is called and if the path matches the path of the new item created by the wearable, it can read the item from the data layer and store it in the SQLite database. It will also update the list of to-do items in the data layer

so the wearable, on subsequent calls to `Wearable.DataApi.getDataItems`, will get the most recent list of to-do items, including the item that was just created on the wearable.

To-do items are now synced between both devices, but if the user creates an item on the wearable while the handheld application is in the foreground, we want the handheld's user interface to update with the new item. This is an unlikely use-case that the user would have both applications in the foreground, but is definitely a possibility.
To solve this issue, the `Fragment` displaying the to-do items on the handheld can implement `DataApi.DataListener` in order to spot changes to the data layer. When the fragment spots a change on the path `"/todo/create"`, it can update its list's `Adapter` and refresh the data displayed to the user.

This change was also applied to the wearable application as it is also possible to create a to-do item on the handheld while the wearable app is in the foreground. Once both sides have the change applied, the user can create to-do items on either side of the connection with both apps in the foreground and see the changes immediately on both UIs.

4.7.3 Timetable

The timetable screen displays the user's lectures for the week in the form of tabbed lists (Figure 4.5). Each day is its own tab and displays the days lectures in order of earliest first. New lectures can be created by pressing the Floating Action Button in the bottom right corner. Only days that have lectures are displayed in the timetable, that is, if the user has not created lectures for Saturday and Sunday, they will not appear on the screen.

Each lecture is stored in the SQLite database using Sugar ORM. Using Sugar ORM is very simple. In order for a Java object to be stored in the database using Sugar ORM, it must extend `SugarOrm<T>` where `T` is the class that will be stored. For example, the `Lecture` class looks something like this:

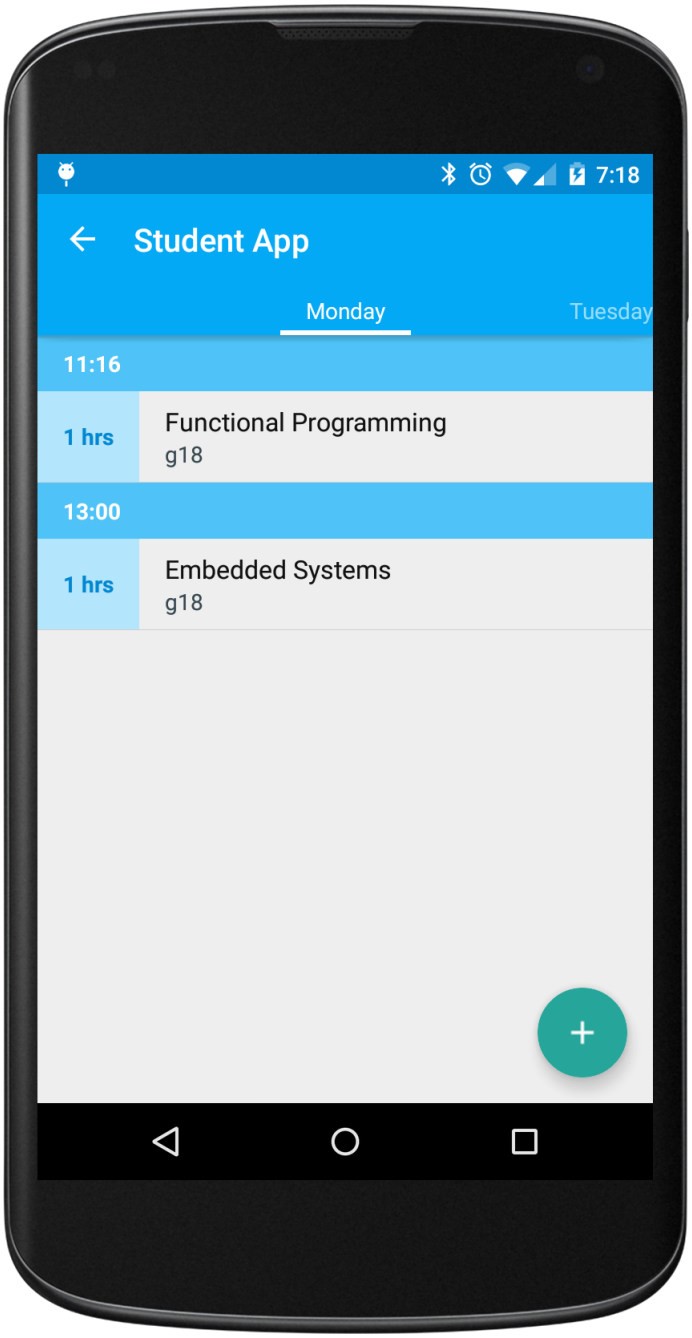

Figure 4.5: Timetable screen

```
public class Lecture extends SugarRecord<Lecture> {

    private String name;

    // an empty constructor is required by Sugar ORM
    public Lecture() {}

}
```

In the above listing, name will be a column in the database and an instance will automatically populate it in the database with it's value when it is stored using Java Reflection. If the object contains data that should not be inserted into the database upon storage, it can be ignored using the `@Ignore` annotation. For example:

```
private long id;

// ignore the password attribute
@Ignore
private String password;

private String title;
```

`Lecture` objects can now be stored by calling `lecture.save()`. In order to retrieve lectures from the database using the ORM, we can use `Lecture.findById( Lecture.class, id)` to find a particular lecture once it's id is known, or `Lecture.findAll( Lecture.class)` to list all lectures currently stored in the database.

The timetable screen is unique to the handheld device and does not have any counterpart on the wearable. This is mainly down to the limited screen real estate available on the wearable. Instead however, notifications are used to display upcoming lectures. This is achieved using Android's `AlarmManager` to schedule a task every 10 minutes to check the created lectures and see if any are occuring within the next 15 minutes. If a lecture is starting in this time, a notification is built and displayed on both the handheld and wearable devices. When the notification is clicked, the timetable Activity is opened.

4.7.4 Results

The results screen displays a list of the user's modules with an animated progress bar showing their average result for each module. Clicking on a module brings the user to a new screen which displays each result for that module in reverse-chronological order. This screen also displays an animated graph of each of the results so the user can see their progress at a glance.

The graph is a custom view which overrides the `onDraw()` method and uses paths to create the shape of the graph. Each vertex of the graph is animated from the 50% mark and the animation is offset by the vertex's x position so each vertex animation begins a few milliseconds after the previous vertex's animation has begun.

This custom view was created in the `common` module so that the view could be used on both the handheld and wearable devices. The main reason for the addition of this graph was to see how the wearable could handle custom views and animations. The animations run smoothly and without problem on both devices.

The animations were achieved using Android's `ValueAnimator` class. This class enables us to animate particular values from a start value to an end value. To animate the results, each vertex in the graph is set to show 50% and is then animated to it's actual value.

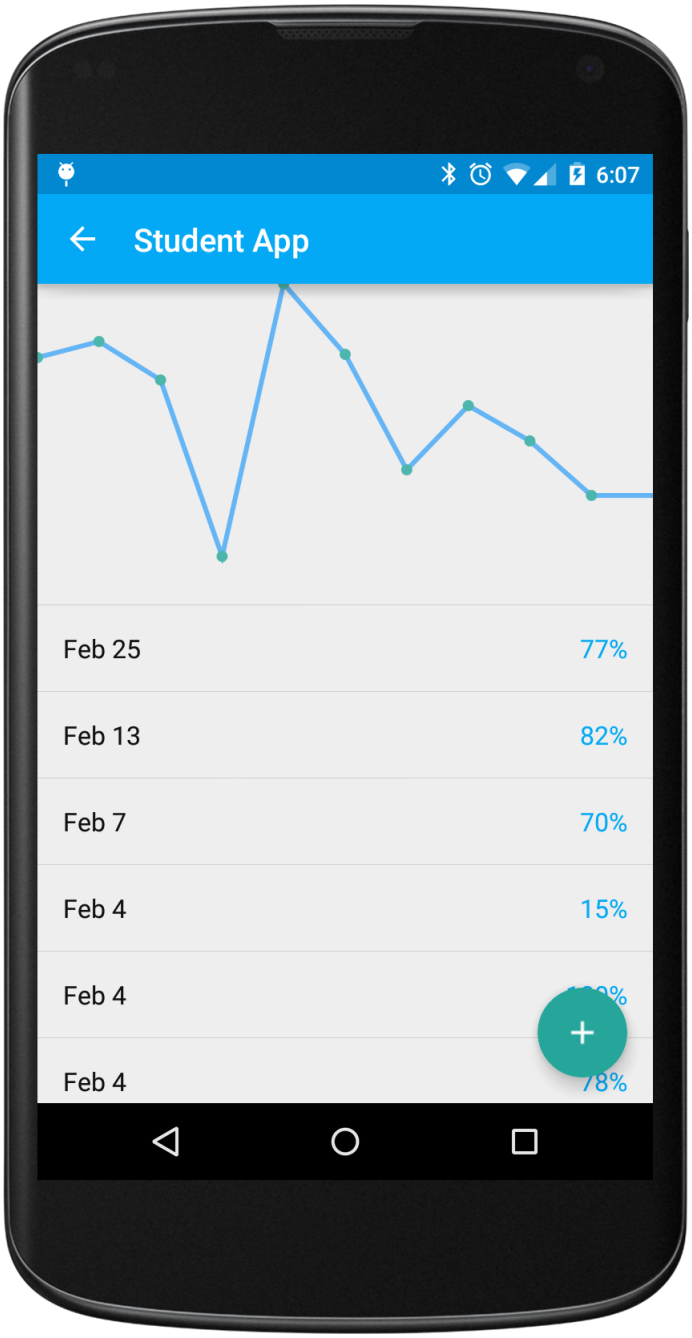

Figure 4.6: Custom Graph View on handheld

Figure 4.7: Custom Graph View on wearable

```
public void animateValues() {
    mAnimations = new ArrayList<>();
    for (int i = 0; i < mValues.size(); i++) {
        final float value = mValues.get(i);
        final int index = i;
        ValueAnimator anim = ValueAnimator.ofFloat(.5f, value);

        // offset each animation by 100 ms
        anim.setStartDelay(500 + (100 * i));
        anim.setDuration(1500);

        // make each vertex start fast and slow down
        anim.setInterpolator(new DecelerateInterpolator(3f));
        anim.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
            @Override
            public void onAnimationUpdate(ValueAnimator animation) {
                // update the value of the vertex
                mValues.set(index, (float) animation.getAnimatedValue());
                invalidate();
            }
        });

        // start each vertex at 50%
        mValues.set(i, .5f);

        anim.start();
        mAnimations.add(anim);
    }
}
```

In order for the wearable to get the results of a given module from the handheld, it retrieves a list of results from the data store in the same way as for the modules. The path for the results is `"/results/get/<module-id>"` where module-id is the id of the module to get the results of. When a new result is added on the handheld application, the result is stored in the database using Sugar ORM. The application then reads all results for that module into a `List` and stores that list in the data layer with the path `"/results/get/3"` where 3 would be the id of the module. The wearable can then access these lists of results by calling `getDataItems()` and filtering

on the same path.

As mentioned previously if the dataset is large, the entire list should not be synced, just the items that have changed and should be stored on both sides of connection.

4.7.5 News

The news screen first scrapes the UCC news page [4] by downloading it, parsing it using the `Jsoup`[5] library to find the news article titles and images. It then displays the articles in a list using the new `RecyclerView` which was introduced to Android in Lollipop and is designed as a more flexible and versatile `ListView`. It implements the `View Holder` pattern, making it more efficient by recycling (hence the name) views rather than inflating a new view from XML every time a new item appears on screen. The `RecyclerView` also has the added benefit of using `LayoutManager`s to decide where and how items should appear. This makes it easy to switch from single column lists to multi-column lists or grids.

Each article appears as an image with a title on the news screen. When one of the articles is clicked, a new screen is shown with the article's contents. To achieve this, a webview is populated with the contents of the article. The app downloads the articles web page, and parses the HTML for the `#content` container in the webpage. This way the UCC website itself is not shown, just the contents of the article.

Android provides many ways of downloading content using Http requests. This project uses the open source Volley[6] library. This library makes downloading things very easy, without having to worry about closing connections, or configuration changes causing the Activity to restart. Since Android does not allow network activity on the main thread, network requests must be made on a background thread. For this reason many applications might use an `AsyncTask` to download content, but this may cause problems when the user rotates the phone, restarting the Activity and causing the `AsyncTask` to get 'lost'. This could be fixed by wrapping the `AsyncTask` in a `Fragment` and calling `setRetainInstance(true)` on the fragment to prevent it from being restarted by the containing Activity, but Volley can handle all of this for us.

[4]UCC News page: http://www.ucc.ie/en/about/uccnews/
[5]Jsoup library: http://jsoup.org
[6]Volley library: http://developer.android.com/training/volley/index.html

Downloading the UCC news page looks as follows:

```
public static final String BASE_URL = "http://www.ucc.ie";
public static final String ALL_ARTICLES_URL = BASE_URL + "/en/news/";

...

mArticleRequest = MyVolley.getInstance(mContext).add(new StringRequest(
        ALL_ARTICLES_URL, new Response.Listener<String>() {
    @Override
    public void onResponse(String response) {
        // this will get called when we get a successful response
        parseResponse(response);
    }
}, new Response.ErrorListener() {
    @Override
    public void onErrorResponse(VolleyError error) {
        // this gets called when an error is returned from Volley or
        // the server
        onError();
        Log.e("E", "Error downloading: " + ALL_ARTICLES_URL);
    }
}));
```

`MyVolley` is just a simple wrapper for a Volley `RequestQueue`. It is a singleton class that returns a single `RequestQueue` when `getInstance()` is called. `RequestQueue` objects are used to queue downloads using Volley.
Parsing the articles from the returned HTML response looks like this:

```
ArrayList<Article> articles = new ArrayList<>();

// parse the document using Jsoup
Document doc = Jsoup.parse(response);

// using CSS selectors we can extract an array of the articles in HTML
Elements articleElements = doc.select(".article");

// loop through each HTML article and create a Java object and add it
// to the List
```

```
for (Element element : articleElements) {
    Article article = createArticle(element);
    if (article != null) {
        articles.add(article);
    }
}
```

This screen was designed in such a way so that any news source could be used to populate the list of articles. A custom interface called `ArticleFetcher` can be implemented and replace the `UccArticleFetcher` class to download the articles from a different source. For example, a college could replace it with a `ArticleFetcher` that fetched articles from an Rss feed. Or implement the code to scrape their own website and populate the screen with articles. The `ArticleFetcher` interface is defined below.

```
public interface ArticleFetcher {

    // callback listener for when the articles are ready
    public interface Listener {
        // called when the articles are downloaded and ready to be
           displayed
        public void onArticlesReady(List<Article> articles);
    }

    // ask the ArticleFetcher to start downloading the articles
    public void fetchArticles(Listener callback);

    // stop the downloading of the articles
    public void cancel();

}
```

Since the wearable is not able to download articles itself, and reading an article would be difficult on such a small screen, a service is scheduled to run in the background every 2 hours using the Android `AlarmManager`. This service runs the `UccArticleFetcher` and downloads the articles from the UCC news page. It checks each article title and stores the most recent article's title in `SharedPreferences`. If the previously stored article title is different to the one just downloaded, a notifica-

Figure 4.8: Open on phone action of notification

Figure 4.9: New article notification with background image of article

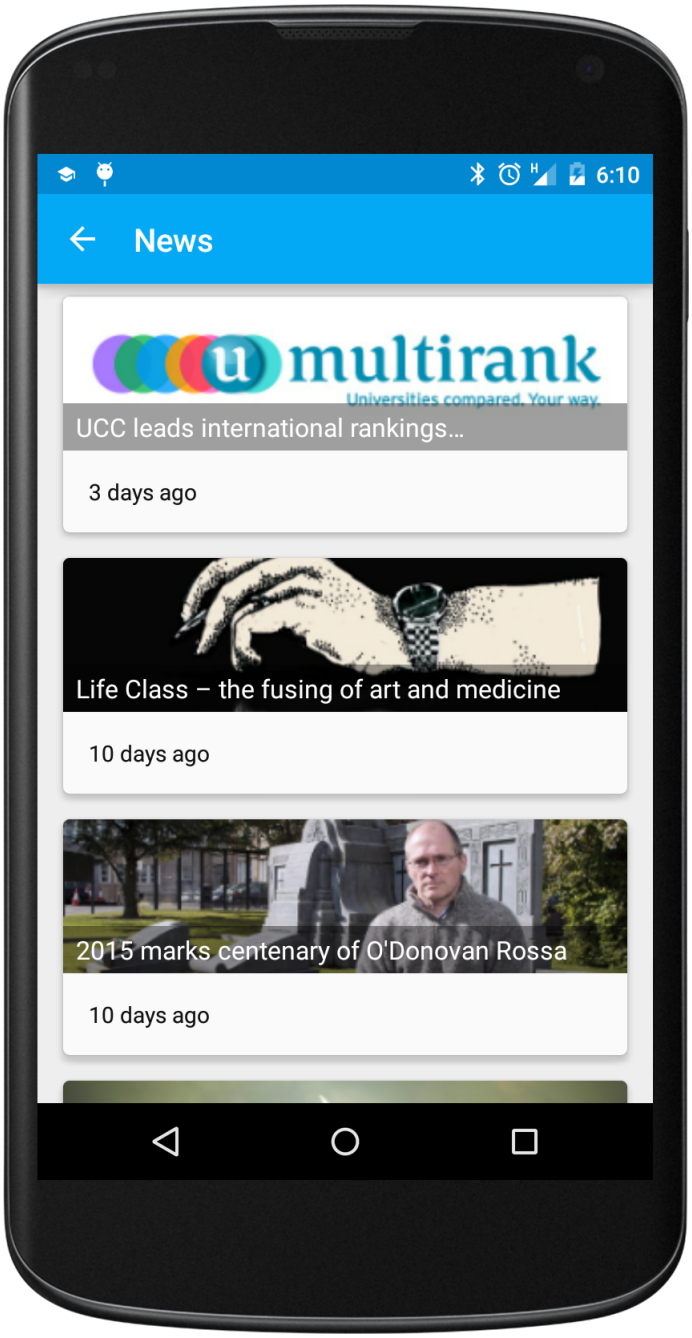

Figure 4.10: News screen with UCC articles

tion is shown to the user on both the handheld and wearable devices. The wearable notifications background is set to the image of the article (Figure 4.9) and a specific action (Figure 4.8) is shown to the user which when clicked (or touched) it will open the particular article on the user's handheld device.

4.8 Campus Map

This screen is only available on the wearable and was added in case the student gets lost on campus and needs a quick glance at the map to get their bearings back. The screen contains a custom view `BitmapRegionView` which was created with the goal of displaying a region of a large bitmap which can be scrolled around to view the whole bitmap. It works by creating a `Rect` object as a source input from the bitmap, and moving this `Rect` object around the bitmap and drawing the region to the screen. The scrolling is done using a `GestureDetector` object and listening for the `onFling` method to be triggered. This view allows a large map image to be displayed at full quality on a small screen and allows the user to pan around the large image at will.

Since the wearable is not able to connect to the internet, it must ask the handheld device for the map image. It does so by using the `MessageApi` to send a message requesting the map image. When the handheld receives this request, it makes an API call to `http://maps.google.com/maps/api/staticmap` endpoint. This endpoint allows us to download a static image of a location by providing a latitude and longitude of the location as parameters. Zoom and map size can also be specified for more control over the downloaded image. This image is then synced to the wearable using the `DataApi`. The wearable app then displays this large image using the custom `BitmapRegionView` implemented for this purpose.

Originally this screen used a `GridViewPager` to display the map bitmap. This is a wearable specific view that gives us a grid of pages which can be scrolled through both horizontally and vertically. Each page was displaying a chunk of the bitmap, but this view, since designed for wearables, behaved in an unfavourable way when scrolling vertically. In a grid, one might expect that when scrolling vertically, the column number would stay the same and the row number would change by 1. However with a `GridViewPager` the column number resets to 0 when scrolling vertically, so it does not function exactly how one might think a grid would function. For this reason the custom view `BitmapRegionView` was created.

Since this Activity uses a `GestureListener` to detect swipes, the system default of swiping from the left to go back to the previous screen was negatively affecting swiping to see different parts of the map. Since this is the default behaviour, it should only be overridden if unavoidable. Thankfully, Android Wear has provided a second alternative way of closing a screen if the back-gesture is not an option, and that is a long-press anywhere on the screen. In order to implement this, the back-gesture must

be disabled and the long-press must be listened for in the Activity instead. In order to disable the back-gesture, a rule must be secified in the style of that Activity.

`AndroidManifest.xml`

```
<application ...>

    <!-- set a style on the Activity -->
    <activity
            android:name=".activities.MapActivity"
            android:theme="@style/NoSwipeBack" />

</application>
```

`styles.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="NoSwipeBack" parent="Theme.Wearable">
        <!-- disable the back-gesture -->
        <item name="android:windowSwipeToDismiss">false</item>
    </style>
</resources>
```

Now that the back-gesture is disabled, the user has no way of leaving the `MapActivity`. We must now implement the long-press gesture instead. Android Wear has a built in view that can be displayed when a long-press has occurred called `DismissOverlayView`. This view can display a message when the Activity is first launched instructing the user about the long-press to go back gesture. When the long-press occurs, it then dimisses the screen automatically. In order to set the `MapActivity` uo to use this view, we must create an instance of the view in the `onCreate` method and add the view to our layout XML file.

`activity_map.xml`

```
<android.support.wearable.view.DismissOverlayView
       android:id="@+id/dismiss_overlay"
       android:layout_height="match_parent"
```

```
        android:layout_width="match_parent"/>
```

MapActivity.java

```
// create the DismissOverlayView
mDismissOverlay = (DismissOverlayView) findViewById(R.id.dismiss_overlay);

// set the text that will be shown if the user has never been to
// this screen before
mDismissOverlay.setIntroText("Long press to exit");
mDismissOverlay.showIntroIfNecessary();


// add the long-press gesture to the BitmapRegionView
mBitmapRegionView.setLongClickListener(new View.OnLongClickListener() {
    @Override
    public boolean onLongClick(View v) {
        mDismissOverlay.show();
        return false;
    }
});
```

Now when the user first enters the screen, they will be presented with a pop-up describing how to exit the Activity. When they long-press the `BitmapRegionView`, they will be presented with another pop up with a large 'X' as shown in figure 4.12 which when clicked will perform the equivalent of the back-gesture we disabled.

Figure 4.11: Map region shown on watch

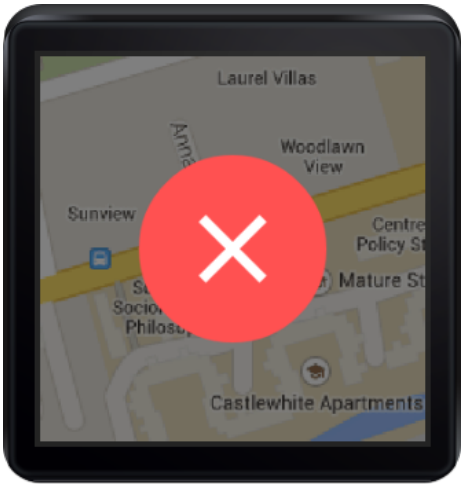

Figure 4.12: DismissOverlayView with long-press gesture

4.9 Games

As part of the student application, a few games were developed and designed to be short and simple to play. The following games were developed as part of the student appliction:
Wearable:

1. Lights Out
   A puzzle game where the user is presented with a grid of lights, some of which are in the on state. The goal is to turn off all the lights by clicking them, but when turning off a light, that light's neighbours are turned on.

2. Minesweeper
   A puzzle game where the player must locate a given number of mines in a grid of squares. Squares that do not contain a mine display the number of adjacent squares which do contain mines. The goal is to flag all the mines by analyzing the numbers and not clicking any square with a mine.

3. Game Controller
   Not a game, but a controller that sends bluetooth commands for up, down, left and right. Used for the multiplayer game of snake on the handheld, but could be used by any Bluetooth game that implements the protocol.

Handheld:

1. Multiplayer Snake
   A game of Snake similar to the snake game preloaded on Nokia phone in 1998 with Bluetooth multiplayer where each player controls a different snake and the aim is to be the last snake that hasn't crashed.

4.9.1 Lights Out

Lights Out [7] (Figure 4.13) is a puzzle game where the goal is to turn off all lights on the screen. It is a grid based game and so was a perfect fit for the square screen of the watch. However, on a circular watch screen the game is un-playable as the squares in the corners are truncated off screen and so cannot be clicked.

The game was implemented using a Model View Controller (MVC) approach. The

[7] Lights Out Wikipedia: http://en.wikipedia.org/wiki/Lights_Out_%28game%29

model is a simple `Tile` class consisting of the tile's position and state. The view is a canvas-based custom view that displays the grid and the tiles contained within the grid. And finally the controller is the `LightsOutController` class which controls the access to the models, and translates input into actions on the view and state of the game.

This MVC pattern allows us to swap out any part for a different implementation. The low coupling between the controller and the view allows us to change the view from being canvas-based to an XML layout of buttons and images for example with little to no code changes.

The game controller keeps track of an array of tiles which form the grid. Each tile has a position and state: on or off. When starting the game a level can be selected from the menu. Levels are stored as strings in the form "OOXOO...". These strings are processed character by character moving over the grid square by square. The first character of the level string is applied to the first square in the grid, the second character on the second square etc.
An "O" character means skip this square, a "X" means click this square. The controller processes each character until the game has been initialized with some lights, which the player has to reverse. The benefit of creating levels in such a way, is that we know the optimal number of moves needed in order to complete the level, that is, reverse the moves performed by the initialization of the level string. Using this optimal number of moves we can then keep track of the number of moves the player makes and calculate a score based on some algorithm.

Due to the small size of the wearable screen, we are limited in the number of tiles we can display in the grid whilst still keeping the tiles large enough for the player to click without accidentally clicking the wrong tile. As a result levels are limited in complexity and size to about 5x5 grids.

In the implementation of the game, the Activity is used as a middle man between the controller and the view. The Activity is responsible for taking input from the user and passing the information to the controller. The Activity is also a registered listener of the controller and can update the view using the defined callback methods specified by the `LightsOutController$Listener` interface shown below. In this way, the Activity acts as the glue between the view and the controller.

```
// interface defined within the LightsOutController class
public interface Listener {
    // called when a tile was clicked
```

```
    public void onTileToggled(Tile tile);

    // called when all lights have been turned off
    public void onGameOver(int numMoves);
}
```

When a level is complete, the score is calculated and a success view is shown to the user displaying their score. This view is a `ConfirmationActivity` (Figure 4.15) which is a built-in Android Wear Activity designed for showing the user a quick message that dismisses itself quickly. It is the recommended way of showing the user quick information that an action has been complete, without distracting them too much from their current task.

4.9.2 Minesweeper

Minesweeper[8] (Figure 4.14) is a puzzle game where the objective is to clean a mine field by flagging the mines. It is a grid based game where a given amount of mines are scattered across the grid randomly and it is up to the player to click on all squares apart from the ones containing mines. To help the player achieve this, squares that do not contain mines display how many neighbouring squares contain mines when clicked. Using these numbers, the player must decide which squares to click and which squares to flag as mines. The game is over when the player clears all non-mine squares or clicks a square containing a mine, in which case the game is lost.

The game is built similarly to Lights Out in that it is grid based and contains tiles. It also uses the MVC pattern explained above, but in this case an Android layout is used instead of a canvas-based custom view. This layout contains a single root `LinearLayout` element which is populated by other `LinearLayouts` at runtime, depending on how many rows are specified by the controller. These `LinearLayouts` are then populated with custom buttons which extend the android `Button` class depending on how many columns the controller has specified. This allows the layout to be just as dynamic as the canvas-based view used for Lights Out, but with the added benefit of being able to handle input/clicks for each individual button in the grid, rather than having to figure out where the player clicked and decide which button was hit.

`activity_grid.xml`:

[8] Minesweeper Wikipedia: http://en.wikipedia.org/wiki/Minesweeper_%28video_game%29

```xml
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/grid_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

</LinearLayout>
```

It does however increase complexity, and adding views at runtime is more error-prone than creating the views at compile time in XML where a GUI can be used to preview what the layout will eventually look like. For describing how views should be layed out programmatically in Android, `LayoutParams` must be used and can be cumbersome and verbose. The rows and buttons are added to the grid as follows:

```java
// setup the minesweeper grid view
// here rootView is the root LinearLayout
private void setupView(LinearLayout rootView) {

    // the layout params for the buttons
    LinearLayout.LayoutParams buttonParams = new LinearLayout.LayoutParams(
            LinearLayout.LayoutParams.MATCH_PARENT,
            LinearLayout.LayoutParams.MATCH_PARENT, 1.0f);

    // add a new view for each row
    for (int y = 0; y < mRows; y++) {

        // layout params for the row views
        LinearLayout.LayoutParams rowParams = new
            LinearLayout.LayoutParams(
                LinearLayout.LayoutParams.MATCH_PARENT,
                LinearLayout.LayoutParams.MATCH_PARENT, 1.0f);

        // the row view itself
        LinearLayout rowLayout = new LinearLayout(getApplicationContext());
        rowLayout.setOrientation(LinearLayout.HORIZONTAL);
```

```
        rowLayout.setLayoutParams(rowParams);

        // create the buttons for each row
        for (int x = 0; x < mCols; x++) {

            // create the button
            GridButton button = createButton(x, y);
            button.setOnClickListener(mButtonClickListener);
            button.setLayoutParams(buttonParams);

            // callback for the controller
            onButtonCreated(button);

            // add the button to the row view
            mGrid[x][y] = button;
            rowLayout.addView(button);
        }

        // add the row to the grid
        rootView.addView(rowLayout);
    }
}
```

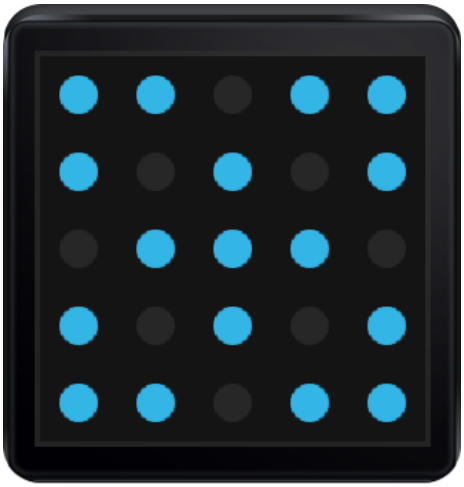

Figure 4.13: Lights Out game

Figure 4.14: Minesweeper

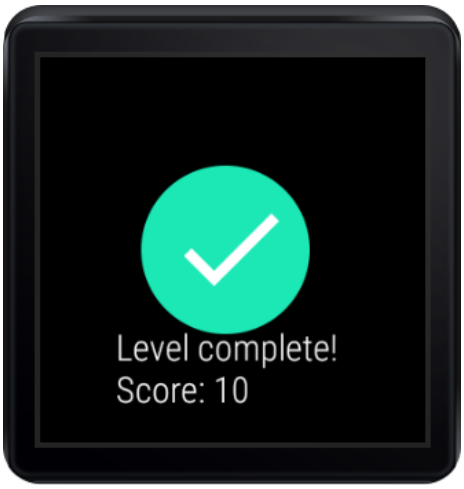

Figure 4.15: Success Screen

4.9.3 Multiplayer Snake

The final game developed for this project was a game of Multiplayer Snake[9]. The game is played over Bluetooth with one handheld acting as the game's screen while the wearables/other handhelds act as the controllers. The game can be played with 1-7 players as Bluetooth has a limit on how many devices can be connected at one time. This limit, however can be raised to 255 by using a Piconet[10].

A Piconet consists of one master device, the screen/handheld device in this case and up to seven slave devices, the controllers in this case. To raise the seven slave device limit, slaves can be set in "parked" mode where they are inactive but still part of the Piconet. We could swap devices in and out of parked mode to give the appearance that all devices are active, when in actual fact only seven are active at any one time. As a result we could connect up to 255 slave or controller devices. However, having 255 players in a game of Snake would not be playable!

The objective of this version of Snake, is to be the last player alive. Each player controls a different snake, and must eat the food (a red square) in order to make their snake longer. A snake dies when it's head collides with it's own or another snakes body. Longer snakes can try and trap other snakes by boxing them in. The winner is the last snake that has not crashed.

This game uses raw Bluetooth sockets as Android Wear does not allow multiple wearables to connect to a single handheld device. The connection is one-to-one which means that one handheld cannot have multiple wearables and vice-versa. Due to this limitation, raw Bluetooth sockets were used. Google advises against doing this, but as a multiplayer game cannot be multiplayer with only one controller device, a workaround had to be implemented.

As a result of not using Android Wear APIs for the game, a protocol had to be invented for the controllers to communicate with the master device. The master device waits for controllers to connect. When a controller wants to connect, it uses the pre-defined `uuid` of the master device to create a `BluetoothSocket` for the master. The master creates a separate thread for each controller it connects to. Each thread has a unique identifier which it uses to differentiate which player sent which message.

[9]Snake Wikipedia: http://en.wikipedia.org/wiki/Snake_%28video_game%29
[10]http://en.wikipedia.org/wiki/Piconet

Bluetooth Controller Protocol

The controllers can send the following messages on their `BluetoothSockets` to have the following effects on that player's snake.

```
"left"     => move the snake left
"right"    => move the snake right
"up"       => move the snake up
"down"     => move the snake down
```

4.10 Custom Watchface

The final part of the project was the implementation of a custom watchface using the official Android Wear APIs. At the beginning of the project, these APIs were not finalized nor made publicly available, so an attempt was made to use whatever made the watchface work. A decent attempt was made, but did not follow the guidelines which were later released, and so the code was updated to follow these guidelines[11] and use the officially released APIs instead.

4.10.1 Overview

A custom watchface, is a service that runs constantly on the wearable. It is the equivalent of the homescreen on a handheld device as it is the first thing the user sees when the device is woken. A watchface has the added benefit of also being able to display things on screen when the device is not woken, but in a low-power mode so the user can constantly see the time. This mode is called "Ambient Mode" and is triggered when the user is no longer interacting with the watch. In Ambient mode, the watchface should be wary of saving power. As such, it should only update the display when necessary and should avoid displaying too much colour or white pixels in order to avoid pixel burn-in.

A watchface cannot handle input as it is the screen used by the system to carry out system-default functions, such as launching an app or pulling down the quick

[11] ttps://developer.android.com/design/wear/watchfaces.html

settings toggles. Overriding these functions would be discouraged and confusing for the user.

Watchfaces use two extra permissions, which must be declared in the `AndroidManifest.xml` file. These permissions are used because the watchface needs to be able to run in the background and keep the processor active to execute it's code.

```
<manifest ...>

    <!-- watchface permission -->
    <uses-permission
        android:name="com.google.android.permission.PROVIDE_BACKGROUND" />

    <!-- permission to wake the processor when device is not active -->
    <uses-permission
        android:name="android.permission.WAKE_LOCK" />
    ...
</manifest>
```

4.10.2 Watchface API

Android Wear provides a Service class which can be extended to create a watchface. This class is a very barebones implementation, and the programmer is left to implement much of the boilerplate code. An easier class to extend is the `CanvasWatchFaceService` which is also provided by Android Wear, but gives the programmer access to a `Canvas` object which can be drawn to for easy displaying of graphics. Other wise OpenGL would have to be used, which is a more powerful graphics API than canvas, but also has a much steeper learning curve as it is much lower level.

The `CanvasWatchFaceService` looks as follows:

```
public class AnalogWatchFaceService extends CanvasWatchFaceService {

    @Override
    public Engine onCreateEngine() {
        // the Engine is the class that will handle all watch-related tasks
        // and will also handle rendering to the screen
```

```
    return new Engine();
}

/* implement service callback methods */
private class Engine extends CanvasWatchFaceService.Engine {

    @Override
    public void onCreate(SurfaceHolder holder) {
        super.onCreate(holder);
        // called when the Engine is created
    }

    @Override
    public void onPropertiesChanged(Bundle properties) {
        super.onPropertiesChanged(properties);
        // called when the properties of the watchface have been altered
    }

    @Override
    public void onTimeTick() {
        super.onTimeTick();
        // called every minute
    }

    @Override
    public void onAmbientModeChanged(boolean inAmbientMode) {
        super.onAmbientModeChanged(inAmbientMode);
        // called when the device goes in or out of Ambient Mode
    }

    @Override
    public void onDraw(Canvas canvas, Rect bounds) {
        // called when the Engine is invalidated
    }

    @Override
    public void onVisibilityChanged(boolean visible) {
        super.onVisibilityChanged(visible);
        // called when the watchface is no longer visible, ie. when the
        // user is in an app or the watchface is no longer in the
        // foreground
```

```
        }
    }
}
```

As you can see above, the system calls the `onTick` method every minute. This may be enough when the device is in Ambient mode, but in order to show the seconds hand on the clock, we will want to update the display every second while not in Ambient mode. This must be implemented by the programmer, and this is how it is done in the Calendar Watchface for this project:

```
// delay in milliseconds
private static long TICK_DELAY = 1000;

private Handler mTimeTick;

// this task will be run every second
private final Runnable mSecondTick = new Runnable() {
    @Override
    public void run() {
        onSecondTick();
        if (isVisible() && !isInAmbientMode()) {
            mTimeTick.postDelayed(this, TICK_DELAY);
        }
    }
};

...

private void startTimer() {
    // cancel the current timer
    mTimeTick.removeCallbacks(mSecondTick);
    if (isVisible() && !isInAmbientMode()) {
        mTimeTick.post(mSecondTick);
    }
}

@Override
public void onCreate() {
```

```
    // start a task on the UI thread
    mTimeTick = new Handler(Looper.myLooper());
    startTimer();

}

public void onAmbientModeChanged(boolean inAmbientMode) {
    // reset the timer
    startTimer();
}

public void onSecondTick() {
    // code to be run every second
    if (isVisible() && !isInAmbientMode()) {
        invalidate();
    }
}
```

Here we use a `Handler` object to post `Runnable` objects to the UI thread. In this case we are executing this runnable every second and refreshing the display if the watchface is visible and not in Ambient mode.

4.10.3 Drawing on the Canvas

Drawing on the canvas is the same as drawing on a canvas for a custom view in Android. `Paint` objects are used for stroking shapes in different styles and colours. Detecting whether the screen is round or square is slightly trickier than in a regular Android Wear app. In a normal app, a `WatchViewStub` stub can be used to automatically apply a given layout depending on the shape of the display as shown below:

```
<android.support.wearable.view.WatchViewStub
    ...
    app:rectLayout="@layout/rect_activity_my_wear"
    app:roundLayout="@layout/round_activity_my_wear"
    ...
>
```

However, since the watchface is a service, it cannot inflate views to display on screen. Instead, it must be done programatically. In order to achieve this, a custom `WindowInsetsListener` must be created. This listener will be given enough information to deterine the shape of the screen. This can be useful for the application to know, so it can draw things on a particular region of the canvas to ensure that it is visible on all screen shapes. A custom listener is shown below:

```
view.setOnApplyWindowInsetsListener(new
    View.OnApplyWindowInsetsListener() {
    @Override
    public WindowInsets onApplyWindowInsets(View v, WindowInsets insets) {
        if (insets.isRound()) {
            // ROUND SCREEN
        } else {
            // SQUARE SCREEN
        }
        return insets;
    }
});
```

In order to draw text and shapes at certain locations on the watch, we must use trigonometry to draw them at the correct locations on the canvas. For example, in order to draw the numbers of a clock along the outside of the clock, sin and cos must be used as follows:

```
// incase we wanted a 24 hour clock
int numNumbers = 12;

mLength = // the radius of the clock

// start at 1 to have numNumbers at the top instead of 0
for (int i = 1; i <= numNumbers; i++) {
    float x, y;
    x = (float) Math.sin(Math.PI * 2 * (i / (float) numNumbers)) * mLength;
    y = - (float) Math.cos(Math.PI * 2 * (i / (float) numNumbers)) *
        mLength;
    canvas.drawText(String.format("%d", i),
        mCenter.x + x, mCenter.y + y,
        mPaint);
}
```

4.10.4 Calendar Events

The custom watchface developed for this project displays the user's calendar events along the circumference of the watchface as long coloured arcs (Figure 4.16). However the wearable device does not directly have access to the handheld's calendar. As a result there are two main ways to get the calendar events to the watch.

1. Manually
   Use the `MessageApi` to request the handheld to send the calendar events over to the wearable using the `DataApi` to store the objects.
2. WearableCalendarContract Use Android Wear's built in `WearableCalendarContract` to query the calendar directly from the wearable. This gives the programmer access to the `Cursor` instance with the results of the query.

Combining the features described above, the custom watchface retrieves the user's events from the handheld, wraps them into objects and displays them on the canvas using the standard Canvas APIs. Figure 4.16 depicts the final outcome.

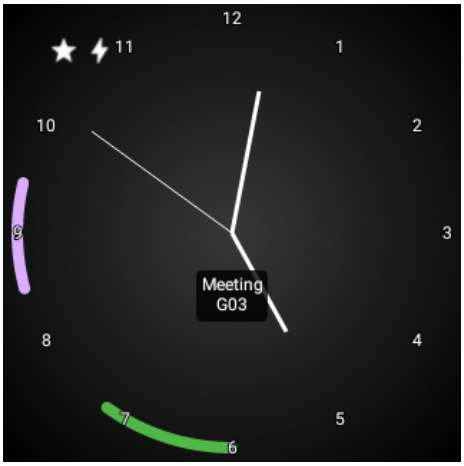

Figure 4.16: The custom Watchface with the events as coloured arcs

4.10.5 Calendar WatchFace

An extension of the watchface developed for this project was published to the Google Play Store[12]. This app is built on the same source code as the watchface for this project, but also adds many features such as a settings application for mobile that allows the users to customize different settings for the watchface such as the thickness of the hands and the background image for the clock face. At the time of writing the watchface has been purchased by 18 people and has a rating of 4.3 in the Play Store.

[12]Calendar WatchFace: https://play.google.com/store/apps/details?id=com.aidangrabe.customwatchface

Chapter 5

Testing

In this chapter, the testing performed on the Student Application and Calendar Watchface will be discussed. Testing was performed throughout the development of the project and no new features were added in the last week before completion, just testing and error/bug fixing.

5.1 Overview

The following testing was performed on the project:

1. Unit Testing
   Individual units of code tested with as many inputs as possible to ensure they work as expected.
2. Performance Testing
   Testing the performance of Android Wear and different methods of communication to see which methods were more efficient and more responsive.
3. Usability Testing
   Day to day testing as well as testing by third parties just by using the apps as regular users and logging or reporting bugs when/if they occurred.

5.2 Unit Testing

Since the source code for the project is written in Java, JUnit was used and easily integrated into the project's workflow. Test cases for individual classes were written, trying different inputs and edge cases or unlikely cases were tested to ensure the class could handle the input.

Unit tests can only really be performed on strictly Java code, however testing can be performed on Android views and state using Android's built-in testing framework. This framework allows you to test whether views are visible, input was received and things a unit test might not be able to test.

Android unit testing is made difficult because a DVM (Dalvik Virtual Machine) is required to execute Android code, or code that refers to the Android SDK. This code cannot be executed on the regular JVM (Java Virtual Machine). The JVM requires .class files to execute, but Android code is stored in .dex files for the DVM. The only way JUnit can be used to unit test Android code, is if the business logic is separated from the UI logic. This way the business logic can be saved as regular Java in .class files and tested on the JVM.

Third-party testing frameworks also exist which do allow proper unit testing. One such framework called Robolectric[1] was used for unit testing Android specific code. Robolectric allows Android code to be run and tested without deploying the Android emulators. Normally in order to use or test code that uses the Android SDK, an emulator would have to be instatiated and the code would have to wait until the emulator is ready to start execution of the code. This is less than ideal for automated testing. Robolectric allows the tests to run directly in the IDE without the need of the emulator. It achieves this by implementing standard JVM code for the Android SDK classes. This allows them to be run and used on the regular JVM as opposed to the DVM. Starting a JVM on a workstation is much quicker than booting up an emulator and starting a DVM on it.

Other frameworks such as Mockito[2] or EasyMock[3] work by mocking or stubbing-out different classes. Data structures and databases can be mocked to give the same effect as a normal data structure or database but without the underlying complexity behind them. To use an analogy, a mock object simulates the required behaviour of

[1]Robolectric: http://robolectric.org/
[2]Mockito: http://site.mockito.org/
[3]EasyMock: http://easymock.org/

an object for a specific test the same way a crash-test dummy simulates a human body in a crash test for a vehicle.

5.3 Performance Testing

The performance of the Student Application is not too critical as the application does not undertake any heavy duty or long processing tasks. The only area which may cause the user to notice the performance of the application is the communication between the handheld and wearable device. If the communication gets blocked or hangs, then the user will be stuck with a blank screen instead of the task they were meant to see.

In order to test the performance of the communications, tests were set up to explore which type of communication was the quickest and most efficient. There are three main ways to communicate between the handheld and wearable devices, and they are:

1. MessageApi
2. DataApi
3. Raw Bluetooth

The outcome of the tests were that Raw Bluetooth was the quickest way to get data from A to B as it has no overhead attached. However, since it is not handled by the system, all the protocol details are left to the programmer which means it can be muc more error-prone than using the Android Wear APIs.

The MessageApi was the next most efficient form of communication, but has the drawback that the payload is limited in comparison to using Assets synced in the DataApi. It's faster because it has less overhead than the DataApi, however it is less reliable than the DataApi as messages are not guaranteed to reach their target. Say for example, that the wearable was out of range of the handheld, and a message was sent to the handheld. The message would probably not reach the handheld and error checking and handling would need to done by the programmer to ensure that no data is lost and the message get reset later if neccessary.

The DataApi was the slowest method of communication, but also the most reliable. It also allows payloads of over 100KB to be transferred by attaching Assets to DataItems. This meant that it might actually be more efficient for large transfers

rather than trying to chunk an array of bytes and send them via Bluetooth or the MessageApi manually.

As a result of these tests, different tasks could change their method of communication to fit these results. Lightweight communication could be left to the MessageApi and larger, heavier communication would be handled by the DataApi.

5.4 Usability Testing

Usability testing was performed by using the application on a day to day basis. Using the application as a normal user might, and loggin bugs as they appeared. The project is hosted on Github, which also has a built-in issue tracker. When a bug appeared during regular use, it would be logged on paper if not near an internet-facing device, or logged directly into the Github issue tracker where it could be assigned a category and tag and where it's state would be logged so it would not be forgotten.

Testing was also performed by third parties such as friends and family. They would follow the same reporting process by logging issues in the tracker, or by keeping note of the issues themselves, and passing them on when appropriate.

Due to the nature of a watchface being always visible to the user, it was easy for people to test as it was always in the foreground, and if something went wrong, it was immediately noticeable as the watchface might restart or stop altogether, so the next time the user checked the time, they'd know there was a problem and could forward on the logs and the stack-trace of the crash.

Chapter 6

Conclusion

In this chapter, the accomplishments of the project will be reviewed and discussed. It will also discuss the potential for future developments.

6.1 Overview

This project met all of its original requirements and provides a detailed view of the brand new technology that is Android Wear. At the time of writing there were few quality, extensive, detailed guides on using Android Wear. This dissertation aims to provide such a guide by explaining the use of the APIs and demonstrating them by providing a sample application that students may also find useful.

The project offers the following features:

1. A detailed description of the Android Wear communication APIs
2. Examples of how to create notifications that work on both a handheld device but also on a wearable.
3. Examples of how to poll sensors and use their data for displaying to a user or performing calculations based on their values.
4. Performing actions based on 3d gestures performed by the user.
5. A useable application that helps students stay on top of their studies by helping them track their lectures, result progress, college news and track their tasks.

6. A few wearable games that were designed with a small screen in mind.
7. An example of how multiple wearables can be used together to create a multi-player game of Snake.
8. A watchface that can show the user their events for the day at a quick glance.
9. An example of how to build a custom watchface which adheres to the Google watchface guidelines and that works on both circular and rectangular watches.

The project demonstrated the use of many different tools and technologies:

1. the applications were written using Java and the Android and Android Wear APIs.
2. Android Studio was used for the IDE.
3. Git was used for version control.
4. Gradle was used as the build system.
5. Bluetooth Low Energy is the underlying technology that makes the communication between handheld and wearable possible.
6. Bluetooth sockets were used to work around the one-to-one device restriction of Android Wear.
7. AppCompat libraries were used to ensure compatibility with older versions of Android.
8. Android L specific features were used such as colouring the status bar.

6.2 Android Wear Conclusion

Android Wear as a platform is still very new and developers have yet to find a really useful function that a smartphone could not perform on its own. Gestures seem to be one of the main benefits of a smartwatch, but the APIs have no built-in support for detecting them.

Due to the small screen size of the smartwatches, and the fact that only one hand can ever be used to touch the screen, navigating menus in applications can be awkward and since only so much text can be displayed on the small screen, a user is usually better off reaching into their pocket and launching the mobile application instead. A

smartwatch is only useful for glanceable information such as notifications and custom watchfaces which display useful information all the time.

In it's current form Android Wear does not make launching apps easy or intuitive. At the time of writing, in order to launch an app the user must touch the screen, scroll to the "Start..." menu item, and then scroll to the desired app and touch it to launch. It would be just as quick to launch the app on a phone, thus defeating the purpose of the smartwatch.
Apps can also be launched by voice, but speech recognition can be awkward in crowded places and inaccurate when background noise is present, or if the user has an unusual accent.

Current generation devices also have very short battery life. Devices usually only last a single day per charge, depending on usage. This is a big change for someone who wears a regular watch on a day to day basis which would not need to be charged for months or years. As mentioned in a previous section, the Pebble watch uses an e-display allowing it to last for much longer on a single charge. Perhaps Android Wear could support this display technology in the future in order to compete with Pebble's battery life.

6.3 Future of Android Wear

At the time of writing, Android Wear was a brand new technology in it's infancy. When development of the project began, Android Wear was at version 4.4W and over the space of a few months has already reached version 5.0.2. As a result many changes will be made to the APIs in a short space of time. Here are some changes that one might expect in future releases:

1. Support for multiple wearable devices to connect to a single phone
2. Support for communicating with the new iteration of Google Glass
3. Support for other Google technologies such as Android TV and Android Auto.
4. WiFi built in to the wearables. Currently many smartwatches use a Snapdragon 400 series as their SoC (System on Chip), many of which support wireless technologies such as Wi-Fi. At the time of writing, Android Wear does not support access to this technology, but will be implementing it in a later iteration. This means that much of the code in this project that accesses internet resources

could be re-written to download the resource directly to the wearable instead of using the handheld as a proxy.

5. Support for other wearables such as smart-shoes, not just smartwatches.
6. Gesture based navigation. As it is, a touchscreen requires the use of two hands to interact with the smartwatch, gesture-based navigation would allow single handed use of the device, such as tilting your wrist to scroll or shaking the device to return to the homescreen etc.

6.4 Future Work

This section suggests possible improvements or additional work that could be undertaken on the Student Application. This work was not completed in the original project due to time constraints.

1. the timetable shown on mobile could have a screen on the wearable showing the user's upcoming lectures.
2. a custom watchface could also be created specifically for the Student Application showing the user's upcoming lectures and outstanding to-do items so the user can see them at a quick glance at their watch.
3. Android Wear will probably add a lot more features in the future, so these features could be displayed or demonstrated in the app.
4. More games could be added to the student application. Currently there is only one game for the handheld device, and users may want a couple more.
5. the Student Application could be extended to support devices without wearables. Currently the application is only available to devices using Android 4.3+. This is because it is Android Wear's minimum requirement. However many of the features of the handheld application do not require a wearable, and so the minimum version could be lowered and wearable-specific content could be hidden to those users.
6. Other wearable devices may be announced and supported by Android Wear, such as smart-shoes, smart-necklace etc. These devices may run Android Wear and as a result may be integrated with the current applications. For example students could track their walking efficiency to college using smart-shoes.