Final Year Project Report

A Comparison of Android and iOS application development

Máire Regan

A thesis submitted in part fulfilment of the degree of

MSc (hons) in Advanced Software Engineering

Supervisor: Mel Ó Cinnéide

UCD School of Computer Science and Informatics
College of Engineering Mathematical and Physical Sciences
University College Dublin

December 22, 2011

Table of Contents

Abstract

The smartphone market has experienced huge growth over the last couple of years, and with this has come a corresponding increase in the number of mobile applications downloaded. With the introduction of developer programs, mobile software development kits and dedicated application markets, this increasingly important area is accessible to any developer who is interested and willing to learn.

Mobile application development differs significantly from traditional software development so different user interface guidelines, application lifecycles and performance considerations must be taken into account. There are a large number of both hardware and software vendors in the smartphone market, and the developer must decide which platforms, tools and devices best suit their needs.

This thesis aims make a practical comparison of Android and iOS development, starting with the basic hardware and software required, the tools provided and the languages used for development. It then continues onto topics covering design and implementation of mobile applications, leading up to a discussion on the distribution of the completed application. All this is examined from the point of view of a developer new to Android and iOS application development.

Acknowledgments

I would like to thank my supervisor Dr Mel Ó Cinnéide for all his guidance and support over the course of the project. Thanks also to Karl Connon for his work on the back end web service that provides the data for the LiveBus application.

Thanks to my family for their patience and encouragement, not only over the last two years, but over the whole course of my education. Thanks especially to Mick for his endless patience, love and support.

Chapter 1: **Introduction**

Research by Gartner shows that smartphones accounted for 25% of overall mobile device sales in the second quarter of 2011, an increase of 74% on the previous year [23]. With this explosion in popularity, more and more developers are turning to mobile application development. There are several different mobile operating systems to choose from, however, this report will focus on the current market leaders: Apple's iOS and Google's Android. According to a recent study of new smartphone purchases in the USA by Nielsen, purchases of Android phones stood at 56% of the smartphone market while iOS phones were in second place at 28% [30]. Between them, these two operating systems hold 84% of the market, and if current trends continue this is set to rise even further. This figure does not include tablet devices, or other devices such as the iPod touch which does not have mobile phone functionality. Varying figures can be seen in the many different reports into this area, but the overriding trend is that Android and iOS are the market leaders.

This chapter will briefly discuss the origins and history of iOS and Android application development, set out the topics covered in this report and introduce the LiveBus mobile applications which were developed for the purpose of investigating both platforms. In section 1.1 background information is provided, and culture is discussed with regards to both platforms. In section 1.2 information is provided on the resources available to a developer working in Android and iOS development. Section 1.3 gives detail on the application that was developed for both platforms while researching this report. Section 1.4 describes the topic covered in this report.

1.1 Background and Cultural Influences

Android is an open-source software stack created for mobile devices. It is an open-source project which contains an operating system, middleware and a set of core applications for mobile devices. Work on Android began at Android Inc., a private company, however, not much is known about its early development before it was bought by Google in 2005. Google then formed the Open Handset Alliance [41], a collection of hardware, software and telecommunications companies, with the aim of producing an open and free mobile platform. The Open Handset Alliance is led by Google and today has over 80 members from a wide variety of background industries. It is responsible for the distribution and maintenance of the Android Platform. The Android Open Source Project also attempts to keep in sync the conflicting aims of the hardware manufacturers, application developers, etc. that use the Android operating system through the Android Compatibility Program. This sets out standards that device manufacturers must meet in order to use the Android trademark, and also provides a set of tools to ensure that developer applications run on licensed devices.

The Android kernel is based on the Linux kernel and this code is made available under version 2 of the Free Software Foundation's General Public License (GPLv2). The majority of Android software, however, is developed under Apache 2.0 Open Source License [3]. This was chosen for the userspace (non-kernel) software as it allows for open source development, but does not force others using it to open up their software. This allows the use of Android code in products that are not open-source without restricting their distribution in any way.

Parts of Android are developed in private by Google and only made public once a new version of Android has been released. Other parts of Android, e.g. the Android Market, are kept completely private by Google and source is not available to third party developers for this. The Android platform is continually and rapidly evolving due to the work by the Open Handset Alliance, the open-source community and Google. Android 1.0 was released in late 2008 and only 5 years and 13 releases later, Android 4.0 or API level 14 has just been released.

The architecture of the Android platform is another feature which promotes openness and reuse. Android was designed to allow third party applications to use the device's core functionality through the same framework APIs as the Android core applications. Any application can access the phone's contacts, messaging, camera, global positioning service, etc. as long as it has the correct permissions. These are agreed by the user when the application is installed. Reuse is encouraged as any application can publish its capabilities, and these can then be used by other applications with the correct permissions. This mechanism also allows the device user to replace certain applications if they prefer others with the same capabilities.

Apple entered the smartphone market with the launch of the iPhone in 2007, having originally started in the 1970s as manufacturer of personal computers and operating systems. Apple was a pioneer in this field, being one of the first to introduce graphical user interfaces, however its fortunes had receded by the 1990s with the growing popularity of IBM PCs and the Windows operating system. Apple refocused on the Macintosh line, a more expensive computer aimed at professionals and colleges. The iMac line was released in the late 1990s and in 2001 Mac OS X was released. iTunes for the Mac was also released in 2001 and this was followed by the release of the iPod, Apple's breakthrough MP3 music player, towards the end of the same year [8].

Several new models of iPod were released in quick succession over the next few years, leading up to the initial release of the iPhone in 2007. The iPhone merged mobile phone, touch-screen iPod and internet communications into one device. The iPhone quickly earned a cult following for its aesthetically pleasing design and unprecedented functionality, with a million devices sold in just 74 days [11]. Apple's next move in this area was the launch of the iPhone Developer Program and the iPhone Software Development Kit. For the first time, third party developers could develop applications for the iPhone and sell these in the App Store, Apple's online marketplace for purchasing applications for the iPod and iPhone [7]. The iPhone 3G proved phenomenally popular with one million devices sold within 3 days of its launch in 2008 [10]. With the launch of the iPhone 3G came the public launch of the App Store which proved enormously popular with users; in its first weekend of launching over 10 million apps were downloaded [9]. Apple's successes have continued with the latest model, the iPhone 4S, selling over 4 million devices in its first weekend on sale [15], and App Store downloads passing 15 billion in July 2011 [12].

Apple provides a number of frameworks to provide applications with access to core functionality, e.g. address book, media player, mapping etc. One area where iPhone development differs substantially from Android development is that Apple does not allow developers to submit their own versions of native iPhone apps to the App Store. As this is the only way to distribute iPhone applications, this restricts the user in their choice of core functionality for their iPhone.

1.2 Resources

There are a huge amount of resources available to application developers for both Android and iPhone development. Both have official developer programs, Android Developers [2], and iOS Developer Program [14], along with a huge amount of third party forums, blogs, tutorials, etc.

The Android Developers website has links to download the latest software development kits and tools, developer guides on how to set these up, tutorials on application development, sample code and examples of using various parts of the framework, best practice guides, guides for publishing applications, javadoc for the APIs, videos, blogs and other resources. A number of Google Groups mailing lists exist for Android development, not to mention sites like Stack Overflow [50] where developers help each other solve problems with application development.

Similarly for iOS, once enrolled in the iOS Developer Program the software development kits is available for download along with a wealth of other material. Apple provides the iOS Developer Library which has documents to help get started with iOS application development, sample code, and coding how-to articles, framework reference documents, release notes and videos. Another useful resource is the Apple Developer Forums website where Apple engineers and other developers provide answers to development questions.

Many universities have started offering mobile application development modules as part of their computing courses, and a number of commercial organisations also provide training in this area. Also available are online courses on iTunes University, e.g. Stanford University provides the free course CS193P iPhone Application Development [27] which includes 8 weeks of recorded lectures and notes.

A number of application development books are also widely available, from familiar names such as O'Reilly Media and Wiley, to the not so well known. Two recent publications for learning iOS development are Head First iPhone and iPad Development [43] and iOS 4 Programming Cookbook [39]. Two comparable publications for Android development are Professional Android 2 Application Development [34] and Beginning Android 3 [38].

1.3 LiveBus Mobile Application

The LiveBus mobile application was developed for the purpose of comparing Android and iOS application development. LiveBus allows a user to track the position of Wexford Bus in real time. It is designed to complement the TextMeMyBus system [52], which allows the user to also register for text messages to inform them when a particular bus is due. Both systems use information provided by a web service from Imeall Software based on live data from Wexford Bus. On opening the application the user is presented with a list of bus stops. A map view is also available which shows the bus stops along with the current bus positions. More information for the bus stop may be viewed by tapping on the stop on either the list or map. A separate page shows the location of the bus stop in more detail, along with the next bus due time, and the scheduled pickup times for the stop. Favourite stops may also be recorded.

1.4 Report Layout

This report chronicles the entire process for developing a mobile application for Android and for iOS. This starts with research into the required hardware and software, followed by the differences in languages for developing in each platform. Applications for both Android and iOS are structured differently and have different lifecycles, so different patterns apply and different constructs must be used. Along with this, implementation details differ for various features found in mobile applications today. The final stage is the distribution of the finished applications and again there are stark differences between both Android and iOS. The following is a brief overview of the structure of this document:

Chapter 2 provides information on hardware and software requirements for developing Android and iOS applications.

Chapter 3 discusses the languages used for Android and iOS application development.

Chapter 4 looks at design considerations for Android and iOS applications.

Chapter 5 gives details on the implementation of some of the main features commonly used in mobile application development.

Chapter 6 provides information on the distribution of iOS and Android applications.

Chapter 7 discusses the merits and demerits of both platforms, along with some alternatives.

Chapter 2: **Getting Started with Development**

For the developer beginning mobile application development, it can be hard to know where to start. Specific hardware and software may be required in order to run the supported development environments, new tools must be used, and the developer may or may not have access to physical devices for testing.

This chapter aims to explore all these issues for Android and iOS development. Section 2.1 gives details on the platforms required for development. This is followed by section 2.2 which details the Integrated Development Environments available for both. Section 2.3 gives more detailed information on the tools available for the developer working on both Android and iOS applications, and section 2.4 provides information on the available devices for both Android and iOS. Section 2.5 provides a summary of the information contained in this chapter.

2.1 Platforms

iPhone development is essentially restricted to Mac OS X with no supported tools available for development on other operating systems. Android is much more flexible however, with tool support for Windows, Mac OS X and Linux along with a wide variety of development environments.

2.2 Integrated Development Environments

iOS development is only supported using the Xcode integrated development environment (IDE) and Interface Builder graphical user interface design tool. All the required development tools are available to download from the iOS Developer Program website or from the Mac App Store. Additional tools such as the Instruments analysis tool and the iOS Simulator are also provided, along with the latest version of the iOS Software Development Kit (SDK). More recently, JetBrains have released a new IDE called AppCode [6], which can open XCode projects and is compatible with Interface Builder and the iOS Simulator. However, this is not free and is not officially supported by Apple.

The use of the Eclipse IDE and the Android Development Tools (ADT) plugin is recommended for Android development. Eclipse is an open source project and so is available for free, and may be extended with a variety of plugins depending on the developer's needs. The Android SDK is a separate download to the Eclipse ADT plugin and must be downloaded before the ADT. The SDK contains only the core tools and can be used to download other required components such as the latest Android platform. Support also exists for the use of other IDEs such as IntelliJ from JetBrains [28]. However, the documentation for this is not maintained by the Android Open Source project and again the IDE is not free. If the developer would rather not use an IDE for development, the tools bundled with the Android

SDK may be run from the command line.

Both Xcode and Eclipse, and the platform SDKs can be downloaded for free and contain all the tools that a developer will need in the course of developing a mobile application. These tools are discussed in more detail in the following section.

2.3 Tools

A large selection of tools is available for both iOS and Android development; the vast majority of these come bundled with the IDEs for both. The subsections below assume the use of Xcode for iOS development, and Eclipse and the ADT for Android development.

2.3.1 Source Code Management Integration

Both Xcode and Eclipse support the use of either Git or Subversion for source code management (SCM). Xcode comes with built-in support for both, allowing the developer to configure SCM repositories in the Organizer. Xcode has an additional feature that allows the developer to take a snapshot of their project, which can be restored at any time. Git is required in order for this to work. Eclipse does not come with built-in SCM support, but it can be easily extended to include this by installing plugins such as Subversive [51] or EGit [20]. Eclipse plugins are also available for other SCM systems.

2.3.2 Debugging

Eclipse comes with a built-in Java Debug Wire Protocol (JDWP) compliant debugger that allows the developer to debug their application on the Android Emulator or on a physical device. This provides the standard debugger functionality of setting breakpoints, inspecting variable values, stepping through code, etc. The SDK provides the Android Debug Bridge (adb) which provides the means to communicate with the emulator or device and provides various device management functions. Also integrated in Eclipse is the Dalvik Debug Monitor Server (DDMS) which uses adb to communicate with the device. DDMS can simulate phone calls and user location, and provide device file system access. It can be used to gather thread and stack information and to view log messages, and has a number of other features. The debugger, however, can be quite slow to start up with the developer left waiting for the application to attach after making only minor code changes.

Xcode also provides a debugger that allows you to debug applications on the Simulator or on a device. Again it allows the developer to set breakpoints, view running threads, inspect values and step through code. It however, offers less functionality than the Eclipse debugger as the developer cannot change variable values or perform tasks such as simulation of incoming calls, messages or core location.

2.3.3 Profiling and Analysis

Xcode ships with the Instrumentation analysis tool which allows you to analyse your applications using a number of different analysis instruments and record the data in a trace document. The same data can be collected repeatedly over multiple runs and displayed side by side, or different types of data can be displayed in the same way. A library of different instruments is provided and the developer can choose any combination of network or file activity, memory allocation, CPU usage, core animation and core data monitoring to name but a few. Instruments can even be used to create test harnesses by recording a user's interaction with the application. The profiled data is displayed in a graphical format and points of interest in the graph can be expanded to see the exact line of code that is causing the problem. Another tool provided by Xcode is the Static Analyzer. This will run thousands of possible code paths against your project source, and reports any potential bugs that might exist. This is extremely useful for highlighting memory management issues.

Heap usage and memory allocation in Android applications is monitored through the use of DDMS mentioned previously in section 2.3.2. Heapdumps can be exported from the DDMS or programatically in code and examined in detail using the Eclipse Memory Analyzer. Method profiling is also supported from the DDMS and Traceview is used to provide a representation of the profiled code. Threads are shown in the Timeline Panel while individual methods are shown in the Profile panel, again allowing the developer to drill right down to the code that is causing problems. For stress testing of Android applications, the Android SDK provides the Monkey command line tool. This generates a pseudo-random stream of user events that can be played back repeatedly. Clicks, touches and other gestures, along with certain system events are included.

Both platforms provide very useful tools for the developer to investigate and improve the performance of their applications. Android does not have a comparable tool to the iOS Static Analyzer, however as garbage collection is used on the Android platform this is not a major issue. More detail on memory management can be found in section 3.4.

2.3.4 Emulation/Simulation

Android provides an Emulator as part of its SDK, whereas Apple provides a Simulator. The Android Emulator emulates both the hardware and the software environments that the application will run on, while the iOS Simulator only mimics the software environment that the application will run on. For this reason, Apple places strong emphasis on device testing, as issues such as running out of disk space will rarely be seen on the Simulator which has all the resources of the Mac on which it is running. Because of this lightweight approach, the Simulator is very quick to launch with little time needed between code changes to rebuild, reinstall and relaunch the application. Apple provides 3 Simulators with Xcode; iPhone, iPhone with Retina display and iPad, and a number of iOS versions can be used with these. It is not possible to fully test every application on the iOS Simulator, however, as it does not provide support for features such as mimicking phone calls or user location.

Android provides the Android Virtual Device (AVD) manager which allows the developer to set up many different Android platforms and hardware configurations. The Android Emulator claims to mimic all of the hardware and software features of a mobile device, with the only caveat being that it cannot place actual phone calls. Due to the large numbers of different Android devices and operating system versions, a large number of AVDs will be needed by the developer when testing their application. As mentioned previously in subsection 2.3.2, the Emulator also allows for debugging, providing the ability to simulate phone calls and messages, hardware events, network access and device location. Android also allows the running of multiple emulators at the same time, and built-in commands can be used to simulate phone calls between them, or to configure network redirections to allow one emulator to send data to another.

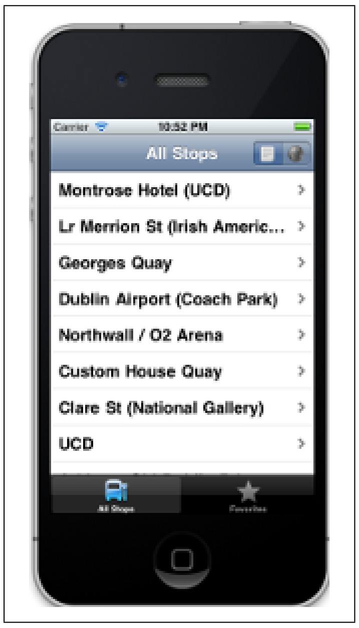

Figure 2.1: Screenshot of the iPhone Simulator

Because the Android Emulator provides both hardware and software emulation, it can be slow to load, often taking up to a minute on its first launch. Less time is needed to rebuild and reinstall the application after code changes as the AVD does not need to be restarted, but it is still not as quick as the iOS Simulator in testing updated code.

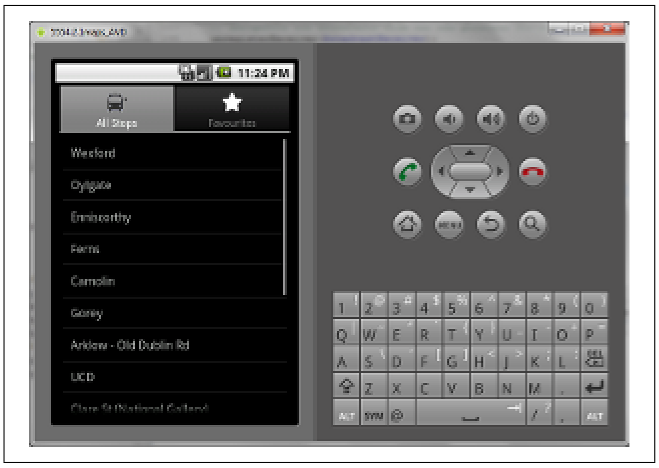

Figure 2.2: Screenshot of the Android Emulator

2.3.5 User interface design

Xcode comes with Interface Builder, a graphical design tool for building iPhone and iPad user interfaces. Interfaces built with this tool are saved in a special type of file called a nib file with a .nib or .xib extension. Interface Builder provides a drag and drop interface where the developer can simply drag a particular type of view object onto the device window, set its properties, and choose which elements in the code to connect it to. Alternatively the developer can create the user interface directly in the code.

iOS comes with a built-in set of user interface objects that can be used, and these can be easily found in the Object and Media library in Interface Builder. Developers may also implement custom views by subclassing one of the standard views if they so desire. On screen objects can be connected to code objects using `IBOutlets` and `IBActions`. E.g. an `IBAction` is used to communicate from the user interface to the background code so would be used for a button, and an `IBOutlets` is how your application controller can communicate with the user interface so may be connected to a text field on the screen. Interface builder provides various menus to allow the developer to correctly size and align user interface objects. Autosizing controls are also included which allows the developer to pin the object to one or more of the edges and to control whether it stretches to fill the available space or not.

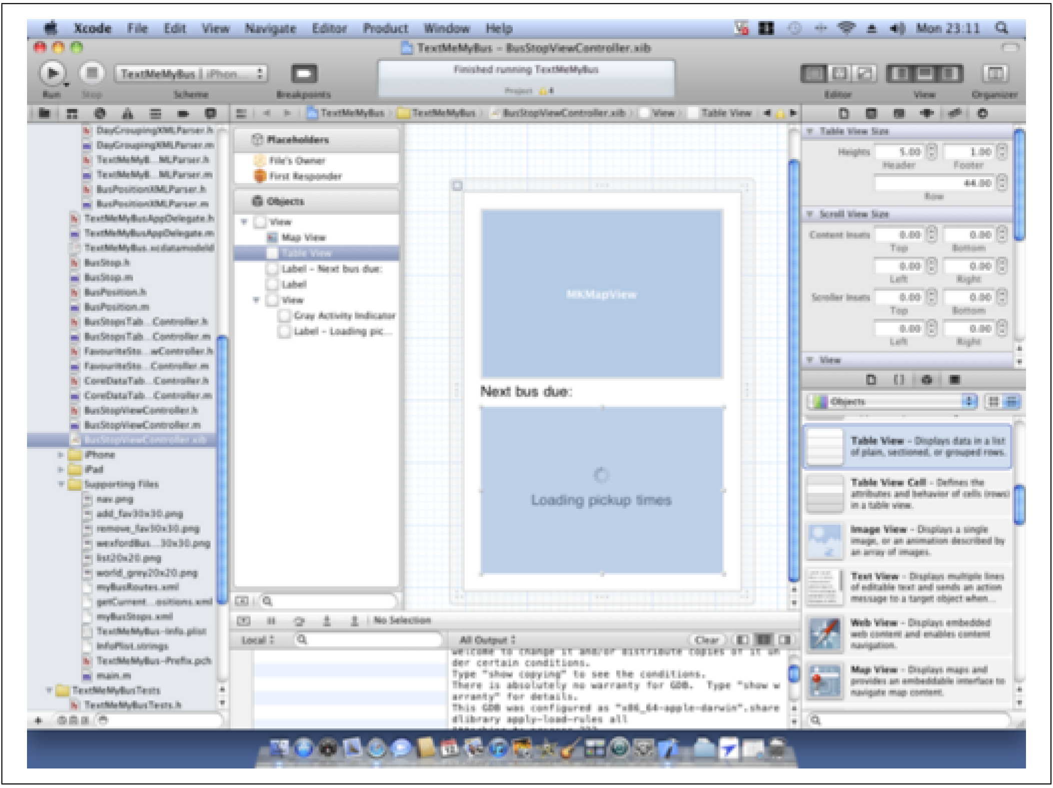

Figure 2.3: Interface Builder and BusStopViewController.xib

Before the release of the iPad, iOS application developers did not really need to worry about application layout on different screen sizes and resolutions. When the iPad launched originally, iPhone applications would only use the portion of the screen that was required to show the application on an iPhone, or it could be stretched to twice that size, which often led to pixelated displays. This meant the applications did not make the most of the screen space available to them. To combat this Apple introduced the Universal Application. This allows a developer to provide separate nib files for both iPhone and iPad and to use programming idioms in the code to discover which device the application is actually running on and how it should behave. Controllers may be shared with the code forking depending on which device is in use, or an alternative approach is to use subclasses of controllers to determine display behaviour.

User interface layout in Android is defined in one of two ways; using XML, or programatically at runtime. In general most layout in Android is done using XML files, but can be manipulated through the code at runtime. The major advantage of keeping the layout defined in the XML is the separation of presentation from the code that controls the application. Another reason is that it is extremely easy in Android to provide alternate layouts for different screen sizes and resolutions. The layout XML files are copied into specific subfolders of the `res/layout` folder for an application, and each version can be customised to the best layout for that particular screen size, density etc. Android will work out what the device screen size and resolution is, and which resource to use that will display the best on this combination. Additional layouts can also be included for orientation and aspect ratio. Within the XML it is easy to enforce how the views will display on different size screens by setting the `fill_parent`[1] or `wrap_content` attributes on a view's horizontal and vertical layout parameters. Specific sizes may also be set, with the available units being pixels, density-independent pixels, scaled pixels, inches or millimetres.

Android comes with a set of predefined layouts and widgets that can be used by the application developer. These layouts can be nested inside each other to combine into more complex layouts. A simple example of this is how scrolling can be added to an application. The `ScrollView` is set to be the root view of the XML and any other views, e.g. `LinearLayout` are nested inside this. Any of the existing widgets can be easily included in the XML or in the code, e.g. Buttons, Checkboxes, Progress bars, Spinners, Time Pickers etc. User input can be handled in the code by implementing Listener interfaces for each of these widgets and views. Android also allows the developer to provide their own custom views and widgets by extending the core layout APIs. These interfaces and classes can be found in the `android.widget` package and are well documented.

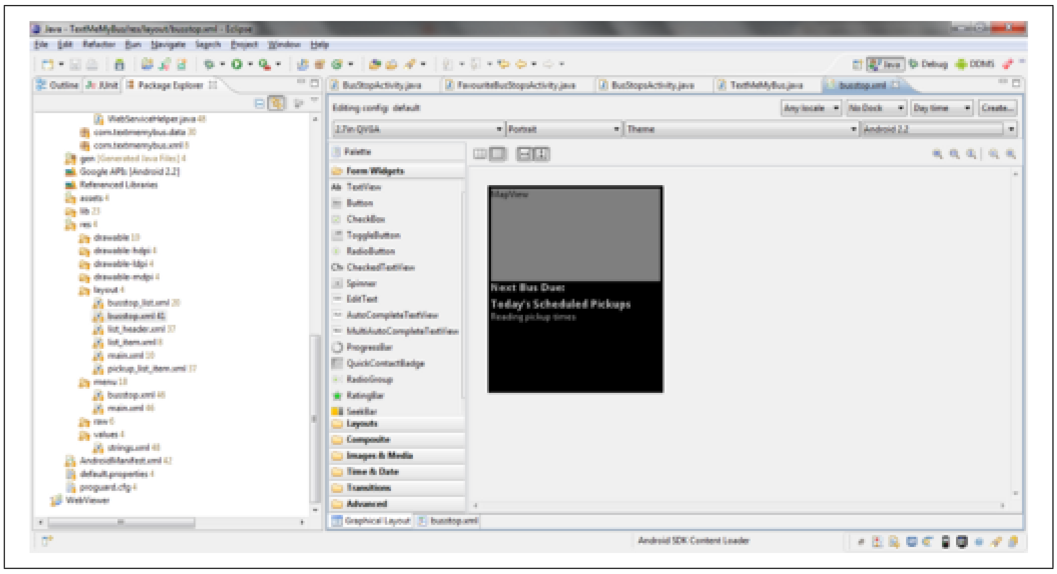

Figure 2.4: Eclipse Graphical Layout editor

Eclipse does not include a full graphical design tool for user interface layouts in Android. There is a Graphical Layout tab for editing layout XML files, but this is poor in comparison to Interface Builder. It may be simpler for the developer to edit the XML manually in the XML editor. The use of the various AVD configurations is essential for determining the actual screen layout, as one can be created for every screen size and density required. Another useful tool included in the Android SDK is the Hierarchy Viewer. This provides static snapshots of the View objects that make up the user interface of the activity that is currently running, the relationships between these and the various properties that each View has. Android also provides the layoutopt tool which can be run against the layout XML files to highlight any inefficiencies in the view hierarchy.

The graphical user interface design tool provided by iOS is far superior to the one provided

[1]This has been renamed to match_parent from API level 8 onwards

by Android. Android attempts to address this by providing extremely flexible layout options through the use of XML. Android also allows for the testing of various layouts using the AVD, however this can be quite slow. Both platforms allow for the separation of the view from the controller code which is extremely useful to the developer.

2.4 Devices

Apple is the sole manufacturer of the iPhone and therefore has total control over the hardware and software on the phone. This means that Apple does not have to compete with other manufacturers offering similar phones with iOS in the way that Android manufacturers do. Because of the phenomenal success of the iPhone, and being the sole manufacturer of the device allows Apple to negotiate with hardware manufacturers, e.g. for components such as flash memory or retina displays. Apple controls the standard of component going into the device and knows exactly how the operating system will perform on it.

Having only one manufacturer of devices for iOS means that developers need to worry less that their applications will not work on all iOS devices. Having said that, there have been 4 models of iPod touch, 5 models of iPhone and 2 models of iPad since the initial launch. The iPhones all have the same 3.5 inch screen and resolution, with the only difference being the iPhone 4S doubles the resolution, so developers have had little to worry about in terms of how their applications will look on iOS devices. With many different devices running different versions of iOS, developers are also able to restrict their applications to certain devices and iOS versions in the App Store by setting the required device capabilities in the application properties, and configuring the base SDK and deployment target settings on their applications in Xcode.

A wide range of Android devices are available on the market today. These are produced by manufacturers such as HTC, LG, Samsung, Motorola, Sony Ericsson and a number of other smaller manufacturers. Because Android is used by numerous device manufacturers, and on many different handsets, fragmentation is more of a worry for the Android developer. How does the developer ensure that their application will operate in a consistent way on the various devices running the Android OS? This is exacerbated by the fact that manufacturers and telecommunications companies often put their own skin on top of Android, e.g. HTC Sense. Ensuring consistency is one of the main focuses of the Open Handset Alliance mentioned in section 1.1 and so far they seem to be doing a good job. Android devices tend to be priced slightly lower than iOS devices and this is probably due to the increased competition of having a range of devices available for the same platform. As Apple is the only iOS device manufacturer they are in a stronger position than Android device manufacturers and are most likely making a higher profit because of this.

The Android Market allows the developer to specify certain hardware and software capabilities that are required for the application to run on a device. These are defined in the Android Manifest and applications are filtered out of the market view for devices which do not meet these requirements. Developers can also specify minimum and maximum SDK versions for their applications, and Android have recently introduced the option to provide different versions of the same application which are targeted at different versions of the platform.

Having a single manufacturer for iOS devices means less work for the developer to ensure that their applications run consistently across all iOS devices. The Android developer must be careful to consider other resolutions, hardware capabilities and software versions. Android attempts to address this by allowing the developer specify different layouts and resources for different device specifications, and by filtering the applications that are available to device users in the Android Market.

2.5 Summary

This chapter has covered the initial steps that a developer must take in order to start developing for Android or iOS. If developing for iOS the developer needs a Mac and Xcode, whereas if developing for Android the developer has a choice of operating system and IDE. Once these are in place the developer should become familiar with the large variety of tools available to them. Developers can choose whichever SCM solution they prefer, but are limited to the debuggers, emulators or simulators and user interface design tools that come with each software development kit and integrated development environment. Finally, the developer must bear in mind the target devices for their applications and, if possible, obtain some for testing purposes. In the case of Android there are a huge number of devices available, with different resolutions, hardware capabilities and running different versions of the Android operating system. In the case of iOS, Apple is the sole device manufacturer so the developer has much less to worry about.

The next stage for the developer is to make sure that they have a good understanding of the languages used to write these native applications, and this is covered in the following chapter.

Chapter 3: **Languages**

A major consideration for the developer starting mobile application development is the language that is used to write the application. If a developer is not familiar with the required language, a considerable amount of time and effort may be required for the developer to learn it. iOS native applications are mostly written in Objective-C, although Ruby and Python can also be used. Android applications are mostly written in Java, although the Android Native Development Kit (NDK) is also provided to support the implementation of parts of applications in native-code languages such as C and C++. This chapter will focus on the use of Objective-C and the Cocoa application frameworks, and Java and the Android frameworks. Both languages are object oriented and offer the standard features such as classes, polymorphism, inheritance, method overriding, etc.

Objective-C [40] is an extension to the standard ANSI C language which enables object oriented programming. As Objective-C is an extension of C, C and C++ code may also be included in applications, and functions outside of the Cocoa frameworks may be called.

Android makes use of the Java language [29] and the majority of the Java Standard Edition class libraries, adding its own libraries while leaving others out. When the code is compiled, Java .class files are first created, and these are then transformed into the .dex format and executed on a Dalvik VM. Each application runs on its own Dalvik VM.

Section 3.1 begins by comparing the constructs required for object creation in Objective-C and in Java. Section 3.2 details how methods are invoked in each language, and this is followed by section 3.3 which discusses the use of properties in Objective-C. Section 3.4 compares and contrasts how memory is managed in both platforms. Section 3.5 and section 3.6 provide information on categories and blocks, neither of which are supported in Java. Section 3.7 provides a summary of the information discussed in this chapter.

3.1 Object Creation

Object instantiation in Objective-C is done using `alloc` and `init`. `alloc` dynamically allocates memory for the instance variables belonging to the object and initialises these all to 0. The `init` method is then responsible for initialising these instance variables to be something useful, and may be implemented to take in additional parameters.

```
MyObject myObject = [[MyObject alloc] initWithName:@"Object Name"];
```

Objective-C uses `id` as the general type for any kind of object, however objects can also be statically typed. `NSObject` should be used the base class of all classes in Objective-C. `nil` is used in Objective-C for defining null objects. Sending a message to nil is allowed and has no effect at runtime.

In Java objects are instantiated using the `new` keyword, which causes the object constructor to be invoked. If no constructor is defined, a default constructor from `Object` is used. `Object` is the base class of all classes in Java. Java classes are statically typed.

```
MyObject myObject = new MyObject("Object Name");
```

3.2 Message Passing

In Objective-C to get an object to do something, you send it a message telling it which method to apply. Objects in Objective-C are dynamically typed at runtime. This means that the exact message invoked is only determined at runtime and not when the code is compiled. Messages can also take parameters or arguments. The parameter name is suffixed with a colon and the parameter follows immediately afterwards. Methods that take parameters should be named in such a way that the method name describes the parameters used, as the parameter names are included when calling the message.

```
[myObject doStuff];
[myObject moveFromPoint:point1 toPoint:point2];
```

In Java methods are invoked directly on the instance of the class to which they belong. In Java the parameter names are not included when calling the method.

```
myObject.doStuff();
myObject.moveFromPointToPoint(point1,point2);
```

3.3 Properties

Getters and setters are commonly used by developers to control access to instance variables. In Objective-C the convention is to simply use the variable name as the name of the getter method and to prefix this with ”set” for the setter method. E.g.

```
-(NSString*)name;
-(void)setName:(NSString*)myName;
```

Objective-C has introduced the concept of a property to make it easier to access instance variables using dot notation. Adding the `@property` notation for the instance variable in the header file causes the compiler to automatically generate the getter and setter declarations. The getter and setter methods for the instance variable can be invoked as follows:

```
NSString *objectName = myObject.name;
myObject.name = @"Object Name";
```

Further addition of the `@synthesize` notation for the instance variable in the implementation causes the compiler to automatically generate the getter and setter for the variable. This means that the developer does not have to implement these methods unless custom processing or validation is needed. This also keeps the code tidy as simple getter and setter code does not need to be included in the object implementation.

This concept is unique to Objective-C as Java does not have anything comparable. Most modern IDEs, however, offer the ability to generate getters and setters for instance variables, but the code for these methods must be explicitly included in the Java interface and implementation. The developer can easily generate these methods and customise them further if required.

3.4 Memory Management

With the limited capabilities of mobile devices, memory management becomes extremely important. No developer wants their application to be uninstalled because it is not responsive, because it crashes at runtime, or because it is draining the battery, so it is important to balance carefully the creation of objects needed by an application. Each application is responsible for creating only the amount of objects that it actually requires, and to ensure that these get destroyed correctly when they are no longer needed so that resources are not wasted.

Objective-C supports three kinds of memory management: automatic reference counting (compiler), manual reference counting (programmer) and garbage collection (not available for iOS). Manual reference counting is provided by NSObject and the runtime environment, but the developer is responsible for keeping track of objects used. Objects are allocated using `alloc` and `init` as described in section 3.1, and must be retained, released and deallocated correctly in the application code. Automatic reference counting is supported for iOS 4 and 5 and uses the same reference counting mechanism as manual reference counting, but the appropriate `release`, `retain` and `autorelease` calls are inserted at compile time so the developer does not have to do this manually. If Core Foundation framework objects are used, however, these still need to be managed appropriately so the developer cannot completely forget their memory management responsibilities. Automatic reference counting is enabled by the use of a compiler flag. Garbage collection is supported by Objective-C but not by iOS so does not apply to mobile application development.

In Android applications, the Dalvik runtime is garbage collected, but developers must still be aware of their application's use of memory. As the garbage collector looks after the allocation and freeing of memory, the developer does not need to write code for this, but instead must be careful not to retain references to objects that are no longer needed. Doing so will prevent these objects being garbage collected and will prevent the allocated memory from being reused. With device rotation, the entire activity in view is destroyed and recreated. If there are any static references to the original view this will cause the memory used by that view's objects to remain on the heap so it will never be garbage collected. This means that there is now less memory available to the application.

Within Android there is a hard limit on the amount of heap size available to each application. This is device dependent so devices with more memory will allow each application a larger heap size. Because the heap size is limited, if an application has already exhausted its allocated size and tries to create a new object, an out of memory error will be seen. The length of time required for garbage collection also depends on the heap size and this is an area that has recently been improved upon with the release of Android 2.3. Previous to this, a "stop the world" garbage collector was used, where the application was paused while garbage collection was in progress. This could lead to the application seeming to be unresponsive. With the release of Android 2.3 concurrent garbage collection has been introduced. This means that the garbage collection runs in a concurrent thread to your application, collecting in the background, which leads to a huge improvement in application pause times.

3.5 Categories

Categories allow the developer to extend the functionality of a class without subclassing. They allow the developer to add additional methods to any class, and these methods are added to the class type. Categories are supported by Objective-C but Java does not offer a similar construct. Categories are defined with the same interface name as the class it is adding to, and the unique name of the category in brackets afterwards. The header and implementation files are usually named with the class being added to joined with a plus sign to the name of the category. E.g. `MyClass+MyCategory.h`, `MyClass+MyCategory.m`

```
@interface MyClass (mycategory)
-(NSString*)myNewMethod;
@end

@implementation MyClass (mycategory)
-(NSString*)myNewMethod{
  // implementation code goes here
}
@end
```

All that is required to use this is the import for the category header file and the methods will now be available on all MyClass objects. At runtime there is no difference between methods added in a category and methods declared on the class itself, and both are available to subclasses. Categories have access to the instance variables of the class they are adding to, but cannot add instance variables of their own. Multiple categories may be added for the same class, but each must have a unique name and must declare different methods.

3.6 Blocks

The use of blocks has been a feature of scripting and programming languages such as Python and Ruby for some time now[1]. iOS 4 introduced the use of blocks in Cocoa application development. Java currently does not offer a similar feature, however this may be added in Java 8 so will probably also make its way into Android. Blocks are essentially anonymous functions, encapsulating a set of instructions and the data required by them, and can be passed around like objects. Blocks can be anonymous or named, can take in parameters and can return values. Blocks close around the variables that they make use of, i.e. if an instance variable is used inside a block, the value used in the block is the value that it had when the block was instantiated.

When used as method arguments, blocks are a form of callback which allow the developer to customise the called method's code. Since iOS 4 new methods which take blocks have been added to the collection classes, the `NSNotification` class, `UIView` for animations, and to many other places. In the example below the animations parameter is a block which simply causes the view to fade out.

```
[UIView animateWithDuration:1.0
                 animations:^ {bgDisplay.alpha = 0.0;}];
```

[1]Blocks are also referred to as closures or lambdas.

One common use of blocks is completion blocks. A completion block is a block that is passed as a parameter with a message and which is executed on completion of the called method. Another common use is error handling blocks, where the block is again passed as a parameter to a method and is executed if an error occurs in the called method. This allows the developer to inject custom error handling code. Blocks are a vital feature of Grand Central Dispatch, a concurrency mechanism which is discussed in more detail in section 5.2.

3.7 Summary

Some of the major differences between Objective-C and Java have been discussed in this chapter. While both languages are object oriented, there are a wide range of differences between them, as discussed in the sections above. Due to this, a developer familiar with only one of these languages may disregard development in the other as the overhead of learning it is simply too great. Objective-C offers a number of features that Java has no support for, for example categories, properties and blocks. The use of properties can reduce the amount of code a developer has to write, and reduces clutter in object implementations where no further customisations are needed. Categories make it extremely easy to add new functionality to a class without subclassing. Developers who have never used blocks before may find this an alien concept at first, but with a little experience find that blocks are extremely flexible and useful to the developer. Java developers in particular may struggle with the object creation and memory management that must be carefully considered with Objective-C development, whereas Objective-C developers may prefer the manual control over the amount of resources in use at any one time rather than relying on garbage collection.

Once a developer has a good knowledge of the language they will be working with, they can move on to their next major task, the design of the application itself.

Chapter 4: **Design**

Good design is crucial for mobile application development. If an application is not designed correctly from the beginning, minor changes at a late stage may require major rework of the application code. Application code should be easy to maintain so that any bugs can be quickly fixed, and so that new features can easily be added. It is also important to ensure that the application performs well, as inefficient code or mismanagement of memory will lead to applications being uninstalled by disgruntled users.

This chapter deals with the design of iOS and Android applications. Section 4.1 begins by looking at the architectures of the Android and iOS platforms. This is followed by a discussion on the software patterns that are commonly found in Android and iOS applications in section 4.2. Section 4.3 and section 4.4 go into more detail on the internals of each type of application with an in depth exploration of the application structures and application lifecycles. Section 4.5 provides a summary of the information discussed in this chapter.

4.1 Platform Architecture and Frameworks

The Android operating system is made up of a number of core components which can be seen in figure 4.1.

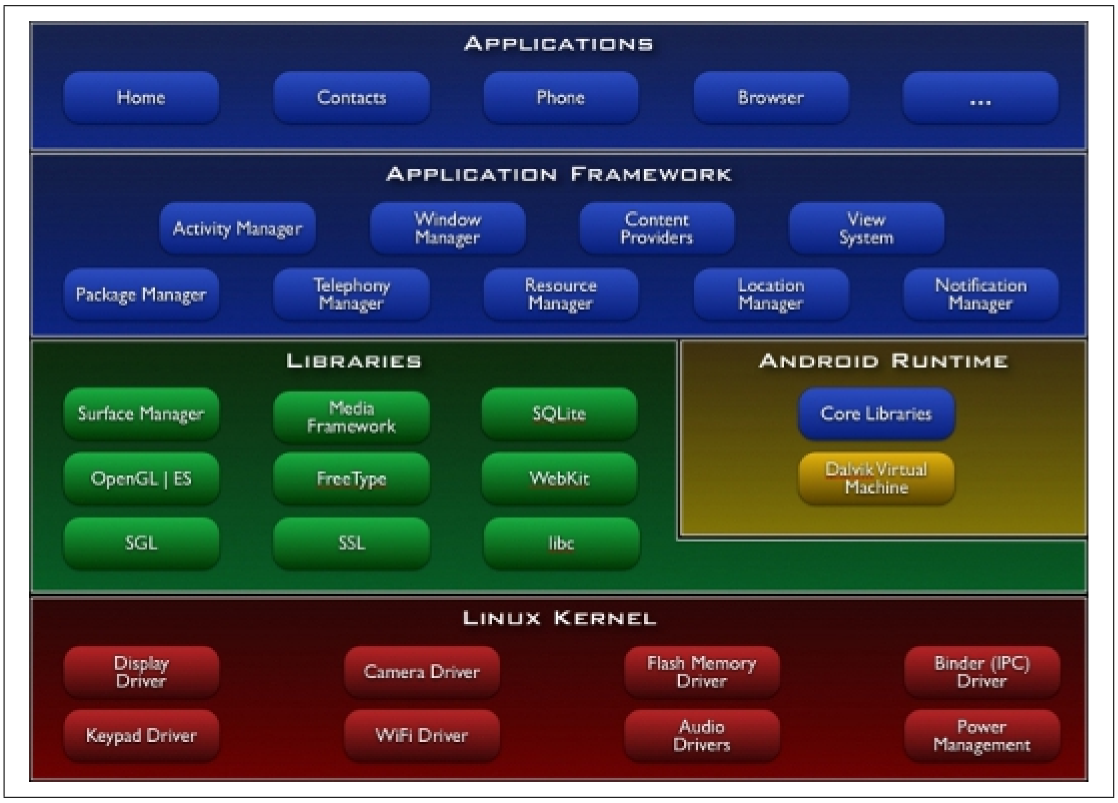

Figure 4.1: The Android Architecture, source Android Developers [2].

Core and third party applications are at the top layer and sit on top of the application framework. This is made up of a number of components which allow the developer to manage activities, windows, resources, location, etc. in their applications. These components are based on a set of libraries and the Android runtime. These Android libraries are mostly written in C and C++ and are exposed to application developers through the framework APIs. The application framework components depend also on the Android Runtime. This consists of the core Android libraries which provide most of the functionality of the core Java libraries, and the Dalvik Virtual Machine upon which all applications run. At the base of the Android platform is the Linux Kernel. This handles hardware, power, process, security and memory management and provides functionality to the Dalvik VM such as threading and low-level memory management.

The iOS architecture can be viewed as a set of 4 main layers which can be seen in figure 4.2.

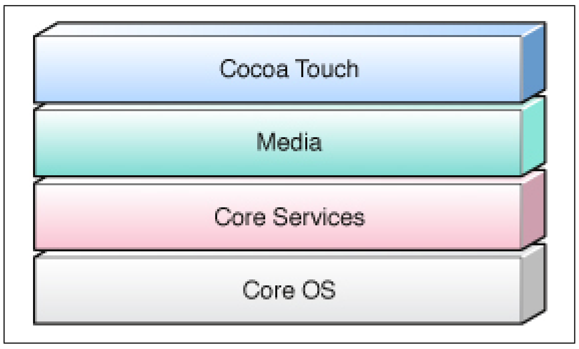

Figure 4.2: Layers of iOS, source iOS Developer Program [14].

The Cocoa Touch layer is at the top of the stack and is the one most used by application developers. It provides object oriented abstractions for the lower layers. The lower layers may also be used by application developers who want to use them directly or access functionality not exposed by higher layers. The Cocoa Touch layer provides multitasking, notifications, gesture recognisers, standard system view controllers, etc. It is made up of a number of frameworks such as Address Book, Event Kit, Game Kit, iAd, Map Kit, Message, Twitter and UI that developers can use in their applications. The next layer in the iOS stack is the Media layer and this is responsible for graphics, audio and video technologies on iOS devices. It consists of a number of core frameworks along with other frameworks such as OpenGL ES [31], Media Player and Assets Library which can be used for multimedia integration. The Core Services layer contains the fundamental system services that all applications use and is built upon by many other parts of the system. This area holds vital technologies such as Block support, Grand Central Dispatch, SQLite [49] and much more. The Core Data, Foundation, Location, Telephony, and many other commonly used frameworks are found in this layer. The Core OS layer is the foundation of the stack and holds the lower-level features that the other technologies are built upon. This contains the security, system level, bluetooth, acceleration and external accessory frameworks. Functionality in all of these layers is available to application developers, however, they are encouraged to use the highest layer possible that contains the functionality they require.

4.2 Patterns

A number of common patterns can be seen in native mobile applications, regardless of platform. Model-View-Controller (MVC) is probably the most common of these [45] [46]. MVC was first described by Reenskaug in 1978 with regards to Smalltalk-80 but is still relevant in object oriented programming today. MVC involves separating the code to handle the application data (the model), from the code used to display this to the user (the view), and also separating out the code to determine the interactions between these two (the controller). In general this is extremely useful as separation of data and user interface means that, for example, a new database can be used, or a new user interface can be swapped in with minimal disruption to the rest of the application. MVC is a composite pattern so several other patterns are used within it. The model object holds the application data, the view presents data to the user, and the view controller communicates with the model and the user interface and determines what data the view should display.

In the LiveBus application, one example of the MVC pattern can be seen in the context of the bus stop. The class responsible for the bus stop data, i.e. their locations and scheduled pickup times, is the model and the controller is responsible for getting this information for the user and setting up the view that is to display them. In the iOS application the controller is `BusStopViewController` while `BusStopViewController.xib` contains the user interface information with the map and the table views to display the relevant data. Similarly in the Android application, `BusStopActivity` is the controller and the `BusStop.xml` layout file contains the map and the table views to display the given data. In both cases the views are generic widgets provided by the platform and the data to be displayed is simply fed to them by the controller. There is no direct communication between the model and the view, all communication is initiated by the controller. The controller is the data source for the view rather than the model, and determines what subset of the model's data should be displayed on the view. The model implementation is specific to the data that needs to be displayed, while the view is generic, and the controller is the bridge between both.

Delegation [33] is commonly seen in the MVC pattern. In this case the view needs to communicate with the controller when certain events happen, however, as the view is a generic one provided by the platform, the view cannot know any specific information about the controller. When the controller first sets up the view, it sets itself as the delegate of the view. The view knows that the delegate responds to a certain set of messages, but knows no more about the delegate object. This allows the view to get more information from the controller by calling the delegate methods. In the iOS application, the `BusStopViewController` adopts the `UITableViewDelegate` protocol by implementing the required methods. When the view is created by the controller it sets itself to be the delegate of the `UITableView` to display the pickup times. On selection of a row in the table, the `UITableView` knows to inform its delegate by calling the method `tableView:didSelectRowAtIndexPath:`. In the case of `BusStopViewController` this is not implemented as it is an optional method on the protocol and no processing is required when a pickup time is selected.

Another way that the view can communicate with the controller is using the target-action mechanism. This is a simplification of the Command pattern as documented by Gamma et al [22], and is commonly seen in Objective-C development. For example, the controller wants to know when a button is pressed so sets up a target action and gives this to the view. In `BusStopViewController` the controller itself is set as a target on the favourites button with a method selector as the action for the event the controller is interested in.

```
[favButton addTarget:self
              action:@selector(favouriteStop)
    forControlEvents:UIControlEventTouchUpInside];
```

A selector is simply an identifier for a method in the target. When the required event occurs, the view then sends the given action message to the given target. The view only knows that it is sending the message to some target, and knows nothing about the implementation details. The view can pass itself as a parameter with the message so the target can query the view for the details of the event that has occurred.

If the model needs to communicate with the controller, for example, to let it know that the data has changed and the view should be refreshed, there are a number of options available. The model should not communicate directly with the controller, the link is one-way and should only be from the controller to the model. The view should instead broadcast a notification which the controller (among others) can listen for, or use a technique like key value observing where the observer is directly notified of the change. In iOS broadcasts are done using `NSNotification`, while in Android `Intents` and `BroadcastReceivers` are used.

MVC groups can be combined to create more complicated applications. These work together to form the whole application. In the LiveBus application, a MVC group can be seen for the list of bus stops, the favourite bus stops and also for individual bus stops. When a bus stop is selected from the list of bus stops or favourite bus stops, the controller for the bus stop is launched and the view for the individual bus stop is shown. In certain cases the view pointers of some controllers can connect to sub-controllers rather than directly to a view. Also models can be shared and views can be reused between MVC groups. With MVC groups working together to form a complete application, the developer needs to be careful to set object responsibilities and boundaries correctly. The developer should aim to keep to a minimum the number of connections in and out, and to minimise interactions with others.

4.3 Application Structure

Four main application components exist for building Android applications: Activities, Services, Content Providers and BroadcastReceivers. Each of these is a unique entry point for an application and helps define how it behaves. Each component has a specific purpose and a distinct lifecycle that determines how it is created and destroyed.

Activities are most familiar to an Android developer as they correspond to a single screen in the user interface. Typically Android applications are made up of a number of activities which are linked together. Each activity is an independent entity and so can be reused across applications if desired. The LiveBus application consists of a number of activities: `BusStopsActivity`, `FavouriteBusStopsActivity` and `BusStopActivity`.

A service can be thought of as a background operation which is designed to handle long running tasks or remote operations, and which does not have an associated user interface. The `BusPositionService` in the LiveBus application is an example of this. It repeatedly polls a web service for bus position data.

A Content provider is responsible for the management of a set of shared data between applications. The data can be stored in a number of different ways, and any application with the required permissions can query or modify it. A number of system content providers are accessible to Android developers, one of these being the `ContactsContract` which allows access to a wide range of functionality around the user's contacts. Developers can write their own content providers to share data with other applications.

A broadcast receiver is a component that listens for and responds to system-wide broadcast events. Broadcasts may come from the system, for example, Low Battery, or from another application. Broadcast receivers generally do not have an associated user interface, but quite often use status bar notifications to alert the user that something has occurred. Often the broadcast receiver is simply used to listen for an event, and then start a service to perform some processing based on this event occurring.

Activities, services, and broadcast receivers are all started from `Intents`. This is an asynchronous message and may be implicit or explicit, i.e. the intent may define a message to start a particular component, or a particular type of component. For activities and services an intent may also contain information required to start the activity or service, e.g. the URI of data to be used. For broadcast receivers the intent simply states that an event has occurred. Content providers are not accessed using intents but by using `ContentResolver` queries. `ContentResolvers` provide a layer of abstraction between the content provider and the component using it.

Activities, services, broadcast receivers and content providers are tied together to form an application using the information provided in the `ApplicationManifest`. This is an XML file which defines which components form the application and any permissions, hardware, OS versions etc. that are required for the application to run.

For iOS applications, the UIKit framework manages the application's behaviour through the `UIApplication` object. This manages the application event loop by handling system events and passing them onto the application delegate which contains custom code for processing these events. The delegate is also responsible for setting up the initial navigation structure for the application, as can be seen in `application:didFinishLaunchingWithOptions:` in the `TextMeMyBusApplicationDelegate` implementation.

Most iOS applications contain a single window which is set up by the application delegate. The content displayed on the window is determined by the current view controller and the application delegate sets up the view controllers and sets the view to be used in the `application:didFinishLaunchingWithOptions:` method.

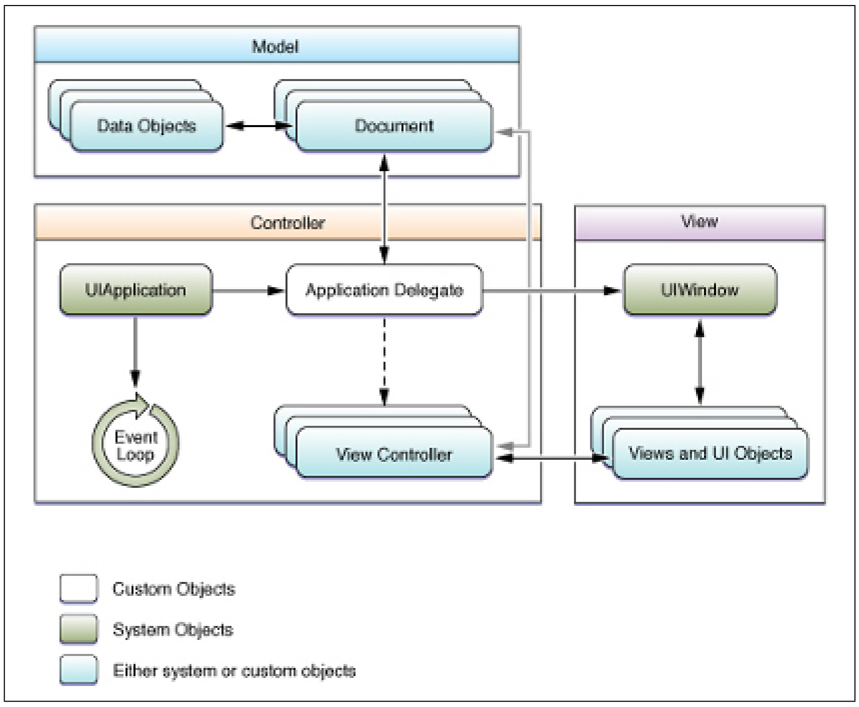

Figure 4.3: iOS application structure, source iOS Developer Program [14]

View controllers manage the presentation of application data on the device screen. A view controller is responsible for a single content view. However, this view may have multiple subviews. The applications custom view controllers are responsible for initialising and loading views, handling device rotation, and responding to events from the user interface usually through the use of delegation or action methods. Usually views are made up of standard

UIKit view objects, for example, text areas or buttons, and may also contain custom views or subviews of the existing view objects. Animation may also be added thorough the use of the Core Animation frameworks, and if more sophisticated drawing is required OpenGL ES views may be used.

All the application configuration data for an iOS application is stored in the `Info.plist` file. This structured key-value pair data file is used by the operating system to determine how to interact with the application. It stores information such as the Minimum OS version, icon files, device hardware requirements, the main nib file name, custom application information and any other information required to run the application.

4.4 Application Lifecycle

There are five possible states for an iOS application: Not Running, Inactive, Active, Background or Suspended. The normal state for an application is Active which means that it is running in the foreground and is receiving events. If an application is in the foreground but is not executing events it is said to be Inactive, in this case it may be executing long running code, or the system is prompting the user to respond to some event, so the application cannot handle user interface events. Applications may also be run in the Background where they can still execute code. If the application is running and, for example, the user answers a phone call, the application is Suspended which means that it is in the background, but not executing any code. An application is in the Not Running state if it has never been started, or has been terminated by the system, for example, to free up resources.

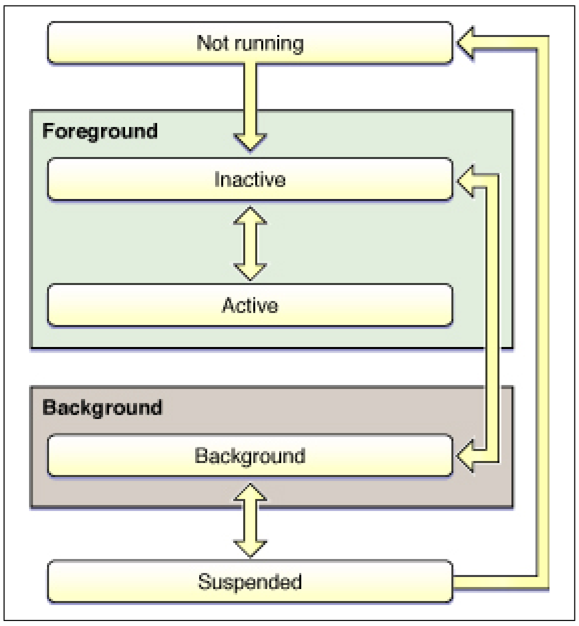

Figure 4.4: State changes in an iOS app, source iOS Developer Program [14].

There are a number of methods in the `UIApplicationDelegate` protocol that handle these state changes for an iOS application.

- application:didFinishLaunchingWithOptions
- applicationWillResignActive:
- applicationDidBecomeActive:

- applicationDidEnterBackground:
- applicationWillEnterForeground:
- applicationDidReceiveMemoryWarning:
- applicationWillTerminate:

These methods allow the developer to respond to state changes in their application in an appropriate way. For example, in the LiveBus iOS application, there is no need to read the bus stop arrival prediction times when the application is in the background. The `applicationDidEnterBackground` method stops the timer when the application has moved to the background, and the `applicationWillEnterForeground` method is used to reload the predictions to start a new timer when the application is about to move to the foreground once more.

As well as `UIApplication` lifecycle changes, the iOS developer has `UIViewController` lifecycle changes to deal with also, and there are a number of methods that can be implemented for this. These methods fall into two cycles, the load cycle and the unload cycle.

If the view is created with Interface Builder and stored in a .xib file, `initWithNibName:bundle` should be implemented with the appropriate information and should be used to initialise the view controller. When the view is actually loaded for the first time `loadView` is called and the default implementation for this is to load the view from the nib file specified previously. If the view is not created in Interface Builder but instead is created programatically, `initWithNibName:bundle` is not needed and the developer should override the implementation of `loadView` to create the necessary view objects and assign the container view to the controller's view property. The `viewDidLoad` method is invoked next to allow the controller to perform any additional load time tasks. Taking the `BusStopViewController` in the LiveBus application as an example, this is loaded from a nib file so `initWithNibName:bundle` is implemented, and `loadView` is not overridden. The `viewDidLoad` method is responsible for loading the pickups from the web service, and setting up the map annotation and region, and the favourites bar button item.

The main methods the developer is concerned about for the unload cycle are `viewDidUnload` and `didReceiveMemoryWarning`. The `didReceiveMemoryWarning` method is called when the system is low on memory and should be used to release all non-critical data used by the view. The `viewDidUnload` method is called when the `UIViewController` releases its views and should be used to release any view specific data e.g. view outlets. The `dealloc` method should never be called directly, but is called by the infrastructure and should be used to release all data structures associated with the view controller.

A number of additional methods are provided by the `UIViewController` to allow the developer to respond to view events.

- viewWillAppear
- viewDidAppear
- viewWillDisappear
- viewDidDisappear

These are called right before and after the view appears and disappears, and allow the developer to do any further processing that might be required. Usage of `viewWillAppear` can be seen in the `CoreDataTableViewController` implementation in the LiveBus application. This is the superclass of BusStopsViewController and FavouriteBusStopsViewController, and calls the method performFetchForTableView before the view appears. This fetches the data to be displayed in the table from core data and calls reloadData on the UITableView. An example of functionality that could be put into the viewWillDisappear method is saving user entered data, although the developer should be careful not to do long running tasks at this stage.

Each of the four android components mentioned previously have distinct lifecycles with Activity and Service being the most complex. There are three main states that are possible for an Android activity: Resumed, Paused and Stopped. A Resumed activity is in the foreground and has focus, it is running. A Paused activity is also in the foreground and is visible, but no longer has focus as another smaller or partly transparent activity is visible on top of it and this has the focus. All state and member data is retained and it is still attached to the window manager, however, a paused activity may be terminated if system memory becomes very low. A Stopped activity is in the background and is completely hidden by another activity. State and member data is maintained, but it is no longer attached to the window manager. The activity can be terminated by the system when memory is needed. Paused and Stopped activities can be terminated by the system either by calling the finish method or by killing its process.

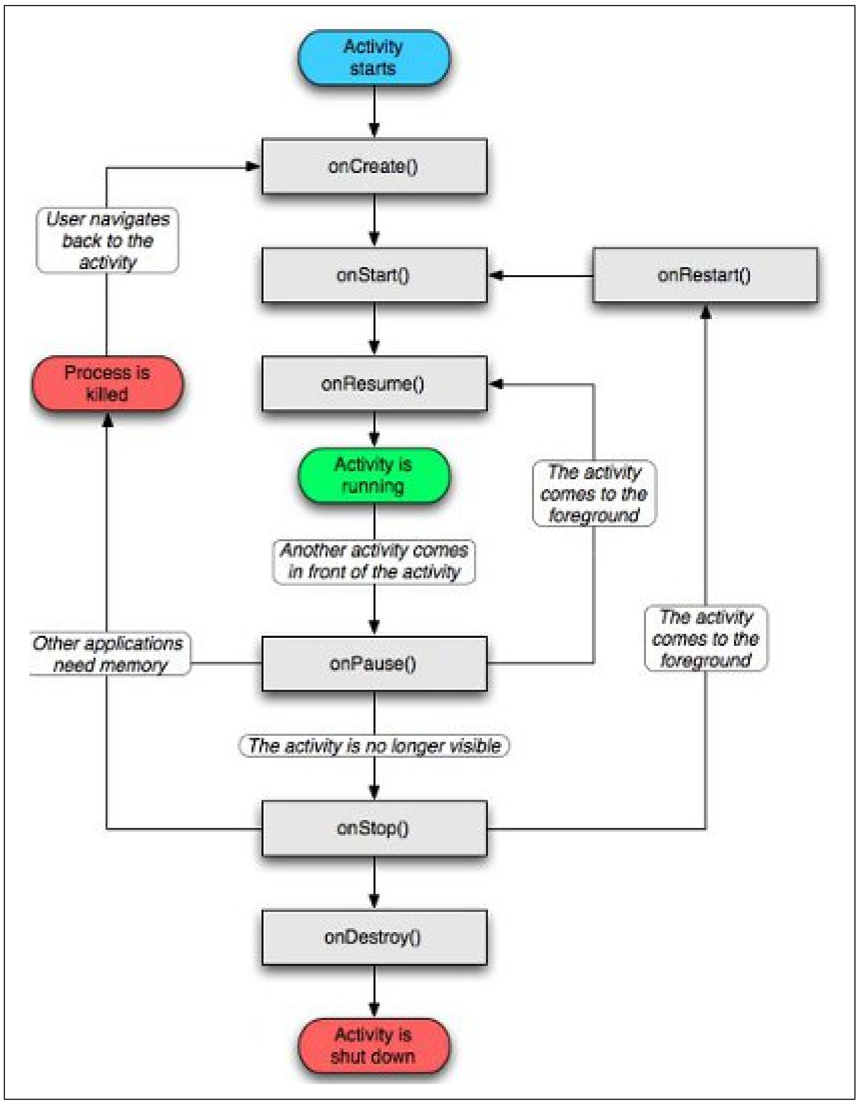
Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
Process is killed
onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
The activity comes to the foreground
Other applications need memory
onPause()
The activity is no longer visible
onStop()
onDestroy()
Activity is shut down

Figure 4.5: The Android Activity lifecycle, source Android Developers [2].

The following callback methods are provided to allow the developer to maintain the correct state for their activity through state transitions.

- onCreate
- onRestart
- onStart
- onResume
- onPause
- onStop
- onDestroy

Any implementation of these methods must call the superclass implementation before any custom code is executed. These methods define the entire activity lifecycle and can be separated into three main execution loops: the entire lifecycle which occurs between `onCreate` and `onDestroy`, the visible lifetime which occurs between `onStart` and `onStop`, and the foreground lifetime which occurs between `onResume` and `onPause`. Figure 4.5 represents a number of possible lifecycle paths for an activity.

`onCreate` is called when the activity is first created and view setup should be done here. This takes in a bundle of the previous activity state if applicable and is always followed by `onStart` which is called just before the activity becomes visible to the user. This is followed by `onResume` if the activity comes to the foreground, or `onStop` if it becomes hidden. `onResume` is called just before the activity starts interacting with the user and when the activity is at the top of the activity stack. In `BusStopsActivity` in the LiveBus application, this is used to register a `BroadcastReceiver` to get updates from the `BusPositionSevice`, and to redraw the map with the data available. `onPause` always follows `onResume` and is called when the system is about to resume another activity. In `BusStopsActivity` the activity unregisters for broadcasts from the `BusPositionSevice`. These are no longer needed as the activity will not be in the foreground. The developer should be careful not to perform long running tasks here as the next activity will not resume until this method has completed. `onStop` is called when the activity is no longer visible to the user, either because another activity has resumed and is hiding it, or because it is being destroyed. If the activity is coming back to the user `onRestart` is called which is followed by `onStart`, or if the activity is going away completely `onDestroy` is called. `onDestroy` is called either because the activity is finishing, or because the system is killing it to regain resources. In `BusStopsActivity` `onDestroy` is used to close off the database connection.

The Service component always runs in the background as it has no user interface and falls into two categories, Started and Bound. A Started service is created when another component such as an activity calls `startService`. Once started the service runs indefinitely in the background until it is stopped by the component calling `stopService`, or until it has completed its function and has stopped itself by calling `stopSelf`. If neither of these things happen the service will run indefinitely in the background. `BusPositionService` in the LiveBus application is an example of a Started service. A Bound service is started when a component calls `bindService` and offers a client-server interface that allows multiple components to interact with it. A Bound service runs as long as a component is bound to it and when the last component is unbound it is destroyed by the system.

Services can actually be both Started and Bound, and this is determined by the implementation of the required callback methods.

- onCreate
- onStartCommand
- onBind
- onUnbind
- onRebind
- onDestroy

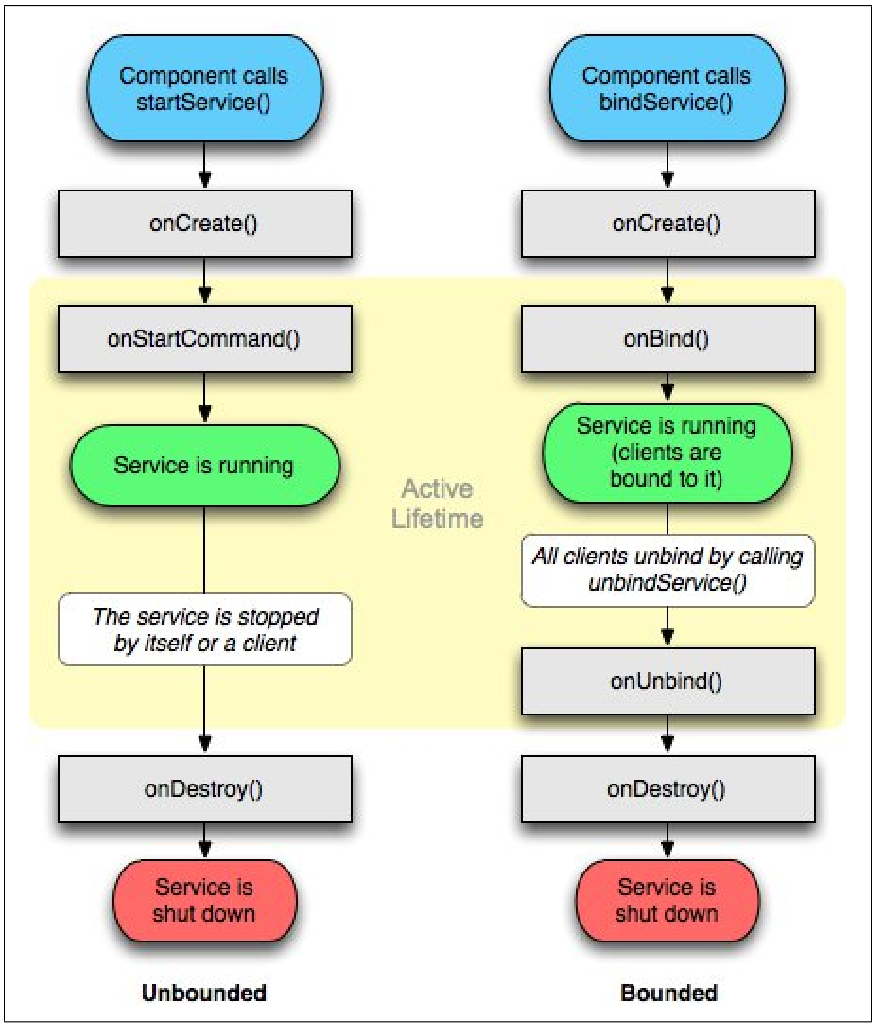
Component calls startService()
Component calls bindService()
onCreate()
onCreate()
onStartCommand()
onBind()
Service is running
Service is running (clients are bound to it)
Active Lifetime
All clients unbind by calling unbindService()
The service is stopped by itself or a client
onUnbind()
onDestroy()
onDestroy()
Service is shut down
Service is shut down
Unbounded
Bounded

Figure 4.6: The Android Service lifecycle, source Android Developers [2].

`onCreate` is called for both Started and Bound services and does the setup for the service. If the service is Started, `onStartCommand` is called next. In the `BusPositionService` this method sets up a `Runnable` which reads the bus positions from the service.[1] If the service is Bound `onBind` is called instead of `onStartCommand`. This method must return an `IBinder` which the bound components use to call methods on the service. `BusPositionService` returns `null` for this method as it is not a Bound service. When all clients have unbound from the service, `onUnbind` is called. If the service is also a Started service, `stopSelf` should be called here to explicitly stop the service. If not, `onUnbind` can return true which indicates

[1]The BusPositionService also provides an implementation of the onStart method to provide backwards compatibility with systems running a version of Android previous to API level 5. This method has since been deprecated and replaced with onStartCommand.

that clients may rebind to the service, at which stage `onRebind` will be called instead of `onBind`. If the service is not a Started service the default implementation of `onUnbind` may be used and there is no need to call `stopSelf` as the system will do this automatically. `onDestroy` is called for both Started and Bound services and should release all remaining resources for the service. In `BusPositionService`, this removes any callbacks from the Runnable created for the service.

The Content Provider lifecycle is mainly managed by the android system rather than though another component. When a request is made via a `ContentResolver` the system inspects the URI to determine the correct ContentProvider to query. If this is not running, an instance of it is created and this same instance is used for future requests. When implementing the ContentProvider, setup should be done in the `onCreate` method which is run on the main thread. The remaining methods that are to be implemented may be accessed by any thread so the developer must be careful to be sure that their implementation is threadsafe.

A BroadcastReceiver is only valid for the duration of the `onReceive` call, and once this is complete it is no longer considered active by the system. BroadcastReceivers may be explicitly declared in the application manifest or registered from within another component, for example, `BusStopsActivity` registers a receiver for the bus position updates in `onResume` and unregisters it in `onPause`.

4.5 Summary

This chapter has covered the aspects of Android and iOS application development that a developer should take into consideration when designing their mobile application. The architectures of both platforms are different, however, many of the patterns that apply are the same. Model View Controller features strongly on both along with the use of delegation, observation and notifications to name but a few. The structure and lifecycle of Android and iOS applications are very different, and the developer must take care to implement the required methods correctly. Android applications are built using Activites, Services, ContentProviders and BroadcastReceivers and each of these has its own particular lifecycle. iOS applications consist of UIApplicationDelegates, UIViewControllers and UIViews and have a completely different lifecycle to an Android application. In particular the developer must be careful not to waste system resources and to create and destroy objects at the appropriate points, as inefficient use of resources will lead to poor performance as applications run out of memory.

Once all these areas have been mastered, the next challenge for the developer is the implementation of the required features for that platform, and this is covered in the following chapter.

Chapter 5: **Features of Mobile Applications**

A number of features are to be expected in mobile applications today. Among these are menus, the ability to run in the background, data storage, access to remote services, location based services and security. Not all are applicable to every application, but a number of these will most likely be found in any application.

This chapter goes into detail on the implementation of each of the features described above on both the Android and iOS platforms. Built-in support is provided for some features, and some require considerable effort on the part of the developer. Section 5.1 describes the implementation of menus in a mobile application. This is followed by section 5.2 which provides details on what is required for an application to make use of multithreading and concurrency. Section 5.3 describes the various storage options available to the developer, and following on from this, section 5.4 describes the use of external services by an application. Section 5.5 investigates the location based services supported by Android and iOS. Section 5.6 takes a look at the security models used by both platforms and the responsibilities of the developer with regard to this. The chapter is closed with 5.7 provides a summary of the features investigated in this chapter and is accompanied by table 5.1.

5.1 Menus

All Android devices have a menu key so every application can take advantage of this fact and every activity can have an options menu if desired. Like other user interface items, menus can be declared in XML or in code. The recommended way to create an options menu is by adding a menu XML file to the `res/menu` folder and inflating this in code by implementing the `onCreateOptionsMenu` method in an Activity. Similarly the code for determining what happens when a menu item is selected should be implemented in `onOptionsItemSelected`. An example of this can be seen `main.xml` in the LiveBus Android application and in the corresponding method implementations in `BusStopsActivity`. Menu items can also be modified at runtime and this is done with the List and Map menu items in the `BusStopsActivity`, where only one of these menu items should be available at any time and this is determined by whether the list or map view is visible.

With the release of Android 3.0, there have been a number of changes to how menus are supported. Primarily support for an Action Bar has also been added, which means that menu items are added to an Action bar across the top of the Activity and are accessible directly from here or from its overflow menu. In addition to this, an `onClick` attribute has also been added for each menu item, meaning that the developer does not have to implement `onOptionsItemSelected` but can define the method to be called when each menu item is clicked.

Another type of menu available in Android is the context menu. This is displayed when a user performs a “long press” on a user interface item, e.g. an item in a list. The menu is generally specific to the item selected by the user. Again, these menus can be inflated from XML or created purely in Java code, and the `onCreateContextMenu` and `onContextItemSelected` methods must be implemented to handle this.

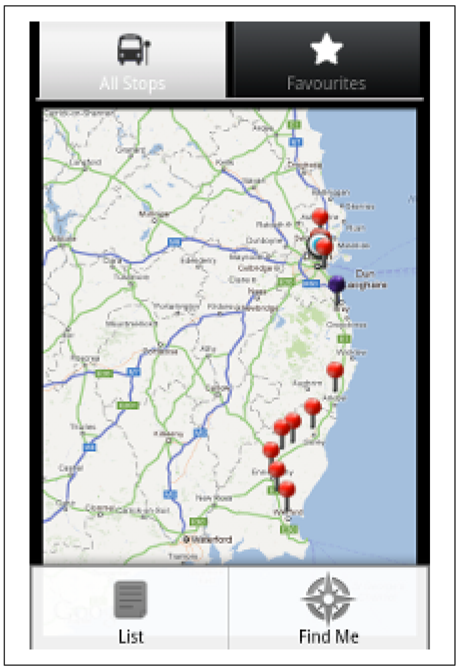

Figure 5.1: Screenshot of the LiveBus application menu

Android also offers more advanced features such as submenus, checkable menu items, menu groups, shortcut keys, and the dynamic addition of menu items to launch other activities based on intent options. Menus are a very flexible and powerful resource available to a developer.

Custom menus have only been supported in iOS application development since iOS 3.2. Previous to this, a Cut, Copy and Paste menu was available in text fields but no other options were supported. This has been extended to allow developers to use menus in other places and to add their own menu items. An instance of `UIMenuController` is obtained through the `sharedMenuController` method and custom menu items are created and added to this. The code below is an example of adding a single custom menu item to the menu. and might be executed as part of a gesture recogniser or touch handler.

```
[self becomeFirstResponder];
UIMenuItem *menuItem =
    [[UIMenuItem alloc] initWithTitle:@"Change Color"
                           action:@selector(changeColour:)];
UIMenuController *menuController = [UIMenuController sharedMenuController];
[menuController setTargetRect:self.frame inView:self.superview];
menuController.arrowDirection = UIMenuControllerArrowLeft;
menuController.menuItems = [NSArray arrayWithObject:menuItem];
[menuController setMenuVisible:YES animated:YES];
```

A number of additional methods must be implemented in order for the menu to appear correctly. The `canBecomeFirstResponder` method in the `UIViewController` must be implemented to return `YES`, and the `canPerformAction:withSender` method must be implemented to inform the `UIMenuController` which menu items should be enabled. In addition to this, the method specified as the menu item action must also be implemented. In the example above this is the `changeColour` method.

An alternative which is supported for iPad applications is the use of `UIPopoverController`. The `UIPopoverController` may contain any type of view, so the container view may be implemented as a menu style view if the developer requires it. This is commonly used with `UISplitViewController`. A `UISplitViewController` contains two `UIViewControllers` which are displayed side by side when the device is in landscape orientation and uses a popover for the first `UIViewController` when the device is in portrait orientation. The two views in the `UISplitViewController` are generally nested inside `UINavigationControllers` as this makes it easier to add titles and bar buttons. The `UIPopoverController` manages a custom view which is tied to a bar button item and hovers over the main view when the iPad application is in portrait mode. When the iPad is in landscape mode the view previously displayed in the popover is shown in the left hand view of the split view. A number of `UISplitViewController` delegate methods must be implemented to show or hide the popover as appropriate and to do any extra processing that may be required.

If the popover is required for both portrait and landscape, then `UISplitViewController` should not be used. The `UIPopoverController` is intialised with a `UIViewController` to display the contents and the popover can be presented from any user interface item using the `presentPopoverFromBarButtonItem:permittedArrowDirections:animated` or `presentPopoverFromRect:inView:permittedArrowDirections:animated` methods. The size of the popover can be set directly or returned from a method implemented in the popover's `UIViewController`, and the popover is dismissed either when the user clicks outside the popover, or programatically in code, e.g. in response to a user selection.

Android provides a much simpler way for developers to add menus to their applications. The ability to specify these in XML also creates a clear separation of user interface and application logic. There is no way for a developer to define menus outside of the code in iOS application development, so the `UIViewController` can become cluttered with code for both presentation and application logic. Buttons can be added to navigation bars to open up additional views but these may become cluttered especially on iPhones and iPod touch devices if more than a couple are required. Android manages this in the action bar by automatically adding an overflow menu to prevent it from becoming overcrowded.

5.2 Multi-threading and Concurrency

Multi-threading and concurrency is very important on mobile devices as users are most likely to uninstall unresponsive or slow applications. This is something that needs to be considered early in the design phase for mobile applications. Multi-threading and concurrency adds considerable complexity to application logic, and different solutions may be required for the different problems to be solved. One important task for the developer is to correctly identify the concrete units of work that can be done asynchronously, and any dependencies between these tasks.

iOS provides a number of ways in which multi-threading and concurrency can be executed. Grand Central Dispatch (GCD) is the preferred technology for this as it provides an asynchronous approach, with thread management code hidden at the system level. The developer simply has to define the tasks that they want to be run asynchronously and the dispatch queue that they should be run on. Dispatch queues are a C-based mechanism which executes on a first-in first-out basis, either serially (one task in queue at a time) or concurrently (many tasks simultaneously). Tasks in this case are either blocks (see section 3.6) or functions. Dispatch queues are preferred to threads as they are more simple and more efficient than the corresponding thread code. There are three types of dispatch queues: private dispatch queues (one task runs at a time in order of insertion), global dispatch queues (multiple run tasks concurrently, in order of insertion), and the main dispatch queue (tasks run on main thread, interleaved with other execution on application run loop). With GCD and dispatch queues, the system manages threads more efficiently than the developer can, scaling dynamically based on system conditions and available resources. An example of the use of dispatch queues and blocks can be seen in the iOS LiveBus application in the `loadPickups` method in the `BusStopViewController` implementation. A serial queue is created and a block to read the pickup times is added asynchronously to this. Once these have been read from the web service, the block uses the main queue (the user interface queue) to display the pickup times.

```
// read and add pickup times in another queue
dispatch_queue_t pickupsQueue = dispatch_queue_create("PickupsQueue", NULL);
NSString *busStop = self.busStop.name;
dispatch_async(pickupsQueue, ^{
  NSDictionary *routesAndPickups =
      [TextMyBusServiceHelper getPickUpsForBusStop:busStop];
  [routesAndPickups retain];

  // go back to main queue
  dispatch_async(dispatch_get_main_queue(), ^{
    // display the pickup times
  });
});
dispatch_release(pickupsQueue);
```

Similar to dispatch queues are dispatch sources, which process specific types of system events asynchronously. Again blocks or functions are used to define the code that should be executed when the particular system event occurs. Dispatch sources can be used for timers, monitoring signals, file and socket based operations, process events, Mach related events, and custom events. When configuring a dispatch source, the developer must specify the event to be monitored, the dispatch queue to use and the event handler code to be executed when the event is triggered.

Also available to the developer are operation queues which are Objective-C objects that act like dispatch queues. A `NSOperationQueue` handles scheduling, execution and all thread management and allows for dependencies between tasks which can be used to determine the order of task execution. Operation queues use key-value observing (KVO) notifications which allows the developer to monitor task progress. Task execution in operation queues is usually concurrent, although serial execution is possible with the configuration of dependencies between tasks. Tasks added to the operation queue are subclasses of `NSOperation`. Two subclasses are provided which are simple to use; `NSInvocationOperation` which is based on a given object and selector, and `NSBlockOperation` which executes a given block object. Developers can also provide a custom implementation of `NSOperation`, although this is much more complex. Some of the advantages of using operation queues are the dependency graph for operations, the optional completion block, the use of KVO notifications, the ability to set operation priorities and the ability to cancel operations.

If none of the options described previously are desirable to the developer they can still use threads directly. This, however, places the burden on the developer to implement their threading solution in an efficient and scalable way. Within threads itself, there are two different approaches available for iOS development; Cocoa threads and POSIX threads. Cocoa threads use `NSThread`, and all objects which extend `NSObject` can spawn threads to execute any of their functions. POSIX threads are a C based thread interface and may be considered by the developer if their code is to be reused across applications and platforms. This provides a simple and flexible API. With both of the approaches above the developer is responsible for managing thread creation and termination, communication and memory usage.

In Android a single thread is used for executing each application. This thread handles all application code from background processing to user interface updates. Due to this, long running background tasks may make applications unresponsive as they cannot redraw or respond to user input while running a background task. If an Application is unresponsive for more than 5 seconds, an application not responding method is displayed. Android provides a number of ways in which a developer can implement a solution to this. Standard Java threads can be used to perform intensive operations, however, they are not permitted to update the user interface. Android have provided a number of methods on the Activity and View class to handle this, however, this code can quickly get complicated as you end up with nested `Runnable` objects.

A version of this can be seen in the Android LiveBus application in `BusPositionService`. Here the code to read and store bus positions is executed in a `Runnable`, however, the service does not have to worry about displaying this information on screen. On completion of the task a broadcast is performed so any listening activities know that the positions have been updated and should be redrawn. Additionally a `Handler` is used here to reschedule the `Runnable` at 30 second intervals and this is repeated until the service is destroyed.

An alternative to the use of Java threads is `AsyncTask`. This allows the developer to perform asynchronous tasks on the user interface thread, with the work being done on a separate thread, and the results being published on the user interface thread when execution is complete. This is used by subclassing `AsyncTask` and implementing the required methods. An example of this can be seen in the `BusTimesActivity` where the inner class `BusTimesTask` extends `AsyncTask` and is used to read the pickup times from the web service, at the same time allowing the user to access the map displaying the bus stop. The `onPreExecute` method is used to inform the user that pickup times are being read. The `doInBackground` method reads the pickup information from the web service, and following its completion, `onPostExecute` is called which causes the pickup times to be displayed if any were found. Additional methods also exist to allow the developer to monitor and control the execution, allowing for cancellation of the task if needed.

```
class BusTimesTask extends AsyncTask<Void, Void, BusRouteListAdapter> {

  @Override
  protected BusRouteListAdapter doInBackground(final Void... params) {
    // read the pickups and return the adapter
  }

  @Override
  protected void onPreExecute() {
    // Tell user we are reading pickups
  }

  @Override
  protected void onPostExecute(final BusRouteListAdapter adapter) {
    // Display the result from the adapter
  }
}
```

iOS provides many options to the developer for handling multithreading. Objective-C developers may prefer the use of `NSOperationQueue`, whereas a developer more familiar with languages such as Ruby or Python may prefer the use of GCD and blocks. The developer may also use Threads directly. Apple recommends the use of GCD which is simple to use, provided the developer has a good understanding of blocks. The code for completing the task is included in the block passed to the background queue, and within this another block is used to update the user interface on the main queue. Android provides two options for multithreading; the use of `AsyncTask`, and the direct use of `Thread` or `Runnable`. Android recommends the use of `AsyncTask` as it is simple to use and encapsulates both the code to be run in the background and the code to update the user interface. The implementation of multithreading on both platforms is comparable, with neither one having a clear advantage over the other, as both are easy to use and perform well.

5.3 Storage

Android provides a variety of options for application data storage. For simple data such as user preferences, the SharedPreferences framework may be used. This is used for storage of simple key-value pair data such as user preferences, and persists even after the application has been closed. Android further supports this by letting the developer declare their preference screens in XML, and provides the `PreferenceActivity` or `PreferenceFragment` superclasses for the implementation of activities to manage this. [1]

Android applications may also store their data in files; either internally on the device storage or externally, e.g. on a SD card. Android includes the `java.io` package with a wide range of different file readers and writers. Files stored internally are restricted to access by the application only and are not even accessible to the user. More details on this can be found in section 5.6 on security. This data is removed if the application is uninstalled or if the user chooses to clear data in the Android Applications Manager. External storage is also available and this may be on removable storage, or alternatively in a shared storage location on the device itself. This storage is not private and data may be accessed, modified or removed by any application or by the user. Android provides a number of methods in the Environment class for checking the status of the storage area itself, and of data that resides there. If static files are required to load application data, these can be stored in the `res/raw` directory at compile time and loaded using the generated resource ID. These files can be read using an `InputStream` but cannot be used to store data by the application. An example of this can be seen in the TextMeMyBus activity where the bus route information is read from a static XML file supplied as a raw resource with the application.

```
final InputSource inputSource = new InputSource(
    getResources().openRawResource(R.raw.busroutes));
```

Database support in Android is provided through the use of SQLite. Any databases created by an application are accessible by any class in it, but by nothing outside of it. Database access is generally implemented using a subclass of `SQLiteOpenHelper` to get an instance of the requested database. The returned `SQLiteDatabase` object allows the developer to access and update the data, with a `Cursor` being used to access values returned from database queries. SQLite usage can be seen in the Android LiveBus application in `BusStopDBAdapter`, and the `busStopFromCursor` and `busStopListFromCursor` methods in the `BusStop` class show how to retrieve data from a cursor. More complex queries can be performed using the additional parameters on the `SQLiteDatabase query` method. An example of this can be seen in `BusStopDBAdapter` where the favourite bus stops are read.

[1]Prior to Android API 11, PreferenceActivity was used to implement activities to display and edit user preferences, since API 11 the use of PreferenceFragment and preference header XML resources is preferred.

```
public Cursor fetchFavouriteBusStops() {
  return mDb.query(DATABASE_TABLE, new String[] { KEY_ROWID, KEY_CODE,
      KEY_LATITUDE, KEY_LONGITUDE, KEY_FAVOURITE }, "favourite=1",
      null, null, null, null);
}
```

The developer also needs to consider how to share data within an application. In the LiveBus application, a large amount of data is shared among the various activities and services. The `BusPositionService` is responsible for reading the bus position data from the web service, but this is needed by the `BusStopsActivity` and `FavouriteBusStopsActivity`. This data is shared through the use of the singleton `BusPositionStore`, and an `Intent` is broadcast to inform the activities every time this data is updated. Intents can also be used to pass data between activities. When a bus stop is selected from the list, an intent is created to launch the `BusStopActivity`. The busStopID, busStopCode, latitude and longitude are passed as extras to this intent, and `startActivity` is called. In the `onCreate` method of `BusStopActivity`, these extras are available as a `Bundle` from the `getIntent().getExtras()` call. Similar to this is the `startActivityForResult` method, which is used when one activity requests data from another. When the child activity exits, it calls `setResult` to return data to its parent, and the result is received in the parent activity's `onActivityResult` method. If more complex data needs to be passed between activities, it can be done by passing weak references to the data in an intent. For example, if an `Application` class is implemented, it can hold a central map of data. When an activity wants to share data with another activity, it stores this in the map, and sends the key to the data to the other activity. The second activity also has access to the application class and to the map so can read the shared data using the key. Another option would be to use public static fields or methods in an application as these are accessible to all classes in the application.

For sharing data with other applications, there are a few options. One such option is the use of content providers which are discussed in section 4.4. As mentioned previously in this section, the external file storage may also be accessed by other applications and so could potentially be used to share data. Intents can also be used to pass data between applications as they can be used in Android to start activities in other applications, e.g. an application can launch the camera activity to take a photo, and this will return the location of the saved photo to the parent application on return.

iOS also provides a comparable number of options for data storage. For simple data such as user preferences, the `NSUserDefaults` object is recommended. This allows for key-value data storage and is very simple to use. For slightly more complex data where use of a database is not necessary, `plists` can be used. A plist file is a structured file which uses key-value storage and can be used to stored serialized data. In iOS development this is stored in XML format, and files can be read and written to, using the methods available in the `NSPropertyListSerialization` class. Related to this is data archiving. This allows the developer to store any object which implements the `NSCoding` protocol to an archive, and this may be unarchived and the object restored at a future point. This may be useful, e.g. for restoring application state when it is restarted. Static documents may also be read from the application bundle, but these are strictly read only and must be copied to another directory if they are to be updated by the application.

```
NSString *filePath = [[NSBundle mainBundle] pathForResource:@"myBusRoutes"
                                                    ofType:@"xml"];
BusRouteXMLParser *busRouteXMLParser = [[BusRouteXMLParser alloc] init];
[busRouteXMLParser parseXMLFile:filePath];
```

Applications can only write documents inside their own sandbox, e.g. each application has its own documents, cache and tmp directories. Files stored in the documents directory can be shared via iTunes. Files in the cache directory are not shared but may be removed by the system in times of low resources, and files in the tmp directory are not shared and should be removed by the application when no longer needed so as not to waste device resources. NSFileManager provides utility methods to allow access to the filesystem, though most of the functionality for reading and writing files is found in classes such as NSData or NSString. Since iOS 5 there is support for a document-based application approach which may be a better option for developer if the whole application is heavily document based, e.g. a text editor application which allows the user to create and manage documents. This is implemented by creating a custom subclass of UIDocument to represent the document data, and by providing a recommended set of functionality to allow the user to list, view, create, delete, and perform other document management functions. The document itself can be stored as a file on the device, on iCloud or in a database depending on the application design.

When it comes to database storage in iOS, the Core Data framework is the preferred option. Xcode provides a core data application template and the developer can define their data model using a graphical editor and generate code to form the basis of their custom data model objects. Entities and relationships can be modelled graphically, and the generated code takes care of the background relationships, saving the developer from having to worry about writing complex SQL. Core data is built on top of SQLite. In the LiveBus application, the Core Data framework is used to store information on the Bus Stops. In BusStopsTableViewController the bus stops are read from the SQLite database in the initWithManagedObjectContext method. This uses an NSFetchedResultsController to read all records from the BusStop entity. Much of the code to manage the core data objects is contained in the CoreDataTableViewController implementation, which is the superclass of both BusStopsTableViewController and FavouriteBusStopsTableViewController. The use of predicates in requests saves the developer from having to write SQL queries as can be seen in initWithManagedObjectContext in FavouriteBusStopsTableController.

```
- initWithManagedObjectContext:(NSManagedObjectContext *)context
{
  if (self == [super initWithStyle:UITableViewStylePlain]) {
      NSFetchRequest *request = [[NSFetchRequest alloc] init];
      request.entity = [NSEntityDescription entityForName:@"BusStop"
                                   inManagedObjectContext:context];
      request.sortDescriptors = [NSArray arrayWithObject:
          [NSSortDescriptor sortDescriptorWithKey:@"name" ascending:NO]];
      request.predicate =
          [NSPredicate predicateWithFormat:@"favourite == YES"];
      NSFetchedResultsController *frc =
          [[NSFetchedResultsController alloc] initWithFetchRequest:request
                                              managedObjectContext:context
                                                sectionNameKeyPath:nil
                                                         cacheName:nil];
      [request release];
      self.fetchedResultsController = frc;
      [frc release];
      self.titleKey = @"name";
  }
  return self;
}
```

If the developer does not wish to use the core data framework, the SQLite libraries are also available, and allow the developer to access and manipulate the database and its tables directly. This approach requires the developer to write the SQL for queries and updates rather than having it generated in the background.

Within an iOS application, data can easily be shared. When launching one view controller from another, the first view controller creates an instance of the second view controller, sets a property with the required data for the second view controller, and pushes the second view controller on to its parent navigation controller's stack for display. This can be seen in the `managedObjectSelected` method of `BusStopsTableViewController`. This method is invoked when a bus stop is selected from the list and the `BusStop` managed object is set as a property on the `BusStopViewController` before it is pushed onto the navigation controller for display. All view controllers within an application also have access to the SQLite databases for an application, either by using the core data framework or by using the SQLite databases directly. As with Android a singleton class can be used to centrally store more complex data, or the data can be shared in an object such as the `ApplicationDataStore` which is stored on the application delegate in the LiveBus application. This is accessible to all view controllers in the application and could be used to share many data objects.

For sharing data with other applications there are very few options. There is no shared storage on an iOS device so there is no option to share files between applications. An alternative could be to share files via a remote server, e.g. a custom server or to use a service such as Dropbox. Services such as iCloud can be used to share data among instances of an application, e.g. the user has an iPhone and an iPad with the application installed so data is accessible to both. iOS applications can also provide custom URL schemes to accept data from other applications. For example Apple provides custom URL schemes to deal with mailto links, so that when an email link is clicked on in an application, it can request the services of the mail application to deal with this. At this point control passes to the second application and there is no way to pass back data to the calling application. The `UIDocumentationInteractionController` uses a similar concept to this to handle the management of document types that it does not understand. For example an application displays a list of files downloaded from the internet, when the user selects a document, the application can pass off responsibility for displaying this to another application using the `UIDocumentInteractionController`. This knows what applications can handle certain document types, and can present the user with a list of applications that can open the selected file.

Both platforms provide a number of options for data storage. The use of `NSUserDefaults` and `SharedPreferences` for storage of simple user data is comparable. However, Android makes this even simpler by allowing the developer to simply declare preference screens in XML and use a `PreferenceFragment` to automatically save changes as the user interacts with the screen, without needing to write additional code to handle this. Both platforms use SQLite for database storage, however, iOS provides an additional Core Data layer on top of SQLite. With this, the developer gets a graphical model of their data and generated code to handle the database access. This can be useful for the developer to visualise the relationships between data stored in the SQLite database, and removes the need to write complicated SQL. This is a clear plus for a developer who does not have a lot of exposure to SQL, and Android has nothing comparable. Where iOS falls down, however, is in its restrictions on data sharing. Android allows shared access to files (in certain areas of the filesystem), and the exchange of data between activities or applications with the correct permissions.

5.4 Web Services and XML

Quite often mobile application developers need to make use of remote services in their applications. The web service used in the LiveBus application is a Simple Object Access Protocol (SOAP) based web service which, unfortunately, does not have a lot of support in either iOS or Android. The use of Representational State Transfer (REST) based web services and JavaScript Object Notation (JSON) seems to be the preferred option for mobile operating systems as neither operating system provides native SOAP libraries but both do for JSON. REST based web services overload the HTTP protocol to locate and update resources on a remote service based on the URI and the HTTP verb used. Data is usually returned in the JSON format which is compact and relatively simple to parse. Apple provides the NSURL APIs to handle the remote communications and in iOS 5 introduced the `NSJSONSerialization` class to handle the response. Android provides the `org.apache.http` package for handling the communication with a web service and the `org.json` package for parsing the responses.

The web service used by the LiveBus applications is a SOAP based web service. Both the Android and iOS applications use third party APIs to handle the HTTP communications and SOAP messages, as well as requiring custom parsers to handle the XML data returned.

The wsdl2objc [54] tool was used for the iOS application to generate Objective-C code from the Web Service Definition Language (WSDL) for the LiveBus web service. This generates Objective-C objects to handle the HTTP communications and the SOAP requests and responses. The majority of this generated code can be found in `TextMeMyBusServiceSvc` and is used in the `TextMeMyBusServiceHelper` implementation. All functions on the web service are called synchronously and an example of this can be seen in the `getCurrentBusPositions` method. This is called from `reloadBusPositions` in `BusStopsTableViewController`, but is run in a dispatch queue using GCD as described in section 5.2. Once the data from the web service has been returned and parsed, control returns to the user interface thread where the map is updated with the latest bus positions.

```
+ (NSArray *)getCurrentBusPositions
{
  TextMyBusServiceSoapBinding *binding =
    [[TextMyBusServiceServiceSvc TextMyBusServiceSoapBinding] retain];

  TextMyBusServiceServiceSvc_getCurrentBusPositions *params =
    [[TextMyBusServiceServiceSvc_getCurrentBusPositions alloc] init];

  TextMyBusServiceSoapBindingResponse *response =
    [binding getCurrentBusPositionsUsingParameters:params];
  [params release];
  [binding release];
  NSString *data = [TextMyBusServiceHelper getDataFromResponse:response];

  // parse the returned data
  BusPositionXMLParser *xmlParser = [[BusPositionXMLParser alloc] init];
  [xmlParser parseXMLString:data];

  NSArray *result = xmlParser.busPositions;
  [busPositionXMLParser release];
  return result;
}
```

The ksoap2-android libraries [32] were used in the implementation of the Android application. A java tool, similar to wsdl2objc, that worked with the WSDL from the web service could not be found, but the use of the ksoap2 libraries proved much simpler. This can be seen in the `getDataFromWebservice` method in `WebServiceHelper`. This makes a synchronous call to the web service using the given operation name and parameters, and returns the data as a string. Because all calls to the web service are synchronous, they are not done in the main thread. The `BusPositionService` performs a call to the web service in a `Runnable` which sends a broadcast when new data is available. This broadcast is listened for by the `BusStopsActivity` in the user interface thread and the map is redrawn with the new bus positions when the correct broadcast is received.

```
public String getDataFromWebservice(final String operationName,
    final Map<String, Object> propertyMap) throws Exception {
  // create the soap request
  final SoapObject request = new SoapObject(WSDL_TARGET_NAMESPACE,
      operationName);

  // add any request parameters
  if (null != propertyMap) {
    final Set<String> keySet = propertyMap.keySet();
    for (final String key : keySet) {
      request.addProperty(key, propertyMap.get(key));
    }
  }

  final SoapSerializationEnvelope envelope = new SoapSerializationEnvelope(
      SoapEnvelope.VER11);
  envelope.setOutputSoapObject(request);

  final HttpTransportSE httpTransport = new HttpTransportSE(SOAP_ADDRESS,
      DEFAULT_TIMEOUT);
  httpTransport.call(SOAP_ACTION, envelope);
  final Object response = envelope.getResponse();
  return response.toString();
}
```

As well as the generated code for the SOAP communications, XML parsers were required for parsing the data returned from the web service. As the returned data simply needed to be parsed once and not modified in any way, a Simple API for XML (SAX) parser [48] was chosen. SAX parsers are event-driven, and call out to specific handler methods whenever certain XML elements are encountered. A major advantage of using a SAX parser in a mobile application is that it requires less memory than Document Object Model (DOM) parser [18], as it only deals with one XML construct at any time. DOM parsers use a tree-based model which loads the whole XML tree into memory at once.

Apple provides the `NSXMLParser` and `NSXMLParserDelegate` classes for event-driven XML parsing. Each operation on the service required a separate parser to handle the returned data. This was done by implementing the `NSXMLParserDelegate` protocol in each parser class and required the implementation of the `didStartElement` and `didEndElement` methods. This can be seen in the `BusPositionXMLParser` implementation which also includes sample XML. Each parser extends from a custom parser class, `TextMeMyBusServiceXMLParser`. This class contains the common code required for all parsers, for example, methods to parse from a file or string, and the `foundCharacters` and `parseErrorOccurred` methods.

For the Android application, individual parsers were again needed for every operation called on the web service. As before event-driven parsers were the preferred option. Each custom parser extends the `DefaultHandler` superclass and provides an implementation for the `startElement`, `endElement` and `characters` methods. An example of this can be seen in the `BusPositionXMLHandler`, which is used by the `readBusPositionsRunnable` and the `loadBusPositionsFromXML` method in the `BusPositionService`.

The lack of support for SOAP on both platforms is striking, although this is most likely Apple and Google trying to encourage use of the more efficient Restful web services and JSON. The third party libraries used in the Android and iOS applications proved easy to use, despite the different implementations for each one. The ksoap2-android libraries will most likely prove the easiest to maintain in the long run, as any changes to the WSDL for the web service will require the developer to rerun wsdl2objc in order to regenerate all the required classes. The was no real difference in terms of the parsers required for both applications, as both were SAX based and so similar methods were implemented for both platforms.

5.5 Location Based Services

With the advent of GPS in mobile devices, location based services (LBS) have become very important, both to users and to vendors. There are many different geo-location technologies available for mobile LBS some of which are described by Rao and Minakis [44]. The main technologies that application developers make use of are GPS, assisted GPS and cell identifier location information. GPS and assisted GPS provide fine detail on the user's location, but are restricted in that the user must be outside, and these can place a higher drain on the device's battery depending on the frequency and granularity of updates required. Location based on the identification of mobile network cells provides much more coarse information on the user's location as this is calculated based on the location of the mobile network base station that the device is currently using. Location information is useful to the developer, not just for mapping or navigation applications, but also enables other services such as targeted advertising based on location. Care must be taken, however, when dealing with user location information as this is sensitive data which can be abused. In April 2011 it was reported that iPhones were recording and backing up the device location without the user either being aware of it or giving permission. However Apple was simply maintaining a cache of Wi-Fi hotspot and cell towers around the user's location, and this was mistakenly being backed up by iTunes. Apple was forced to issue a statement to reassure users of their intentions, and to issue a software update which fixed a number of bugs in this area [16].

Within iOS application development there are a number of frameworks available to developers. The CoreLocation framework provides most of the functionality and developers can leverage this by implementing the `CLLocationManagerDelegate` protocol. This requires the developer to implement `locationManager:didUpdateToLocation:fromLocation` and `locationManager:didFailWithError`. The first method provides the delegate with the new location every time one is received. It is up to the developer to determine if this location is new enough, and if the accuracy is sufficient. The developer can use this, for example, to update an annotation on a map to show the user's current location. The second method is used to handle errors when location updates fail. This may be because the user has disabled location updates for the application, or simply because the device is unable to get a fix on the current location, for example, if it is indoors. Apple recommends that developers check if location updates are enabled for the device and are authorised for the application before using this in code; it could be extremely annoying for the user who has intentionally disabled location services for whatever reason, to be repeatedly requested by an application to re-enable them. Developers should also be careful to only request location updates when they are really required and to turn these off when they are no longer needed. All this can be managed by calling methods on the `CLLocationManager` class.

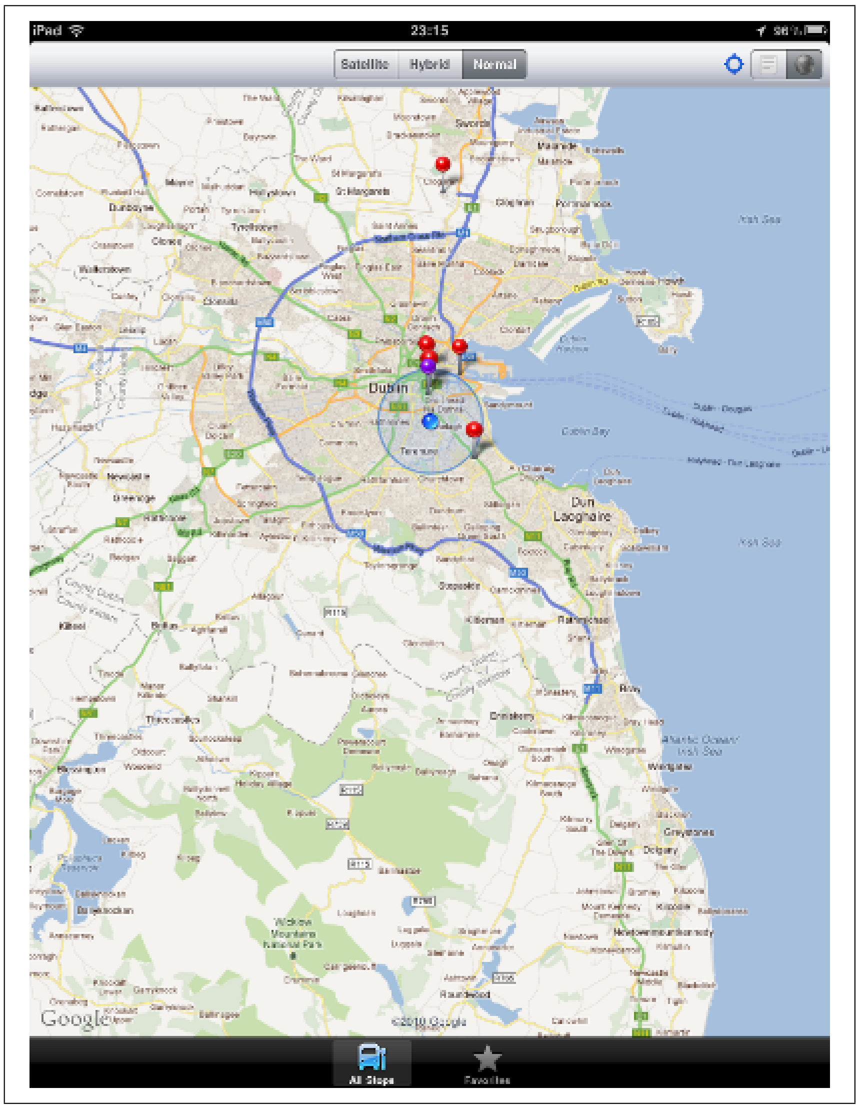

Figure 5.2: Screenshot of the LiveBus iPad application showing the user location

For displaying location information to the user, Apple provides Google Maps integration through the MapKit framework. This allows the developer to specify which region of the map should be displayed, the level of zoom, and any annotations, etc. that may be required. The developer can add a `MKMapView` to an existing view either by using Interface Builder or in code. A number of configurations are possible such as whether zoom controls are enabled, whether the map should pan, whether the map type is default, hybrid or satellite, and whether the map should display the user's location. The `MKMapView` has its own location manager that may be used to display the user's location if the functionality of the CoreLocation framework is not required. This is enabled by simply setting the property `showsUserLocation = YES` on the map view. Usage of this can be seen in the `findMe` method in `BusStopsTableViewController` which uses the user location from the map view to zoom to the user's location on the map. The functionality of the CoreLocation framework is still needed here, however, as there is no other way to check whether location services are enabled. Custom annotations can also be added to the map and this is done for both bus stops and bus positions in the `BusStopsTableViewController`. This is done by implementing the `mapView:viewForAnnotation` method and returning custom views for these types of annotations. In `BusStopsTableViewController`, different pins are used to display bus stops and bus positions, and bus position annotations include the Wexford Bus logo, whereas bus stop annotations have a callout accessory view. When this callout accessory view is clicked, the `mapView:annotationView:calloutAccessoryControlTapped` method is called and in the case of bus stops, is used to call a method to push the `BusStopViewController` to the navigation controller and display the bus stop information.

Location Services are provided as a system service by Android and an instance of the `LocationManager` can be obtained with `getSystemService(Context.LOCATION_SERVICE)`. This provides access to the `LocationProviders` for the device and allows an application to register for updates to these. Developers can register for updates from more than one provider, for example, the GPS provider and the Network provider, but this will only work if the application has the required permissions to access each. A `LocationListener` implementation handles the returned updates, and it is again up to the developer to determine if the update is new enough and accurate enough to warrant action. This can be seen in the `DeviceLocationListener` inner class in `BusStopsActivity`. The developer must also be careful to specify a location frequency that is not excessive and to remove location updates when they are no longer required so as to not drain the device's battery.

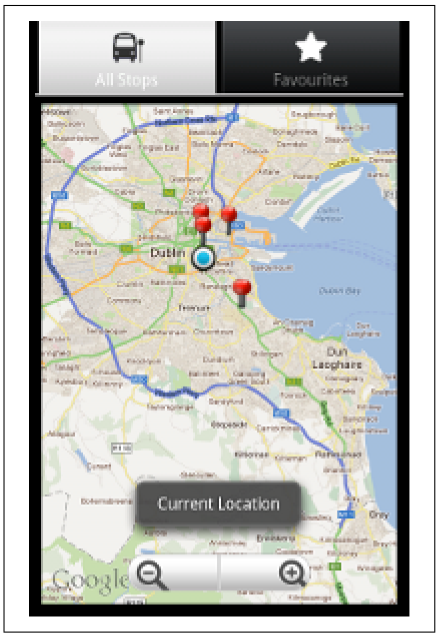

Figure 5.3: Screenshot of the LiveBus Android application showing the user location

In order to display the user's location on a map in Android, the developer must include the Google Maps API, for which they must obtain a license key. This differs to iOS, where the developer does not have to obtain a license key, but still has to abide by the Google Maps terms and conditions. With Android development, a debug key can be used until the application is ready for deployment. The map view can be specified in an XML layout file, or added in the code. The developer has a number of options as to how the map is configured, what area is displayed, what the zoom level is, etc. In the `BusStopsActivity`, bus stops and bus positions are represented by `OverlayItems` drawn on the map. `BusStopOverlayItem`, `BusPositionOverlayItem` and `CurrentLocationOverlayItem` hold the details of the relevant point of interest, and the `DrawableItemizedOverlay` class handles the management of the overlay items on the map, and what happens when on of these is tapped on. In the case of a bus position, a `Toast` is displayed with the relevant information, and in the case of a bus stop, a `Dialog` is displayed which allows the user to open the `BusStopsActivity`. Google Maps in Android doesn't offer the same annotation view functionality as iOS so this is why `Toast` and `Dialogs` were used to display the information, rather than views with icons and callouts. There are also issues in Android with having more than one `MapActivity` in an application. All activities in an application, by default, run in the same process, so the map activities tend to interact in strange ways and often do not display properly. Google recommends only using one `MapActivity` per application, or running each `MapActivity` in its own process. This will not work in the case of the LiveBus application, as multiple map activities are needed, and running in multiple processes causes problems accessing the `BusRoute` and `BusPosition` data which is held in static singletons shared across the activities. Also running the activities in separate processes seems to prevent debugging except for on the main application process.

iOS and Android provide broadly the same functionality when it comes to location based services. However, there are a few key differences. Android allows the developer to specify the source of the location updates, so GPS can be used if high levels of accuracy are required, whereas the Network location can be used if less accuracy is needed. The use of GPS drains the battery on the mobile device so the ability to turn off updates is very useful. iOS does not give the developer the same options, but makes use of whatever hardware is on the device to determine the user's location, whether it is required or not. Both platforms make use of Google Maps. Surprisingly, the MapKit APIs for Google Maps integration in iOS proved much more reliable and provided more useful functionality in terms of annotation views, than the Google Maps APIs for Android.

5.6 Security

With the growing popularity in smartphones (and the vast amount of personal data they contain) has come a corresponding increase in the number of malicious applications and attacks on mobile devices. Apple and Android have differing views on how this should be handled and this can be seen in their respective operating systems, application capabilities, and application distribution.

iOS takes a sandboxing approach for each application installed on an iOS device. Every application lives in its own area, with no access to other application data, to system files, resources or to the kernel. Some access to other application data is given, but only though the iOS resource APIs so this is strictly controlled. For example, no access is given to a user's SMS. Applications can only be installed on an iOS device from the Apple App Store and every application submitted to the App Store must be approved before it is made available for distribution. In order to submit an application to the App Store, developers must join Apple's Developer program and pay a yearly fee. Every version of every application is reviewed by Apple, and the inclusion of functionality such as the dynamic generation of code, or not following Apple guidelines will result in an application being rejected. This ultimately gives Apple control over which applications may be installed on a device, so should prevent malicious applications from ever getting included in the App Store. Further to this, Apple has the ability to remotely remove malicious applications from every device they are installed on, if a malicious application does slip past Apple's inspection. More detailed information on the distribution of iOS applications can be found in chapter 6. Even the distribution of applications that are not available on the App Store is strictly controlled. Applications developed privately may only be installed on devices provisioned by Apple, and this also requires the developer or development team to be registered with the Apple Developer program and to have paid the yearly fee.

All iOS applications must be signed with a certificate issued by Apple through the developer program. At runtime, the iOS system checks the application signature to make sure that it has not been tampered with since the application was last used. Apple offers further opportunities to secure applications, by providing a number of features such as keychain services and hardware encryption. Keychain services can be used to encrypt and store sensitive data such as user usernames and passwords, and this data is only accessible to the application that created it. Apple also allows developers to make use of the hardware encryption APIs, where certain files can be locked by the system when the device itself is locked, thus making the file inaccessible by the application itself, as well as by others. A number of additional encryption APIs are also included in the iOS libraries.

Android also uses a sandboxing approach when it comes to applications. However, it also permits inter-application communication through the use of Intents, and allows access to shared areas in external storage. All of this is subject to an application having the correct permissions. Permissions are defined for functionality such as internet access, access to removable storage, location information, etc. and are stored in `AndroidManifest.xml`. On choosing to install an application, the user is presented with a list of all permissions that the application declares, and must agree to these in order that the application to installed on the device. Permissions cannot be altered after an application has been installed. The following permissions are those required by the LiveBus application:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

Applications are run on the device in their own virtual machine, and as their own Linux user. The Linux kernel handles much of the security at the process level through the use of user and group IDs and file access permissions.

Android applications also must be signed before they can be submitted to the Android Market, but the certificate is not granted by Android and developers sign their own applications with their own certificates. Every new version of an application is checked to ensure that it is signed with the same key as the original version, thus checking that it is from the same source. It is the responsibility of the developer to keep their key secure and prevent it from being used by potentially malicious sources.

Neither are the applications reviewed before they are accepted onto the Android Market. Android allows users to review and comment on applications so those that are unresponsive, buggy or do not behave as advertised will soon get negative reviews. Users may also report applications which are found to be malicious, and again Android has a mechanism which allows them to remotely uninstall any application from every device on which it is installed. Applications may also be installed from non-Market sources, but this requires the device user to explicitly set the option to allow installation from unknown sources. This allows developers to install their own applications without any restrictions, and allows for installation from a source such as the Amazon Appstore for Android. The Amazon Appstore also has a developer program that requires a yearly fee, and like the iOS App Store, applications must be approved before they are accepted for distribution.

A large proportion of the Android operating system code is publicly available and may be updated by any contributor. While some may fear that this might allow for the introduction of malicious code, in reality it allows for close monitoring by the open source community. Bugs in the operating system code are found and fixed by community members, and the fixes are verified by experienced contributors before they are accepted and merged into the public code.

Neither operating system's security is perfect as can be seen in the work of Miller [35], Egele et al [19], and Shabtai et al [47]. This can also be seen in the more regular news reports of security flaws. Because Apple reviews every application that is distributed via the App Store, iOS applications are less likely to contain malicious code than Android ones. iOS applications are very much restricted in what they have access to, however, these same restrictions prevent developers from leveraging the full functionality of the device in their applications. Android places the onus on the user to determine what an application should have access to, by requiring them to explicitly agree the permissions an application has when it is installed.

Groups such as the iOS jailbreaking community mentioned in the article above, try to promote openness in iOS application development by using these weaknesses to expose the system internals, but they also highlight security holes that malicious developers may exploit.[2] Similar communities exist for Android, with a number of rooting solutions available to users who are not happy with the software solution provided by their hardware vendor. Application developers who wish to remain part of the approved developer programs and distribution channels must be careful to steer clear of any security pitfalls. Developers need to build trust with their users, and aim to deliver secure applications that behave as described.

5.7 Summary

This chapter has covered features that most application developers will encounter in the course of developing an application for Android or iOS. There are vast differences in the implementation of all the features covered in this chapter, some with better support on one platform than on the other. Android offers better support for menus, while both platforms provide easy access to multithreading solutions. iOS leads the way in storage through the user of the core data frameworks, and again both platforms provide comparable solutions for web services and XML. In terms of location based services, Android allows the user more control over the source of the location information, while iOS leads the way with Google Maps integration. Finally, due to its strict controls, iOS tends to be more secure, however this is at the cost of exposing more functionality to third party developers.

Having solved any issues encountered with these through careful design and coding, the developer should be left with a fully functional application. Thoughts now turn to how best to distribute the application, and this is covered in the following chapter.

[2]Jailbreaking or rooting a device refers to gaining superuser or root access to the Linux operating system on the device in order to give the user or applications additional permissions. Jailbreaking is expressly forbidden by Apple, but is much more widely acceptable in the Android community with many different solutions available.

Table 5.1: Summary of Features

Menus	• XML or in code creation	• In code creation only
Multithreading	• 2 main options – AysncTask is recommended	• Many options – GCD is recommended
Storage	• SharedPreferences and PreferenceFragment so no custom code needed • SQLite and Cursors • Data sharing between applications • Shared filesystem access	• NSUserDefaults and custom view and controller needed • Core Data Framework – Graphical Editor and generated code – Built on top of SQLite • No exchange of data between applications • No shared filesystem access
Web Services & XML	• REST & JSON libraries included • 3rd party SOAP libraries • SAX and DOM parser support	• REST & JSON libraries included • 3rd party SOAP libraries • SAX and DOM parser support
Location Based Services	• Multiple providers • Problems with multiple MapActivities in single application • No annotation views • Developer signs license with Google	• Uses whatever providers device has • Can have multiple MapViews in single application • Annotation views with images and callouts • Google Maps licensed by Apple, developer agrees to terms
Security	• Open source (mostly) • No application reviews for Android Market • User agrees to permissions on installation • Sandboxing with access to external storage • Application signed by developer	• Closed model • Every application reviewed on App Store • iOS only allows access to certain features • Sandboxing with access to application directories only • Application signed by Apple

Chapter 6: **Distribution**

Once the main functionality of the application is complete, the developer should consider distribution of the application. Both Android and iOS have very different processes and requirements for submission, and this is discussed in the following sections. There is much stricter criteria for submission to the App Store than to the Android Market so it is very important that the developer completes the steps carefully.

Section 6.1 details the steps that must be followed in order for the applications to be accepted onto the App Store and the Android Market. Section 6.2 looks at the costs involved for developers of each platform, and this is followed by section 6.3 which details how developers are paid for their applications. Section 6.4 provides a summary of the information discussed in this chapter.

6.1 Process

The only authorised way to publicly distribute iOS applications is via the Apple App Store, and only applications from registered Apple developers are accepted. An iTunes Connect account must be set up to manage the distribution and allows the developer to set up payment and taxation information, view sales reports, In-App purchasing, etc. Apart from the application binary, other information must be supplied such as the name, description, artwork (in a variety of resolutions), support information, ratings, keywords and more. The application binary can be uploaded directly from Xcode using the Application Loader. Once all the required data has been supplied, applications can be submitted for review. There is a long list of App Store Review Guidelines which an application must meet in order to be approved and these range from items such as the application behaving as described, not mentioning other mobile platforms, having suitable icons and having a good user interface. Applications must also conform to the iOS Human Interface Guidelines. This includes guidelines on user experience along with instructions on how each of the standard user interface elements should be used. Part of the review includes testing the application on devices so applications with bugs will be rejected. If an application is rejected, the developer will be notified in detail as to the reason why it has failed the review. The developer then has the opportunity to appeal this, or to make the necessary changes and resubmit the application. Once the application has passed review, it is released automatically, based on the availability date. Alternatively the developer can use iTunes Connect to release the application to the App Store where it should appear within 24 hours.

There are a number of options for distribution of Android applications, the main one being the Android Market. This is a much less restrictive process than the App Store as no approval is needed before an application is made available for distribution. A number of configuration tasks are required, however, before the application can be submitted. The application code must be cleaned up to remove any debugging code, to make sure the correct packages are used, and to tidy up any resources that are to be included. The manifest should be reviewed to ensure it contains the correct permissions and the correct label, version code, version name, icon and SDK versions are specified. Once this has been done, the application should be compiled in release mode. An export wizard is included in Eclipse for this, or command line tools may be used. Part of this process is to sign and optimise the .apk file. The developer must obtain a cryptographic key and use this key to sign the .apk file. This may be a self-signed key created with the keytool utility included in the SDK, if the developer does not have a certificate issued by certificate authority. The zipalign tool is then used to optimise the application. Once this is completed the .apk file can be uploaded to the Android Market Developer Console. A number of screenshots and icons are now required, along with the application language, title and description. Additional data such as promotional graphics and videos, and publishing options such as rating, pricing and countries in which the application can be sold may be provided. Once the required information has been uploaded, the developer can publish their application and it will appear on the Market almost immediately.

The process for distributing an Android application requires much less work by the developer than the corresponding iOS process. Apple requires much more legal information and reviews every application submitted, while Android merely requires the developer to follow a few steps to optimise the application and provide some promotional information.

6.2 Cost

The cost of mobile application development varies widely depending on where the application is to be distributed. For iOS development, there is no charge if the application is never run on a device and only in the Simulator, but there are 3 main options if the application is to be distributed. Apple requires developers to enrol in a developer program and the basic one, the iOS Developer Program, is $99 a year. This allows the developer, or a team of developers in a single organisation, the ability to run the application on provisioned devices. It also allows access to all iOS Developer resources, free technical support requests and access to the App Store. For in-house applications, which are not distributed publicly, there is the iOS Enterprise Developer Program which costs $299 a year. This again allows access to resources and technical support from Apple engineers but is restricted to internal distribution only. An iOS University Developer Program also exists which is free to approved universities to introduce iOS development to the curriculum. This allows up to 200 students access to resources and internal distribution and testing on simulators and devices.

As well as the yearly fee for the developer program Apple also charges a 30% transaction fee on the price of every applications sold. This is not applicable to free applications.

In contrast to Apple, Android charges a one time $25 registration fee to join the Android Developer program. They also charge a 30% transaction fee on the application selling price, but again this does not apply to free applications.

Another factor to bear in mind when looking into the cost of development of mobile application is the cost of the hardware that is required. iOS development can only be done on an Apple Mac, and the cost of these is generally much higher than the cost of a personal computer. Android development can be done on a machine running Windows, Mac OS X or Linux so covers a much wider range of devices. Also because of the huge selection of different Android devices on the market, getting access to devices on which to test applications is much cheaper. For example the new Amazon Kindle Fire tablet device retails at just $199 in comparison to the Apple iPad 2 which starts at $499.

6.3 Payment

In order to distribute an application in the App Store, the developer must request a contract from Apple in iTunes Connect. There are contracts available for free applications, paid applications and the iAd Network. Legal entity, banking and tax information must be provided before the contract comes into effect allowing applications to be sold. It is the responsibility of the developer to ensure that the correct tax is paid on application sales. Once all documentation is complete Apple pays developers on a monthly basis, provided that the payment threshold of $150 dollars has been met. Payments are made to the bank account given in iTunes Connect and financial reports are available on a monthly basis. Revenue from the iAd Network and In-App purchases is also an option for the developer. An iAd Network account can be set up and the developer gets revenue from ads displayed in the application. In-App purchasing allows the developer to sell to the user from within the application, e.g. new levels in a game, or subscriptions to media content.

In order for an Android developer to sell and application on the Market, they must register for a Google Checkout Merchant account. The developer registers their bank account details and payments are made on a monthly basis with no minimum payment threshold. It is up to the developer to ensure that the correct tax is paid on application sales. Google may hold back portion of application income in reserve if there are excessive unauthorised credit card charge-backs, refunds, disputes, etc. Android developers can also gain revenue from embedding advertising in their applications, and this requires an AdSense account. In-App billing also exists as an option for Android developers, although there is currently no subscription mechanism available.

6.4 Summary

This chapter has examined distribution of mobile applications on both Android and iOS. There are a range of differences between both platforms, across process, cost and payment. The control Apple exercises over the App Store may deter developers, but reassures device owners who feel secure knowing that each application has been reviewed. The Android Market is easier to use for developers as there is a simple submission process and applications are not reviewed, but users may be more wary of downloading applications. Another downside for developers is the yearly fee charged by Apple for distribution via the App Store. This is in stark contrast to the one time administration fee charged by Google for distribution via the Android Market. Differences even exist when it comes to payment as Apple imposes a minimum sales threshold so developers may face months where no income is received. Both platforms provide the ability to earn income from embedded advertising and in-app purchases, however, iOS is the only one which offers a subscription mechanism.

Chapter 7: **Conclusion**

There are a huge number of factors that a developer needs to be aware of when developing a mobile application for either iOS or Android, and many have been discussed in the previous chapters covering the development of the LiveBus applications. The remainder of this chapter attempts to summarise the overall experience of the developer working with each, and to draw some conclusions.

Section 7.1 discusses the flexibilities and restrictions encountered when working with the Android and iOS platforms. This is followed by section 7.2 which looks at reusabilty. Section 7.3 looks at the popularity of the two platforms and the effect this has on developers. Section 7.4 looks at some alternatives to Android and iOS application development. Section 7.5 provides a summary of the information discussed in this report, and is followed by section 7.6 which discusses the personal views of the author.

7.1 Flexibility and Restrictions

In terms of flexibility of process, Android wins hands down, as there are very few steps that must be followed in order to get an application onto the Android Market. It generally takes about a day to get a completed Android application, along with all its marketing information onto the Android Market. Apple on the other hand has a rigorous inspection process, and developers must also supply the relevant legal information before they can sell applications. It could take a couple of weeks to get an iOS application onto the App Store depending on how long the review takes, and whether the developer has submitted all necessary legal documentation.

Android expose much of the internals of their code so developers can get a good understanding of how it works, and can even contribute to it through the Android Open Source project. Apple code remains closed to developers, and applications that make use of undocumented APIs will be rejected. Developers are also restricted for writing applications that perform the same function as an existing Apple application.

From a user's point of view Apple are much more strict with what you can do with your device. As mentioned previously, Apple iPhones and iPads can only install software from the App Store, and jailbreaking a device voids the warranty. Even when it comes to simple issues like sharing photographs, Apple enforces restrictions. For example, even though the iPad has bluetooth it will only connect to other Apple devices or accessories, so there is no way to share photographs with another phone or another computer using bluetooth. Android phones can connect to any other device that implements the required Bluetooth protocols, and can easily share any kind of file. Alternatively, any application that declares the relevant intent for sharing photographs can be used to share them in many different ways. iPhone and iPad users have to sync photographs via iTunes, iCloud, or another application like Dropbox, or can simply email the photographs to themselves.

7.2 Reusability

As mentioned in the previous section, with regards to sharing photographs, the use of Intents in Android allows applications to work together. This is an extremely useful mechanism as it allows applications to take advantage of the strengths of other native or third party applications. For example, when the user chooses to share a photograph, they are given a number of applications to use such as Dropbox, Facebook, Flickr, Gmail, etc. Apple does not allow this kind of interaction, except through some approved custom URL schemes as mentioned in section 5.3. Even with this, data cannot be passed back and forth between applications as it can with Android. With the introduction of iCloud in iOS 5, Apple now allows sharing of data between instances of the same application on the user's different devices.

In terms of code reuse, both Android and iOS support the use of external libraries or Frameworks, such as Google Maps integration. One interesting point here was how much better the Google Maps APIs (MapKit) was on iOS than the corresponding APIs were on Android. MapKit was much more stable and offered much more in terms of features.

7.3 Popularity

When the Apple iPhone first launched it quickly gained a cult following and soon became the smartphone to beat. The first Android device launched a year later, but a rapid cycle of software releases and new device launches followed meant that Android was soon a serious competitor for Apple's iOS. As discussed in section 1.1 iOS and Android top the smartphone market and are quite a distance ahead of the competition. Due to the larger number of Android manufacturers and their more competitive pricing, there are currently more Android smartphones in circulation than there are Apple. However, Apple remains the number one manufacturer beating off competition from Samsung [17], the leading Android manufacturer.

According to Apple's own website, there are currently over 500,000 applications available for download on the App Store. Apple reported over 15 billion application downloads in July 2011 [13], but Android is quickly gaining ground having jumped from over 6 billion application downloads in July to over 10 billion in December 2011 [25]. There are no official figures on the number of applications on the Android Market, however source such as AppBrain [5] estimate it at about 350,000.

7.4 Alternatives

Other smartphone operating systems also exist, but none come close to Android or iOS in terms of market penetration. Symbian was once the market leader but has since been eclipsed by Android and iOS, and Windows Phone 7 and Blackberry OS are also lagging behind.

Many commercial organisations are looking for a solution where they can write an application once and deploy it across all of these platforms. This can be achieved with varying degrees of success, depending on the functionality required and the target platforms. A number of free tools exist where developers can write their application using HTML and JavaScript and this is generated into native applications for the required operating systems. PhoneGap [42] is one example of this and supports applications for 7 platforms including Android, iOS, Symbian, Web OS, and Windows Phone. It is an open source framework which provides APIs for access to native features. Another such tool is MoSync [37], however, this is licensed under GPL2 so any code written using it must be made publicly available or a commercial subscription must be paid. Commercial tools also exist, such as Adobe AIR [1], Corona [4] and Monkey [36]. Applications on Adobe AIR can be written in HTML and JavaScript, or Flex and ActionScript. Not all native functionality is supported, however, so the developer will need to write native extensions if this is required. For development with Corona or Monkey, the developer must learn a new language. Lua is used for development with Corona, and Monkey code is used for development with Monkey. Corona can only be used to develop Android and iOS applications and not all features are supported across both platforms. Monkey can be used to develop applications for Android, iOS, HTML5, along with Windows and Mac. Such tools sound great, but may not offer all the functionality that comes with writing a native application. In addition to this, the developer will still need to have an understanding of the underlying platform, as the generated code may need to be tweaked; for example it may not be efficient, it may not work on every device, or it may fail the iOS review process.

Another option for the developer is to write web applications for each platform. Here the main functionality is written in HTML5 and JavaScript, and minimal native code is required to launch the application, or it can be run exclusively in a web browser. HTML5 web applications can be written to function offline and can also make use of databases on the device so network functionality is not required in order to run them. The main advantage of writing web applications, is that the same HTML5 code can be reused across the platforms. One drawback of HTML5 is that it is still quite a young technology and may not provide all the functionality the developer requires.

7.5 Summary

Both platforms may seem daunting on first look, but developers soon find there is a multitude of resources available. Both allow developers with a good knowledge of object oriented programming to develop relatively complex applications, in a relatively short time frame. Both platforms offer developers the opportunity to sell their work to a large user base. Both platforms have their weak and strong points, with neither one a clear winner or loser. Neither platform faces any real competition from the other alternatives at this stage.

Others such as Goadrich [24] and Grundström [26] have made similar comparisons between the two platforms. Goadrich looks at both from the perspective of a college lecturer trying to decide which one to teach, and reaches the conclusion that a course in either iOS or Android would be both interesting and valuable to the student. Grundström comes out in favour of Android because of its technological focus and the restrictions of Apple. However, Wong's article in Electronic Design takes another approach, that there is no clear winner between Android and iOS, so essentially the only winner is the end user [53].

From a financial point of view Android makes more sense for the hobbyist, or independent developer. There are fewer applications on the Android Market, therefore greater opportunity to get your application sold. The costs are less with no yearly fee. Also Android Market downloads have more than doubled in the past year alone so this is a rapidly growing market. For developers looking to make a living out of developing applications, iOS currently seems to generate more income than Android. Recent reports from analytics company Flurry point to a much higher percentage of new iOS versus Android applications being developed, with 73% of new projects using iOS and just 27% choosing Android [21]. The same report also points to a much higher return from paid iOS applications than from paid Android applications; estimating that for every $1 earned for an iOS application, the corresponding Android application returns just $0.24. It is unclear if this trend will continue, due to the massive growth in both sales of Android smartphones and the number of application downloads.

One major difference between the two platforms is that Android is mostly open source, while iOS uses a closed model, both in terms of software and device manufacture. The open source nature of Android means that code is extensively reviewed by the open source community, who may make their own contributions. The variety of manufacturers means that users have a wider choice of devices and pricing is more competitive. However, this leads to more work for the developer to ensure their application runs on as many devices as possible. The relaxed control of the Android Market is good as it allows developers to get their applications on sale in a short time frame, however, it leaves end users vulnerable as they have no guarantees that applications are genuine and will not cause malicious damage. iOS code is strictly controlled by Apple, and developers only have access to approved APIs which limits what an application can do. As Apple is the only manufacturer of iOS devices, application development is much easier due to the homogeneity of target devices. Developers know how their software will perform on any Apple device and how their user interfaces will look. Apple imposes a much stricter process in order to make an application available for distribution, but end users like this as they are confident that any application downloaded has been reviewed and will not do any harm. Android promotes interoperability between devices through open standards, whereas Apple goes against this and tries to lock the user in to the Apple ecosystem. The Apple business model is a successful one, with developers making more money from iOS applications than Android, however the recent growth in numbers of Android devices and downloads may change this.

7.6 Personal Reflections

From a developer's point of view, I cannot say definitively that Android is better than iOS or vice versa. With one relatively small application complete, there is still much to learn about both platforms. A developer will no doubt be influenced in this by a number of factors, one of which is their own personal experience, both as a programmer and as a smartphone owner. If a developer is more familiar with Java than Objective-C then they are likely to prefer programming in the language they know best. If the developer owns and likes a smartphone with one particular mobile operating system, they are more likely to try developing applications for that platform before another. I enjoyed developing both applications, and learnt a lot throughout the whole process. I am predominantly a java developer and own an Android smartphone, so found this application much easier to develop. Part of this was due to the steeper learning curve required to develop the iOS application, and part due to the flexibility and lack of restrictions on the Android platform. However, I successfully developed an application with the same basic functionality for both platforms in a comparable amount of time. I prefer the open model used by Android, but in reality there is not a huge difference between both platforms. I feel that knowledge of both Android and iOS will ultimately prove useful to me. With the recent release of iOS 5 and Android 4, even more functionality is available to the application developer, and it will be interesting to see where the future of mobile application development leads.

Bibliography

[1] Adobe. Adobe AIR. http://www.adobe.com/products/air.html, December 2011.

[2] Android Developers. http://developer.android.com/index.html, November 2011.

[3] Android Licenses. http://source.android.com/source/licenses.html, November 2011.

[4] Ansca. Corona SDK. http://www.anscamobile.com/corona/, December 2011.

[5] AppBrain. Number of available Android applications. http://www.appbrain.com/stats/number-of-android-apps, December 2011.

[6] JetBrains AppCode Objective-C IDE. http://www.jetbrains.com/objc/, November 2011.

[7] Apple Announces iPhone 2.0 Software Beta. http://www.apple.com/pr/library/2008/03/06Apple-Announces-iPhone-2-0-Software-Beta.html, November 2011.

[8] Apple Presents iPod. http://www.apple.com/pr/library/2001/10/23Apple-Presents-iPod.html, November 2011.

[9] Apple Press Info: iPod + iTunes Timeline. http://www.apple.com/pr/products/ipodhistory/, November 2011.

[10] Apple Sells One Million iPhone 3Gs in First Weekend. http://www.apple.com/pr/library/2008/07/14Apple-Sells-One-Million-iPhone-3Gs-in-First-Weekend.html, November 2011.

[11] Apple Sells One Millionth iPhone. http://www.apple.com/pr/library/2007/09/10Apple-Sells-One-Millionth-iPhone.html, November 2011.

[12] Apple's App Store Downloads Top 15 Billion. http://www.apple.com/pr/library/2011/07/07Apples-App-Store-Downloads-Top-15-Billion.html, November 2011.

[13] Apple. Apples App Store downloads top 15 billion. http://www.apple.com/pr/library/2011/07/07Apples-App-Store-Downloads-Top-15-Billion.html, July 2011.

[14] iOS Developer Program. http://developer.apple.com/programs/ios/, November 2011.

[15] iPhone 4S First Weekend Sales Top Four Million. http://www.apple.com/pr/library/2011/10/17iPhone-4S-First-Weekend-Sales-Top-Four-Million.html, November 2011.

[16] Apple Q&A on Location Data. http://www.apple.com/pr/library/2011/04/27Apple-Q-A-on-Location-Data.html, April 2011.

[17] International Data Corporation. Apple Rises to the Top as Worldwide Smartphone Market Grows 65.4% in the Second Quarter of 2011. http://www.businesswire.com/news/home/20110804006519/en/Apple-Rises-Top-Worldwide-Smartphone-Market-Grows, August 2011.

[18] Document Object Model (DOM). `http://www.w3.org/DOM/`, December 2011.

[19] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[20] EGit Eclipse plugin. `http://eclipse.org/egit/`, November 2011.

[21] Flurry. App Developers Bet on iOS over Android this Holiday Season. `http://blog.flurry.com/bid/79061/App-Developers-Bet-on-iOS-over-Android-this-Holiday-Season`, December 2011.

[22] E. Gamma. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[23] Gartner. Market Share: Mobile Communication Devices by Region and Country, 2Q11. `http://www.gartner.com/resId=1764117`, August 2011.

[24] M.H. Goadrich and M.P. Rogers. Smart Smartphone Development: iOS versus Android. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 607–612. ACM, 2011.

[25] Google. 10 Billion Android Market downloads and counting. `http://googleblog.blogspot.com/2011/12/10-billion-android-market-downloads-and.html`, December 2011.

[26] P. Grundström. Mobile Development for iPhone and Android, 2010.

[27] Paul Hegarty. CS193P iPhone Application Development, Stanford University. `http://www.stanford.edu/class/cs193p/cgi-bin/drupal/`, November 2011.

[28] JetBrains IntelliJ IDEA. `http://www.jetbrains.com/idea/`, November 2011.

[29] Java SE at a Glance. `http://www.oracle.com/technetwork/java/javase/overview/index.html`, November 2011.

[30] Don Kellogg. In U.S. Market, New Smartphone Buyers Increasingly Embracing Android. `http://blog.nielsen.com/nielsenwire/online_mobile/in-u-s-market-new-smartphone-buyers-increasingly-embracing-android/`, September 2011.

[31] Khronos. Khronos OpenGL ES API Registry. `http://www.khronos.org/registry/gles/`, November 2011.

[32] ksoap2-android. `http://code.google.com/p/ksoap2-android//`, November 2011.

[33] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *ACM Sigplan Notices*, volume 21, pages 214–223. ACM, 1986.

[34] R. Meier. *Professional Android 2 Application Development.* John Wiley & Sons, 2010.

[35] C. Miller. Mobile attacks and defense. *Security & Privacy, IEEE*, 9(4):68–70, 2011.

[36] Monkey. Corona SDK. `http://www.monkeycoder.co.nz/`, December 2011.

[37] MoSync. MoSync. `http://www.mosync.com/`, December 2011.

[38] M. Murphy. *Beginning Android 3.* Apress Series. Apress, 2011.

[39] V. Nahavandipoor. *IOS 4 Programming Cookbook: Solutions & Examples for IPhone, IPad, and IPod Touch Apps*. O'Reilly Media, 2011.

[40] The Objective-C Programming Language. `http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html`, November 2011.

[41] Open Handset Alliance. `http://www.openhandsetalliance.com/index.html`, November 2011.

[42] PhoneGap. PhoneGap. `http://phonegap.com/`, December 2011.

[43] D. Pilone and T. Pilone. *Head First IPhone and IPad Development: A Learner's Guide to Creating Objective-C Applications for the IPhone and IPad*. Head First Series. O'Reilly Media, 2011.

[44] B. Rao and L. Minakakis. Evolution of mobile location-based services. *Communications of the ACM*, 46(12):61–65, 2003.

[45] T. Reenskaug. Thing-model-view-editor-an example from a planningsystem. *Xerox PARC technical note*, 12, 1979.

[46] T.M.H. Reenskaug. Models-views-controllers. technical note, xerox parc, 1979.

[47] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google Android: A comprehensive security assessment. *Security & Privacy, IEEE*, 8(2):35–44, 2010.

[48] Simple API for XML (SAX). `http://www.saxproject.org`, December 2011.

[49] SQlite Database Engine. `http://www.sqlite.org/`, November 2011.

[50] Stack Overflow: Android tag. `http://stackoverflow.com/questions/tagged/android`, November 2011.

[51] Subversive Eclipse plugin. `http://www.eclipse.org/subversive/`, November 2011.

[52] Text Me My Bus by Wexford Bus. `http://www.textmemybus.com/`, November 2011.

[53] B. Wong. Google's Android vs. Apple's iOS and the winner is? *Electronic Design*, 58(15), 2010.

[54] wsdl2objc. `http://code.google.com/p/wsdl2objc/`, November 2011.