**Table of Contents**

[Introduction 3](#_Toc446416712)

[Project Planning and Organisation 4](#_Toc446416713)

[Original Project Plan 4](#_Toc446416714)

[Final Project Plan 4](#_Toc446416715)

[Reasons for Change 4](#_Toc446416716)

[Organisational Tools 4](#_Toc446416717)

[Research 5](#_Toc446416718)

[Application Type 5](#_Toc446416719)

[Web Application 5](#_Toc446416720)

[Native Application 5](#_Toc446416721)

[Comparison of Native and Web Applications 5](#_Toc446416722)

[Android 7](#_Toc446416723)

[iOS 7](#_Toc446416724)

[Comparison of Android and iOS Applications 7](#_Toc446416725)

[Possible Competition 8](#_Toc446416726)

[Standard Routing 8](#_Toc446416727)

[Disability Improvements 8](#_Toc446416728)

[Institutional Routing 8](#_Toc446416729)

[Static or Dynamic Routing 8](#_Toc446416730)

[Static Routing 8](#_Toc446416731)

[Dynamic Routing 8](#_Toc446416732)

[Location Services 10](#_Toc446416733)

[Global Navigation Satellite System (GNSS) 10](#_Toc446416734)

[Cell Tower Based Location 11](#_Toc446416735)

[Wi-Fi Based Location 11](#_Toc446416736)

[Current Android Solution 12](#_Toc446416737)

[Improved Suggestion 13](#_Toc446416738)

[Routing 13](#_Toc446416739)

[Representing a Graph 13](#_Toc446416740)

[Edge List 13](#_Toc446416741)

[Adjacency Matrix 13](#_Toc446416742)

[Adjacency List 14](#_Toc446416743)

[Path Finding 14](#_Toc446416744)

[Breadth-First Search 14](#_Toc446416745)

[Dijkstra’s Algorithm 15](#_Toc446416746)

[A\* Algorithm 15](#_Toc446416747)

[Software Engineering 16](#_Toc446416748)

[Architectural Style 16](#_Toc446416749)

[Object-Oriented 16](#_Toc446416750)

[Event-Based Implicit Invocation 16](#_Toc446416751)

[Programming Style 16](#_Toc446416752)

[Design Patterns 17](#_Toc446416753)

[Observer Pattern 17](#_Toc446416754)

[Adapter Pattern 18](#_Toc446416755)

[Template Pattern 18](#_Toc446416756)

[Strategy Pattern 19](#_Toc446416757)

[Composite Pattern 20](#_Toc446416758)

[Design 21](#_Toc446416759)

[Initial High-Level Design 21](#_Toc446416760)

[Responsibilities 21](#_Toc446416761)

[Communications 21](#_Toc446416762)

[Initial View Design 22](#_Toc446416763)

[Initial Model Design 24](#_Toc446416764)

[Initial Controller Design 24](#_Toc446416765)

[Start-Up 25](#_Toc446416766)

[Data Requirements 25](#_Toc446416767)

[Testing Strategies 26](#_Toc446416768)

[Performance Analysis & Future Proofing 26](#_Toc446416769)

[Routing Performance 26](#_Toc446416770)

[Drawing Performance 27](#_Toc446416771)

[Future Work 27](#_Toc446416772)

[Improving the Route Modifiers Implementation 27](#_Toc446416773)

[Adding Location Services 28](#_Toc446416774)

[Disjoint Routes 28](#_Toc446416775)

[Reaching More Platforms 28](#_Toc446416776)

[Release and Updates 28](#_Toc446416777)

[Future Applications 28](#_Toc446416778)

[Without Extensions 28](#_Toc446416779)

[Recruitment 28](#_Toc446416780)

[Conferences 28](#_Toc446416781)

[Deliveries 28](#_Toc446416782)

[With Extensions 29](#_Toc446416783)

[Disability Services 29](#_Toc446416784)

[Fire Safety 29](#_Toc446416785)

[Appendices 30](#_Toc446416786)

[Appendix 1 Strathclyde Disability Adviser Interview 30](#_Toc446416787)

[Appendix 2 Initial High Level UML Diagram 31](#_Toc446416788)

[Appendix 3 View UML Diagram 32](#_Toc446416789)

[Appendix 4 Initial Model UML Diagram 33](#_Toc446416790)

[Appendix 5 Initial Controller UML Diagram 34](#_Toc446416791)

[Appendix 6 Final UML Diagram 35](#_Toc446416792)

[Appendix 7 User Guide 35](#_Toc446416793)

[Appendix 8 Installation Guide 35](#_Toc446416794)

[References 36](#_Toc446416795)

# Introduction

The main aim of this project is to create an application that can:

* Implement a routing algorithm capable of dealing with an institute the size of the John Anderson campus.
* Visualise this route in a way that a user can easily understand.

In addition to this over the course of the project an investigation should be carried out into the feasibility of including route modifiers and location services into this application. For the purposes of this report Route Modifiers are defined as settings which depending on their values will give different routes for different users. An example of this would be to avoid stairs. Location services would be some way of automatically determining where the user is within the mapped section. This could then be used to either provide Dynamic Routing as they follow the path or to simply allow them to input their start location more easily.

# Project Planning and Organisation

This project took place over several months which meant that if time was not managed carefully important tasks may have been left incomplete. From this need a project plan was developed.

## Original Project Plan

## Final Project Plan

## Reasons for Change

## Organisational Tools

# Research

Before any development work could begin several key questions had to be answered about how the project should proceed: Would the application provide static or dynamic routing to the user? How would the Application be developed? How would the shortest path be found?

## Application Type

It was decided that the application should be able to be used on mobile devices as this offers the greatest levels of flexibility to the user. This means the user could use it throughout the day as they traverse the mapped area as opposed to having to plan their routes for the day in advance. With this requirement in mind there are two main types of application that would be possible a Web Application or a Native Application [1].

### Web Application

These applications are developed to be ran within a web browser. This means that they can be used across virtually any device including both mobile devices and more traditional computing platforms. They are designed and built as a website with special care taken to appear correctly on mobile devices. This means they can be accessed by navigating to a particular webpage on the phone.

### Native Application

Native applications are built specifically for the mobile operating system being targeted. This means if it is built for Android it cannot be used on iOS or vice versa. To use a Native application, they are first installed from an OS specific store (either the App Store for iOS or the Google Play Store for Android). They can then be launched from a home menu.

### Comparison of Native and Web Applications

To decide between Native and Web Applications they must be compared on a number of key features to the project. This reasoning can be seen in Table 1.

Table Comparison of Native and Web Applications on Key Issues

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Issue to compare | Native Application | Web Application | Which is better for this project | Reason |
| Portability | Can only be used on the OS it was designed for. | Can be used on virtually any Operating System that includes support for a web browser. | Web | If this was to be released it would be important for as many students to have access to it as possible. Although multiple Native Applications could be created it would be far easier to have one Web Application. |
| Performance | Developed specifically for each platform which means greater performance. | Lower performance. | Native | Performance may be an issue as the size of the map increases. Although this is also an issue for the choice of routing algorithm it makes sense to make the later choice easier. |
| Access to phone sensors | Virtually full access to all of the phone sensors. | More restricted access to phone sensors. | Native | This could be important for location services. Although Web applications have access to geolocation information [2] they do not appear to have access to information on Wi-Fi access points they are not currently connected to [3]. Native applications by comparison have access to this information [4]. |
| Programming Language | Uses Object Oriented languages (Java or Objective C) which makes Encapsulation, Abstraction, etc far easier than they would have been otherwise. | Uses JavaScript for the main bulk of work. This can be programmed in an object oriented fashion but it is much more difficult to do so. Does not enforce strict type-checking meaning errors can be made far more easily during development. | Native | It provides a better framework for producing an error free application more quickly. |

As can be seen from Table 1 Native applications have a number of advantages that are important to the project particularly for the study into Location Services feasibility. There are two major choices to be considered when developing a Native Application: Android and iOS.

### Android

These applications are developed specifically for the Android OS and are widely available for download in the Google Play Store. Android phones are produced by a range of manufacturers including Sony, Samsung and HTC.

### iOS

iOS applications are developed specifically for Apple’s mobile devices i.e. the iPhone and iPad. They are available for download on these devices through the App Store.

### Comparison of Android and iOS Applications

A number of issues should be considered before deciding on a platform to build the application for. Several of these are examined in

Table Comparison of Android and iOS Applications on Key Issues

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Issue | iOS | Android | Better for this project | Reason |
| Percentage of users worldwide | 13.9% [5] | 82.8% [5] | Android | Users are far more likely to own an Android phone and therefore be able to use the application. |
| Integration with mapping services | Wide support for many major mapping applications including Google [6] and Bing [7]. | Similar levels of support from the major mapping applications [8] [9]. | Tie | Both options offer similar levels of integration with popular mapping applications. |
| Availability | None. | Several devices of different sizes and features. | Android | Circumstances surrounding the project have dictated these are more easily available. |

For the reasons detailed above the application will be developed as an Android application.

## Possible Competition

### Standard Routing

### Disability Improvements

### Institutional Routing

## Static or Dynamic Routing

To determine whether the application should provide static or dynamic routing it is first important to understand what both involve.

### Static Routing

For the purposes of this report Static Routing will be defined as routing between two fixed points with the whole route available for the user to view. A popular example of this is google maps which can be seen in Figure 1.

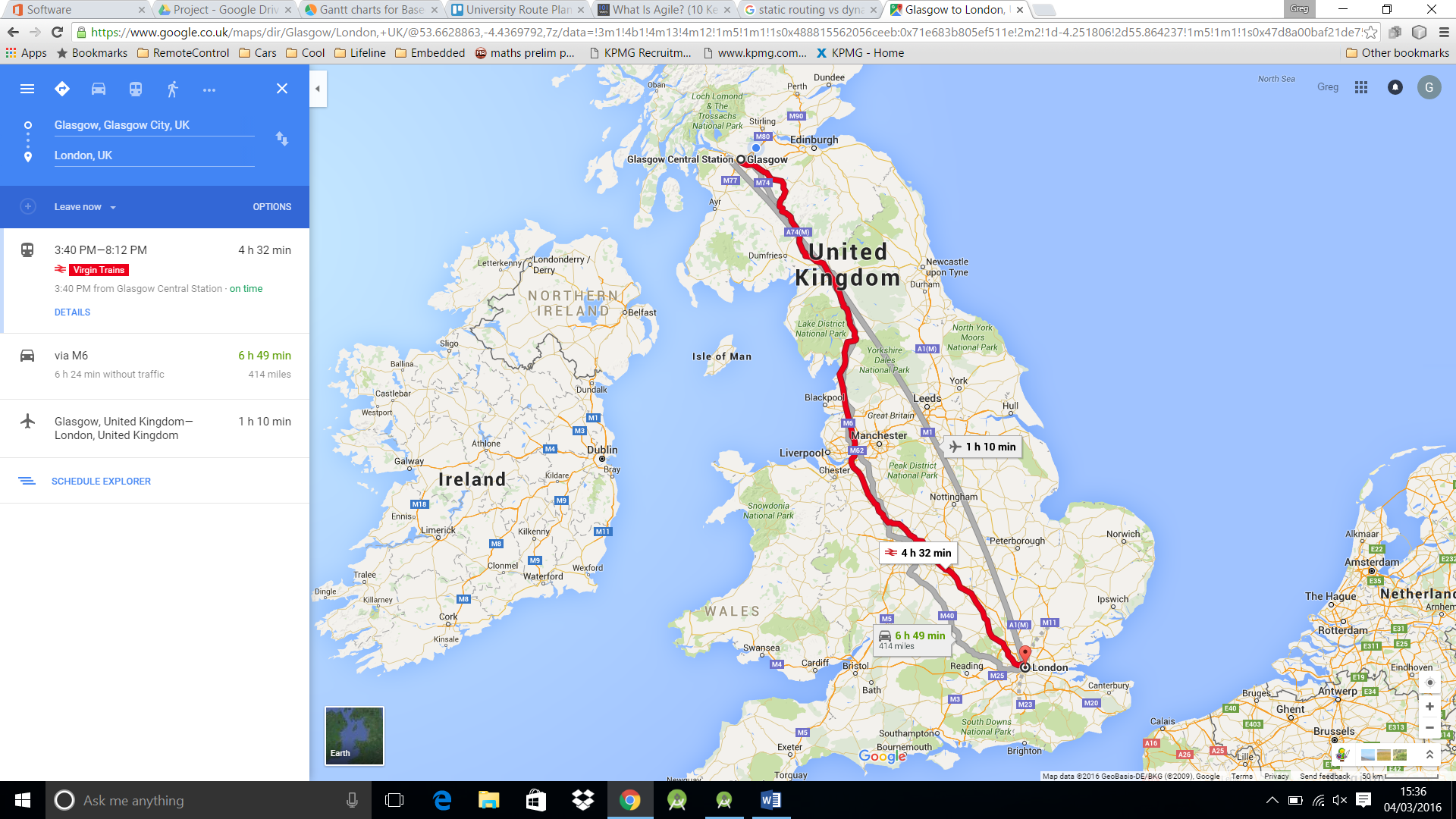


Figure Static Routing Example [10]

### Dynamic Routing

By comparison Dynamic Routing will be defined as routing from the user’s changing location to a fixed endpoint. This means as the user moves the route will automatically adjust to their new location. This is the type of routing used in Satellite Navigation systems or Google Navigation which can be viewed below in Figure 2.

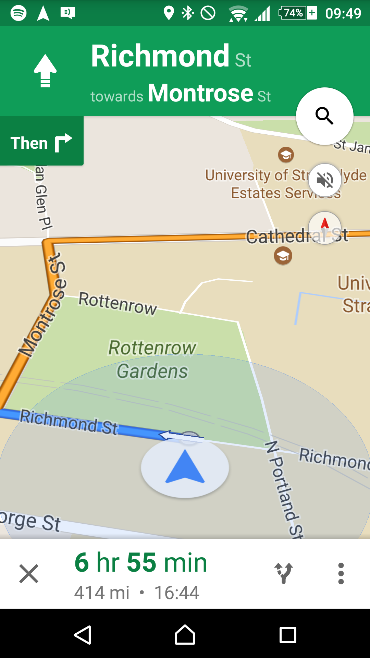


Figure Dynamic Routing Example

It is also common for the display to be more restricted showing only a short section of the route at a time i.e. where the user is to go next.

## Location Services

The main difference between the two types of routing is whether the user’s location is utilised to improve the user interface or not. In order to decide between them then it is important to determine whether this is possible.

There are three main technologies which could be used to determine the user location: Global Navigation Satellite System, Cellular Positioning and Wi-Fi.

### Global Navigation Satellite System (GNSS)

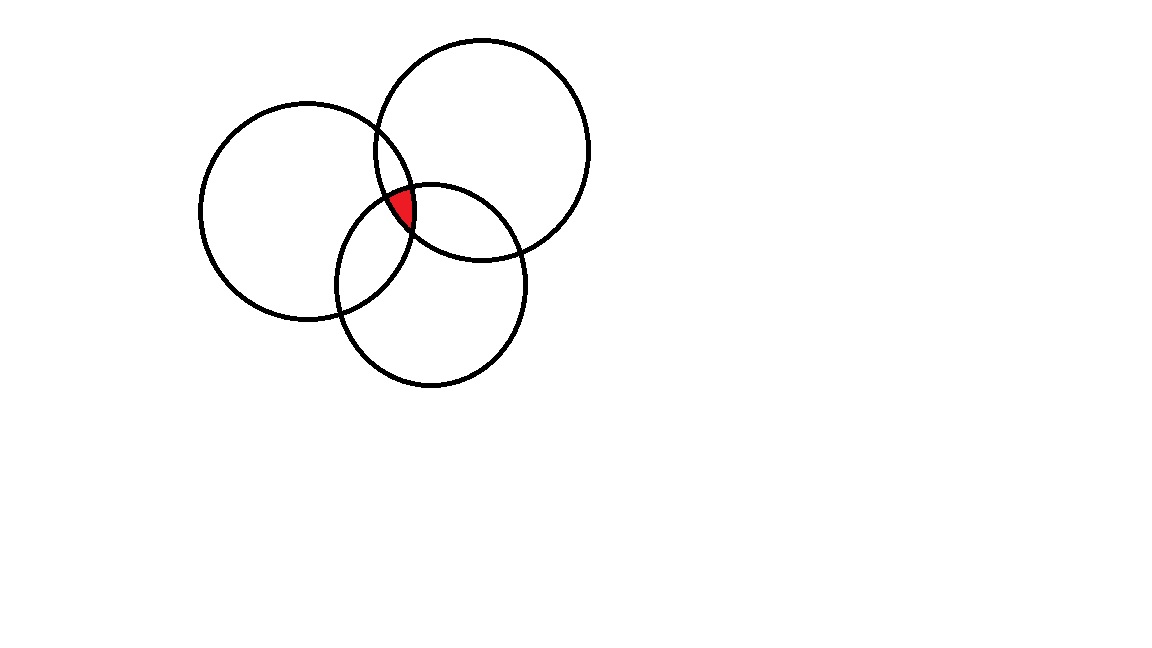
There are several of these systems currently in operation. The most famous of these being the U.S. funded Global Positioning Service (GPS) however there are others such as the Russian Federation’s GLONASS and soon the European Galileo [11]. These were devised as systems to enable users worldwide to calculate their current location. The GPS system for example comprises of twenty-four satellites continuously producing signals that tell a user where a satellite is and highly accurate timing information about when this signal was produced. All twenty-four satellites communicate on the same two frequencies using Code Division Multiple Access in order to avoid collisions. By listening to the signals generated by at least four of the satellites to discover their locations and calculating the time it has taken to reach the user and hence how far away they are from each the users own location can be calculated. [12]

The accuracy of GPS is roughly 3.5m on the horizontal plane without using additional information. There are other systems which can be used in conjunction with GPS to provide up to millimetre accuracy however 3.5m should be accurate enough for this project [13]. The accuracy of Galileo is expected to improve to the centimetre range [14] once it is completed in 2020 [15].

There are two main limiting factors however: the accuracy of the GPS receiver in the user’s phone and that much of the time the user will be indoors. The accuracy of the receiver in a typical mobile phone is roughly 2-3m under good multipath conditions or up to 10m under bad multipath conditions [16]. Multipath is when the signal may reach the receiver via many paths for example direct from the satellite compared to being reflected by a building in between the satellite and receiver. This means that the signal will degrade when near buildings due to the weak multipath suppression available on mobile phone antennae. The app will always be expected to be running in an urban landscape therefore meaning the GNSS location may have quite a low accuracy. Receiving GNSS signals indoors can become a challenge as the signal when received outdoors is already very weak when compared with the background noise and the signal can be 10-100000 times weaker when indoors [17]. This can make it difficult to receive the necessary signals from four satellites at once making it near impossible to reliably and quickly determine location indoors via this method.

### Cell Tower Based Location

Location can be calculated using cell towers in a similar way to that of GNSS. Essentially the phone sends a ping to the nearby cell towers and times how long it takes to receive a response. If the locations of the cell towers are known then an estimate of the phone’s location can be calculated using the respective distances from each cell tower [18]. An example of how this works can be seen below in Figure 3 Cell Tower Triangulation.



1

3

2

Figure Cell Tower Triangulation

In the figure above the arrows represent the calculated distance from the cell tower. With only one cell tower response this means the user could be anywhere in the surrounding circle. This is as the furthest the user could possibly be from the tower would be is a straight line from it (the radius of the circle) however there is no guarantee that is the path being taken. Instead it could be reflecting off of buildings or other materials meaning the user could be anywhere within the surrounding circle. With responses from more cell towers the location becomes more accurate. As can be seen in Figure 3 Cell Tower Triangulation with these responses the user could only be in the overlapping area from each circle.

Cell Tower Triangulation could potentially be a more reliable method of geolocation as it even tends to work indoors, presuming there is a mobile phone signal, and even works better in more urban areas where this application would be expected to be used, due to the higher number of cell towers [18]. It does however have some drawbacks in that it requires an accurate knowledge of cell tower locations which are not typically available publically making this more difficult [19]. On average for example in the United States of America (USA) the average accuracy of a three tower triangulation is 0.75 square miles [20]. This is far too inaccurate for the purposes of this application.

### Wi-Fi Based Location

In much the same way cell towers can be used to calculate the user’s location so can Wi-Fi base stations. By using easily available data such as signal strength a rough estimate of distance from the base station can be calculated. If several base stations can be identified, then the user’s location can be calculated using triangulation.

### Current Android Solution

Android as an OS has several built in features for detecting the user’s current position and provides app developers with the opportunity to use this data. It currently uses a combination of all 3 of the location services previously mentioned [4]. The accuracy while outside is particularly high as can be seen in Figure 4.

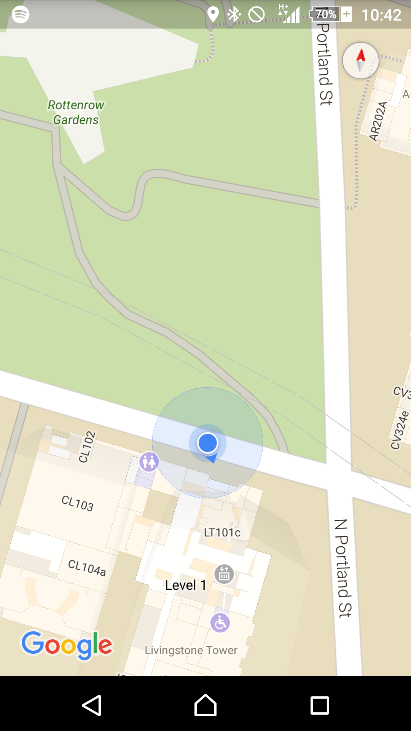


Figure Outdoor Location Example

This is likely due to the availability of a strong GPS signal in combination with cell tower and Wi-Fi location services. When inside however the provided location services are far weaker as can be seen in Figure 5.

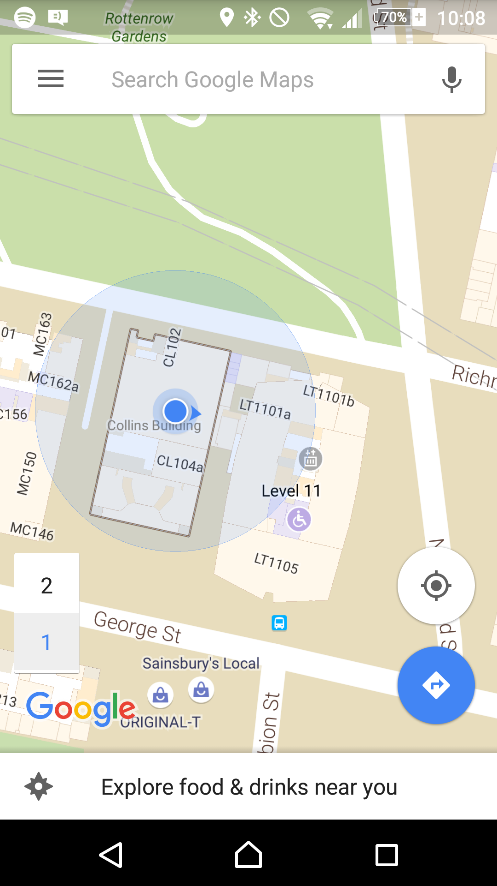


Figure Indoor Location Example

In Figure 5 again the reported location is shown with a small blue circle this can be compared to the actual location represented by the red circle to show that when indoors there can be a large margin of error. The larger blue circle represents the area in which google admits the location may fall within. Although technically the actual location does fall within this area the data is still almost unusable from the perspective of this app. The area of uncertainty shown above covers an entire building as well as significant amounts of two neighbouring buildings meaning this is unlikely to be useful when trying to navigate within one building. This large degree of uncertainty and inaccuracy is most likely due to a lack of a reliable GPS signal combined with a low knowledge of the exact Wi-Fi and Cell Tower locations. This should be improving however as google is trying to improve its knowledge of Wi-Fi locations using their massive user base. This is being done by taking the times when a user has a good knowledge of their own location using GPS to send google information on the surrounding Wi-Fi signals and strengths [21]. This could eventually theoretically provide strong indoor locations worldwide however is currently falling short of this. Even if it does improve using GPS as the starting point means height will always likely be a problem meaning differentiating between floors of a building could be difficult.

### Improved Suggestion

As previously stated it is believed that the main problem currently facing indoor location services is a knowledge of the locations of different Wi-Fi sources. Although steps are being made to try to improve this globally within a smaller area that is to be mapped such as a University campus this problem becomes far easier to deal with. Within this area the locations could be found manually preferably by using documentation written during the installation of a Wi-Fi network however if not a Wi-Fi map could potentially be built up when the app is being built.

## Routing

In order to build an application that will provide a route for the user between their chosen start and destination a graph representing the area to be mapped must be built. Once the graph has been built a method for finding the shortest path between two nodes in that graph must be found to indicate the route that the user should follow.

### Representing a Graph

There are three main ways of representing a graph within a computer system Edge List, Adjacency Matrix and Adjacency List [22].

#### Edge List

Consists of a list of all of the edges in the system. This is a fairly easy to understand representation of a graph however can be difficult to traverse during path finding algorithms. By indexing the information by edges rather than nodes it can make moving from one node to another more difficult. Each time it is needed to check what edges are connected to the current node it is necessary to move through all of the edges checking whether or not that node is attached to that particular edge.

#### Adjacency Matrix

Consists of a Matrix with the nodes along both axes and the weight of the edge connecting them at the point where they meet. An example of this can be seen in Figure 6 and Figure 7.

Figure Example Graph

4

C

6

5

D

1

A

B

2

2

E

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | Null | 2 | 4 | Null | 1 |
| B | 2 | Null | Null | Null | 2 |
| C | 4 | Null | Null | 6 | Null |
| D | Null | Null | 6 | Null | 5 |
| E | 1 | 2 | Null | 5 | Null |

Figure Example Adjacency Matrix

Adjacency matrices are typically used in very densely populated graphs i.e. those with a lot of edges per nodes. This is as the number of edges does not affect the storage space required for this implementation however if the graph is sparser with each node having far fewer edges than there are other nodes this method becomes increasingly inefficient with almost the entire matrix being filled with null values to represent no edge.

#### Adjacency List

This is a method in which each node has a list of the nodes it is connected to. By storing the data in this way it makes routing algorithms far simpler as it is very easy to quickly determine which nodes are connected to the one that is currently being examined. Naturally this does not support edge weights however this functionality can be added via a slight modification. By storing a list of edges for each node this provides all of the benefits of an Adjacency List however also allows for weights to be added to the connections allowing for a representation of a weighted graph.

### Path Finding

Once the graph representation has been constructed an algorithm for finding the shortest path must be implemented. There are many algorithms that have been developed for finding the shortest path between two nodes in a graph and each have their own areas of expertise depending on the type of graph.

#### Breadth-First Search

This algorithm starts at one of the two nodes it is to find the route between. From there it then moves out to all of the surrounding nodes and assigns them a value of one. It then takes each of the nodes with a value of one and assigns all of the surrounding nodes with a value of two. This continues until it reaches the desired node. It is guaranteed to find the path between the two in the fewest number of hops however does not take into account edge weights.

By not taking into account differing edge weights it would make constructing a graph representing the campus to be mapped far more difficult. This is as each of the nodes would have to take roughly the same amount of time to walk between to produce an acceptable route. This becomes particularly difficult when adding lifts to the routes as waiting on lifts could potentially take several minutes. Waiting times like this mean either all of the nodes would have to be particularly far apart, which is likely not possible in a corridor with many rooms coming off of it, or there would have to be conceptually many nodes the user would move between while waiting on a lift which is essentially just adding weights in a very inefficient way.

#### Dijkstra’s Algorithm

The main problem with breadth first is that it doesn’t allow for different weights on the edges. Dijkstra’s Algorithm by comparison will always find the shortest path in a weighted graph. It works by essentially starting at one of the two nodes that the path is between and marking it as examined. Each of the nodes that it is connected to then have a number representing the weight of the edge taken to reach them assigned to them. From there the node with the lowest number associated with it is marked as examined and all the nodes it is connected to have their numbers updated to be the sum of the weight to the examined node plus the new edge unless they already have an associated number which is less. This continues until the destination node is marked as examined at which point the shortest path between them has been discovered.

#### A\* Algorithm

The A\* Algorithm is essentially a modification of Dijkstra’s algorithm that allows the shortest path to be found more quickly. It does this by adding a heuristic value to the comparison when deciding which node to examine next. An example of a possible heuristic value for this case would be the straight line distance between each node and the current destination. This means that when choosing which node to examine next it would add both the difficulty in reaching that node and how physically far away from the destination the node is. This means that instead of spreading out equally in all directions like in breadth-first or towards the easiest direction like in Dijkstra’s it would instead spread towards the destination quicker than in other directions. The algorithm relies on the fact that the heuristic function always returns a value less than or equal to the length of the shortest path from that node to the destination. Calculating a heuristic value like this can be fairly simple for some applications for example a single level mapping application could use straight line distance fairly easily. However, a multi-level mapping application such as this would face difficulties in implementing such a system. Calculating the straight line distance for example would be impossible without detailed knowledge of the heights of each building. Even if the straight line distance could be calculated it would likely be a very poor heuristic value as it could lead the search algorithm to move towards the destination node even if the quickest route would be to move in the opposite direction towards a stair case. It would still find the quickest route but in this case it would likely be less efficient than Dijkstra’s despite the additional effort required to include the heuristics.

## Software Engineering

To develop a system of this scale and ensure it is easy to maintain and extend it is important to try to use good software engineering practices throughout. This should aid development and testing easier by making the software more flexible and forgiving of change. During this report the various styles and patterns will be referred to as taught in the class CS409 Software Architecture and Design at Strathclyde University.

### Architectural Style

An Architectural Style is the highest level programming choice to be made. It is concerned with how data and functionality should be separated. For example, the “Main Program/Subroutine shared data style” in which all of the data is available throughout the program and the functionality is broken up into subroutines which can be called when needed. For this project the architectural style is for a large part dictated by the decisions that have already been made. By choosing to develop for android this means the programming language that will be used is Java. As java is best suited to the Object-Oriented architectural style this is the main style that will be used throughout the project. However due to the way the Android OS and support libraries have been built up there will also be aspects of Event-Based Implicit Invocation included throughout.

#### Object-Oriented

This particular architectural style relies on splitting the program into objects, instances of classes, which each have access to their own particular data and can perform their own actions. Objects can then communicate via method calls between them. This leads to greater levels of abstraction meaning a change to one section of code will not dramatically alter other sections within the program. An instance of this could be if one object requests the quickest route between two nodes from a second object. The second object could be changed internally, to represent the graph differently or use a different path finding algorithm, but it wouldn’t affect the first class in any way provided the nodes to be passed in and the route returned still obeyed some kind of standard definition. The best way to create a standard definition is by creating an interface. An interface is a list of the publically available method calls that other classes can use to communicate with any classes implementing that interface. Many languages that support interfaces and classes, including java, also support abstract classes which are to some extent an amalgamation of the two. They give implementations to some of their methods however leave it to subclasses to implement others.

#### Event-Based Implicit Invocation

Event Based Implicit Invocation is similar to the Object Oriented style in many ways. However, instead of communicating between components via method calls an object can register for updates based on a particular event. This is best exemplified during use of the User Interface (UI) objects provided in the Android Software Development Kit(SDK). When a UI object is created a class which implements the provided interface can register for updates about events such as it being touched or swiped. Then when the user next touches or swipes on that UI object the registered object will be notified with relevant information such as the location of the touch.

### Programming Style

A good Programming Style will provide a high-level guide to how the program should be laid out and how both functionality and data will be separated. There are many different programming styles however since re-usability was a major factor in choosing which to adopt it was decided Model-View-Controller would be used.

Model-View-Controller divides data and functionality into the three main sections mentioned in its’ name. The Model should deal with everything programmatically for example in this project that would be the graph and path finding algorithms amongst other matters. The View should determine how that information is provided to the user for example this could be a map or simply a list of instructions. It is important that out-with the defined interface for communicating that these components do not rely on each other. This means the View could be changed entirely without any changes at all being necessary in the model. The Controller deals with user input and determines how that should affect the model. An instance of this in the project could be if the user selects a new room in some way it is the controller’s responsibility to alert the model to this change in state. Again outside the well-defined interfaces the controller should be able to be changed entirely without affecting either the model or the view.

By using this programming style it means components even at a particularly high level are particularly interchangeable. This is important in case at some point during development something changes dramatically. A possible reason for this would be if instead of using Google Maps it was decided university blueprints would be more accurate. Even in this case it would mean only the view would need to be changed leaving two thirds of the code with no alterations needed despite the large change in functionality.

### Design Patterns

Design Patterns are essentially proven solutions to commonly recurring problems. Throughout programming certain problems such as how to change algorithms during runtime appear again and again. Through use of these design patterns we have a way to deal with them effectively. Below is an overview of each of the design patterns which have been used throughout the project.

#### Observer Pattern

The Observer Design Pattern is essentially a lower level version of the same principles the Event-Based Implicit Invocation Architectural Style was based on. Essentially with it one component allows others implementing a specific interface to register with it for updates. Whenever this first object determines itself to have changed in some way it notifies all of those listening for updates. Using the Observer Pattern allows for two components which hold similar data to remain consistent while also staying loosely coupled. This design pattern can be seen most clearly in the sample UML diagram shown in Figure 8.

Observer

Update()

Update(State)

Concrete Subject

getState()

Subject

addObserver(Observer)

notify()

Concrete Observer

Figure Observer Design Pattern UML

The main advantage to this design pattern is that it very strongly decouples the concrete subject from the concrete observer allowing any changes to the concrete observer to be made easily. The main disadvantage however is that it can make debugging potential problems more difficult as the flow of code becomes less obvious. There are two versions of the Observer pattern: the push model and the pull model. They both follow the same basic structure however with one major difference. The push model gives the Observer the information it requires when it is notified. By comparison the pull model simply alerts the Observer that something has happened. This means the Concrete Observer requires a reference to the Concrete Subject in order to determine what has changed and react accordingly. These differences can be seen clearly in Figure 8 with the aspects only in the push model shown in blue and those only in the pull model shown in red. The push model has the added advantage of also cleanly decoupling the Concrete Observer from the Concrete Subject.

#### Adapter Pattern

This pattern was designed to solve the problem in which two different parts of a system rely on different interfaces to do essentially the same thing. It can be particularly useful when using frameworks or libraries which have already been developed with new code that for some reason needs to have its’ own independent interface. There are two main variations of the Adapter pattern, Class and Object, however as Class relies on multiple inheritance a feature which is only available to a minimal extent in java for the purposes of this project only the Object variation will be considered. The structure of this pattern can be seen in Figure 9.

Client

Target

Adapter

Adaptee

Figure UML Representation of Adapter Pattern

In the diagram shown in Figure 9 the client needs to use the specific interface “Target” however the necessary function is actually within “Adaptee” which cannot implement this interface. In order to solve this a new class, “Adapter”, was created to take the data from client and change it into the format needed by “Adaptee”.

#### Template Pattern

The template pattern is used to define the order in which steps are carried out but leave the implementation of these steps open to be changed dependent on what is needed at the time. This allows much of the outline and other functionality of a class to be defined in one class and for a certain bit of functionality to only be defined in a subclass. The Template Pattern can be seen in Figure 10 UML Representation of Template Pattern.

Abstract Class

templateMethod()

fixedMethod1()

variableMethod()

fixedMethod2()

Concrete Class

variableMethod()

Figure UML Representation of Template Pattern

In Figure 10 the abstract class would implement the template method and both fixed methods. It would then leave the variable method to be implemented by its’ subclasses. The template method would make calls to the fixed and variable methods in the order it requires them knowing that the concrete class will be able to provide the variable methods. The concrete class then implements the variable methods allowing the class to perform the duties it needs to. The main benefit of the template pattern is that it allows multiple classes to be created quickly that differ in one area of their implementation. For example, in a drawing package it would allow multiple shapes (the abstract class) such as square, circle or triangle (the concrete classes) to draw themselves differently despite needing to perform the same actions before and after such as checking there was adequate space and alerting a listener to the change. The main disadvantage is that this cannot be changed at runtime once the class is created it will always perform that action in the same way.

#### Strategy Pattern

The Strategy Pattern is used when a particular piece of functionality needs the ability to be changed at runtime. For instance, if a camera application used an algorithm to provide photographs with a filter applied to them the Strategy Pattern would allow users to change which filter they wished to apply while the application was running. A representation of this pattern can be seen in Figure 11.

Context

Strategy

Algorithm()

Concrete Strategy A

Algorithm()

Concrete Strategy B

Algorithm()

Figure UML Representation of Strategy Pattern

Essentially how this works is by having multiple classes implementing the Strategy interface. The Context can then change what functionality to use by changing which class it calls the method on. Frequently the context will have a method allowing another class to set the strategy. This means that the context doesn’t need any knowledge of the underlying implementation and can simply use the strategy interface allowing for more strategies to be implemented and used easily.

#### Composite Pattern

A common problem within programming is that sometimes it is beneficial to be able to treat an individual object or a group of objects the same. An example of this would be an application which controls a user’s lights. It would be beneficial to group all of the lights in an area, say a room, together and then be able to use all of the functionality available for one light bulb (i.e. turn on, turn off or set timer), with the group of lightbulbs in a room. Programmatically the way to do this is the Composite Pattern. A representation of this pattern can be seen in Figure 12.

Figure UML Representation of Composite Pattern

 Client

 Leaf

 Composite

Add(Component)

Remove(Component)

 Component

Operation()

Add(Component)

Remove (Component)

As can be seen above the composite pattern has both the group of objects (the composite) and the individual object (the leaf) implementing the same interface Component. The shared interface allows the client to interact with either of them in the same way. This means it may call the Operation method and expect either of them to fulfil this in the way they see fit. There are two variations of the Composite Pattern: safe and transparent. The methods only available in the safe method are shown in Figure 12 in blue while those only available in the transparent variation are shown in red. Both have their disadvantages the safe variation means that to add or remove elements to the composite the client needs to specifically communicate with it as a composite and not through the interface. Whereas the transparent variation means that the “leaf” object will need to provide implementations for add and remove component which make no sense for it as an individual object. A light bulb can’t be added to a light bulb it has to be added to a group of light bulbs to make sense.

# Design

## Initial High-Level Design

As previously discussed the high level design should follow the Model-View-Controller programming style. To do this successfully it is important to outline exactly what each subsystem will be responsible for in this design and how they will communicate. For a full overview the high level UML diagram is shown in Appendix 2 Initial High Level UML Diagram.

### Responsibilities

The model will deal with the state of the program i.e. what the currently selected start and end nodes are, where the user is currently looking, what errors have occurred and what the current route is. It will also be the only subsystem to deal with the data as a graph and will therefore be responsible for route finding.

The view will deal with how the current state of the program will be displayed to the user for example what rooms will be displayed, how the route will be given and what inputs will be taken from the user.

The controller will deal with reacting to user input and changing the model accordingly. When it is informed of some form of user input from the view it will decide in which way this user input will affect the model.

### Communications

The communications between each of the three subsystems can be seen in Table 3.

Table High Level Communications

|  |  |  |  |
| --- | --- | --- | --- |
|  | Model | View | Controller |
| Model | n/a | Using the observer design pattern in the push variation. | n/a |
| View | n/a | n/a | View will hold a reference to Controller through the IController interface. This will allow |
| Controller | Controller will hold a reference to the Model through the IModel interface. | n/a | n/a |

By using the Observer pattern to communicate between the model and view it keeps these two sub-systems independent from one another. The controller to view and controller to model communications will pass through interfaces this will aid decoupling however not to the same extent as the Observer pattern does between the model and view. This is as instead of just passing through the data and allowing the other object to with it what it wants the components now call specific methods. This is a compromise that has to be made however because the links between the view to controller and controller to model are too complex to be broken down into an observer pattern.

## Initial View Design

The first issue that must be addressed when designing how the view should function is understanding exactly how it will display the supplied information and interact with the user. It was decided early in the project that from an aesthetic point of view it would be preferable to show the route on top of a pre-existing map.

By using a pre-existing map, it would both cut development time and ensure that the output looked pleasing. As previously discussed there are many mapping applications which could be used or extended in order to provide this functionality [8] [9]. As the application was being developed for the Android OS the users would likely be familiar with Google Maps as it comes pre-installed on most Android phones. Google provides an extensive number of library classes to aid in the production of Android applications using their maps. These classes provide ways to add the map to a section of the screen and then perform a series of actions to the map such as moving location or drawing elements on top. It also provides a number of interfaces which can be implemented and then added as listeners to the map object. These listeners are one half of the Observer pattern and provide an easy way to react to the user performing actions on the map.

To display the route to the user it was decided that showing the entire map at once would be far too overwhelming for the user so instead the concept of planes was developed. A plane is an area of the map that can be displayed for example outside or a particular floor of a building. Each plane then has a series of endpoints and midpoints associated with it. Endpoints are where the user may be trying to reach for example outside these could be a building or on a floor of a building these would be rooms. Midpoints are points that would be needed to follow the route for example entrances to rooms or points where corridors join. As Midpoints are not necessarily useful for the user unless they are on the route they are following it was decided that Endpoints would always be displayed when on that plane however midpoints would only be used to draw the route.

As discussed Google provides a variety of options for adding items to the map two in particular stood out as ways in which to provide information to the user: Polygons and Polylines. Polygons can be drawn onto the map with a variety of design choices and would be used to alert the user to the locations of the Endpoints mentioned above. Polylines are essentially lines that can be drawn between two points on the map in a variety of colours. It was decided that these would be drawn between the relevant midpoints to display the route to the user.

The main user interactions that will be required from the applications are: selecting the start and destination, starting routes and navigating around the map to discover the campus.

To select the start and destination it was decided that a variety of options should be available to the user. The choice of using Polygons to display the endpoints was also partially influenced by the ability to add a listener to the map to discover when these polygons are clicked by the user. This ability was then used to give the user the option of selecting endpoints by tapping on them. This selection would need to be displayed to the user again in some way to inform them that their selection had been successful. Two methods were employed to do this: changing the colour of the selected endpoints and adding two text boxes to the application displaying the names of the currently selected endpoints. It is also a possibility that the user won’t know the exact location of the room however and so it was also decided that they should be able to type the name of the start or destination into the provided text boxes in order to select them as well.

Starting routes is relatively simple and only requires the option to refresh the route when they wish so a button was added providing this functionality.

Navigating around the map was more difficult than providing the other user interactions. Although at a basic level swiping to move, pinching to zoom and other basic navigation interactions are provided by Google there are some more complex interactions introduced by the concept of planes. An instance of this would be when the application starts and the user is viewing the exterior of the campus they need some way to alert the application they now wish to view the interior of one if the buildings. To do this an “inOut” button was added that would be used to either move inside the last selected endpoint if the user was currently viewing the outside plane or return outside if the user was currently viewing one of the indoor planes.

Moving inside however caused a variety of new problems. Once the user was viewing one of the internal planes the map provided by google was modified to show floor plans allowing the user to see corridors, rooms, etc. However, this meant theoretically the user could enter one building and then simply swipe along to view the inside of another building. Doing this however meant the application had no way of knowing which building the user was viewing meaning it could not draw on the necessary additions. Although google does provide a listener to alert the application when a new building is focused the information available at this time was not sufficient to distinguish one building from another as all that is provided is the number of floors in that building. To stop this from happening it was decided that the user would be stopped from viewing anything but the building they had selected while on an interior plane. This was achieved by the development of a CameraLimiter object which would question the IController to determine which building the user was currently viewing and force the view to return to the bounds of that building if the user tried to leave the area.

Once inside it was also necessary to allow the user to select which floor they would like to view. Google provides a level selector toolbar to allow the user to do this however they do not provide a direct listener for this UI element. Fortunately, when a user selects a new floor and it is displayed an event is created named IndoorLevelActivated which a listener is available for. Unfortunately, this event is also created for the default floor in every building when the floor plans are revealed i.e. whenever the user leaves the Outside plane. This means that the listener needs the ability to be turned off in certain circumstances to stop the view forcing the controller to try to move to a variety of other floors which may not even be within that building.

It was decided that an easy way for the user to follow the route between planes was needed. In order to do this another google maps component was used: Markers. A marker is an icon which can be added to the map and have a listener added to detect when the user clicks on it. This ability was then used to allow the application to move to the next relevant plane when the marker was selected by the user. It was decided these markers would be added at the first and last midpoints on the route for the plane being viewed.

As discussed earlier in order to react to user interactions a variety of Google Maps interfaces needed to be implemented. As it is the Controller’s responsibility to react to user interactions ideally the controller should implement these new interfaces. However, many of the interfaces involve passing data such as Polygons or Markers to the class which implements them. This would be an inappropriate amount of coupling between the view and controller as it would reveal the view’s inner workings to the component in the form of google maps specific components. If these were exposed it would make it far more difficult to change the view in any serious way without dramatically altering the controller. In order to combat this the adapter pattern was used. For each of the google maps specific interfaces a corresponding class was developed which would take the view specific data and convert it into a form which could be understood by the Controller. They would then convey these changes directly to the Controller using the pre-defined IController interface. This usage of the Adapter pattern can be seen in Appendix 3 View UML Diagram where IController is the Adaptee and the view (or more specifically it’s google maps components) are the client.

## Initial Model Design

The model is mainly responsible for the routing in the application. It is also used to store the current state of the program as in which plane the user is viewing and what nodes have been selected.

As discussed in the Path Finding Research Section there are many possible ways in which to find the shortest path between the nodes. However, during finding the shortest path there are steps which must be taken regardless of the path-finding algorithm used – checking the start and end are valid and alerting the listeners to the new route – it is also unlikely that it would be beneficial to change the route finding algorithm during runtime. For these reasons it was decided that the Template design pattern would be used to implement the routing functionality. This can be seen in Appendix 4 Initial Model UML Diagram where the Abstract Class is Model and the Concrete Class is DijkstrasModel.

Dijkstra’s Algorithm was used initially instead of the A\* Algorithm or the Breadth-First Search because to be built easily the graph would require weighted edges ruling out breadth-first search and the A\* Algorithm requires a heuristic which as discussed is difficult in a multi-level system.

To allow the algorithm to have elements that cannot be passed through - i.e. the concept of a building as opposed to actual corridors within that building. There needs to be a check placed on each node to ensure that node can be used. This fits well within Dijkstra’s algorithm as it can be used to only update the weights of surrounding nodes when a node is being examined if the examined node passes this check.

By expanding upon the idea of adding checks to the route finding algorithm this could be used to incorporate some form of disability aid into the app. To do this a check would be added that would exclude certain nodes that are unusable for different users. As a proof of concept for a much larger expansion this check was expanded to allow different checks depending on user entry i.e. whether the node was on stairs or not. To do this the Strategy design pattern was used as it was important to allow these checks to be changed during runtime. This gives the functionality of allowing the user to choose whether or not they wish to avoid stairs or lifts and change this while they use the app. This can be seen in Appendix 4 Initial Model UML Diagram with the Model being the context and the NodeCheck interface being the Strategy. The Concrete Strategies are currently hidden within the Controller for reasons which will become apparent during the discussion of that section.

## Initial Controller Design

The Controllers main function is to add meaning to the user input. An instance of this within the project would be when the user types something in the first text box. Once this has happened it is the Controllers responsibility to understand that this means the starting position has to change and to update the model accordingly. To do this for most of the user actions is fairly simple and is just a matter of translating the action into an appropriate method call for the model. However, two user interactions require more complex thought namely moving planes and changing the routing modifiers.

The user can move planes in many ways: using the provided In/Out button to change between viewing the inside and outside of buildings, moving level once inside a building, clicking a route marker, choosing a new start/destination using the text boxes or starting a route. Many of these changes involve changing only the level or the structure. As google keeps track of what level was last viewed for a building it is important to maintain a record of this within the app to display the correct map additions for that floor. In order to do this, it was decided a collection of objects known as structures would be created. There is a structure object created for each building which contains the name of that building and the last viewed level. In addition to this it also holds a variety of information that the view can request via the cameraLimiter to ensure the app stays focused on the correct location. When an event requires a change in structure such as moving building the current structure variable can simply be updated to indicate the new structure. Changing levels within a building then becomes as simple as changing the level variable within that structure.

Changing the route modifiers is more difficult as the model requires a check to be given to it which can tell whether a node is valid or not, one implementing the NodeCheck interface. However although in the default mode this will only require a basic check that the node can be used to pass through once more options such as avoid stairs or lifts have been added this will actually need to be a series of checks. In order to provide this functionality, the Composite Pattern was used. This can be seen clearly in Appendix 5 Initial Controller UML Diagram with the controller being the client, the NodeCheck interface being the component, CompositeCheck being the Composite and BasicCheck, StairCheck and LiftCheck being the leaves. The Composite pattern was used in the safe variation as this means that the model needs no knowledge that the composite pattern is being used at all. The controller handles all of the composition and creation and simply passes the model a check to use.

## Start-Up

## Data Requirements

The main problem with the design outlined so far is the massive amount of data it would require. To use the design as given above requires three major groups of data: Endpoint data, Midpoint Data and Link Data. Although none of this data is difficult to obtain constructing it and putting it into the correct form could be time consuming.

To display the endpoints in the way outlined in the view section it would require a list of latitudes and longitudes that would make up the polygons to be drawn. These would then have to be paired with the names of the rooms/buildings and the plane that they should be displayed on. Although this isn’t necessarily a lot of information it does scale up quite significantly with the area being mapped. For example, just to outline three floors and a small subset of buildings within the campus requires almost fifty endpoints. Each endpoint then requiring a minimum of four precise latitude and longitude positions with more being required if the room/building isn’t perfectly square as many of them are not.

Acquiring the precise co-ordinates of the rooms and buildings is not difficult and it can be done using the google maps web interface. They can be obtained by clicking on the points that are to be used and then copying the co-ordinates when they appear into a text file. This is however particularly time consuming and can take up to an hour to outline only six endpoints. This is particularly concerning given that to map the entirety of the John Anderson Campus would require hundreds of endpoints. In order to efficiently create this data a second application was created which allowed the user to enter the name and plane of an endpoint using text boxes and then draw a shape on top of google maps to allow them to create a new endpoint. These new endpoints were then written to a file which could be added to the original application.

As discussed earlier Midpoint data is also vitally important to the success of the application these are the points which the route should be drawn between i.e. where a room meets a corridor or two corridors meet. These require similar data to endpoints in that they need a name and plane on which they can be used. However, differently from endpoints instead of requiring a list of co-ordinates to be drawn they only require one latitude and longitude each. This again could be performed manually but with only three indoor floors and a small subset of the outdoor area within the university being mapped there was over 150 midpoints required.

In order to efficiently create this data a similar third application was developed which allows the user to enter a name and plane for the midpoint then place it on top of google maps using a tap. As these are easily placed incorrectly the new application has the added functionality that allows newly placed midpoints to be moved and renamed easily. Once a new set of midpoints has been finalised they are saved to a text file which can then be added to the main application.

The final set of required data for the implementation outlined above is Link data. This is essentially a list of which midpoints and endpoints are joined together. As well as determining the connections this link data must also give an attached weight to these connections to allow the routing algorithm to find the shortest path. This data could have been created manually however in the small subset of the John Anderson Campus mapped so far over well over 200 links have already been required. In order to efficiently create this data a fourth application was developed. This new application is very similar to the main application in that it allows full navigation throughout the map. However, it has three textboxes instead of two. The first two display the currently selected mid/endpoints and the third allows the user to enter a weight for the new edge between them. The new edge can then be added by pressing a button. Once a user selects a first mid/endpoint all of the edges from it are drawn onto the map. Originally the idea was to manually enter the edge weights based on how long it took to walk between them however it quickly became apparent this was going to take too long. Instead an algorithm was developed for working out roughly how long it would take to walk that distance.

To calculate the length of time it would take to walk in a straight line between two points two pieces of information were required: a distance in latitude and longitude and the length of time taken to walk that distance. A straight line distance of 443ft was measured using Google Maps. This was between the co-ordinates (55.861574, -4.245258) and (55.861330, -4.243139). Using Pythagoras’ Theorem this gives a straight line distance of 0.0021330018752922. The average walking speed of the expected user is roughly 4.95ft/sec according to the Road Engineering Journal [23]. This means the time taken to walk 443ft would be 443/4.95 = 89.49 seconds. This means that if the time taken is divided by the distance it will give a number that can be used to convert from distance to time.

When two points are selected in the Link Builder application the estimated weight is given in a text box however the user can edit this allowing links between floors such as using stairs to be put in manually.

# Testing Strategies

# Performance Analysis & Future Proofing

If the application is to work over an entire campus or similar area, it must be shown that it is likely to continue to function at an acceptable level when the map size increases. There are two main areas which have been identified as potential problems as the map size increases: the routing algorithm and drawing the map additions.

## Routing Performance

As the map size increases it will likely

Full Graph

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Description | No. of Endpoints | No. of Midpoints | No. of Links | No. of Planes | Average Routing Time (ms) |
| Full Graph | 45 | 164 | 240 | 4 | 7 |
| Minus RC2 | 25 | 95 | 153 | 3 | 5 |
| Minus LT | 26 | 98 | 120 | 2 | 3 |

CONSIDER SMALL TEST WHICH FINDS ALL OF THE POSSIBLE ROUTES FOR EVERY NODE THEN REPORTS THE AVERAGE TIME TAKEN TO FIND THE ROUTE. RUN THIS FOR WHOLE MAP THEN MAYBE TRY TAKING OUT A LEVEL OR TWO SEE WHAT HAPPENS TO AVERAGE ROUTING TIME

Could potentially take over a second. Probably ok much bigger campus though you might want to add some kind of waiting thing to show the user it’s working.

CHECKS??

## Drawing Performance

At first glance it would seem as if when the number of endpoints increases there could be an impact on the length of time it takes to display the map. However, as the view depends on the concept of planes as discussed earlier only one plane will ever be shown at a time. This means regardless of how big the overall map becomes in terms of drawing times it will only ever depend on the maximum number of endpoints per plane. Due to the nature of the application this is going to be a relatively low number, presumably either the number of buildings in the campus or the number of rooms on a floor. As a precaution a test was developed to find out how long it would take to draw a varying number of endpoints. The results of this test can be seen in

|  |  |
| --- | --- |
| Number of Endpoints | Time taken to draw (ms) |
| 10 | 25 |
| 100 | 5 |
| 1000 | 3 |
| 10000 | 2 |
| 100000 | 3 |
| 1000000 |  |

# Future Work

Although in its’ current form the application should work for a large scale campus there are certain features which could be added or extended to improve the finished product.

## Improving the Route Modifiers Implementation

The Route Modifiers are currently able to identify stairs, lifts and could be easily extended to avoid other hazards such as hills. Route Modifiers could be incredibly useful as discussed later in the Future Applications section. However, the method of implementing the route modifiers could be improved given more time. The current implementation relies on a naming convention in which only nodes that are on stairs or lifts have the term “Stair” or “lift” in the name. This was created mainly as a Proof of Concept and shows that using route modifiers in a final application is possible.

As discussed during the design section it is very easy to modify this functionality and change the checks without affecting either the model or the controller in any way due to the modularity of the design. To implement this feature better all that would be required would be to add extra fields and get methods to the Node object. These could then be set up with whether or not they are available under the different route modifiers on creation and the NodeCheck objects could query the fields rather than check the names.

The difficulty involved in this, and the reason it hasn’t been completed so far, is not in the main app but in the “Start-up” section. To implement this properly it would require changing the file reading section to build the graph in the new format. Although this in itself is not difficult it would mean the text files it uses would be in the wrong format and therefore new text files would be needed. The applications mentioned earlier (EndpointBuilder and MidpointBuilder) would need to be altered heavily to provide the new functionality.

All of the difficulties mentioned above would be easy to overcome. They would however, require more time than is currently available.

## Multiple Route Options

Currently most of the competing routing applications offer multiple options for the route you can take between the selected points. Implementing multiple routes could be useful in the app for a variety of reasons. One such reason would be to show users alternative routes in case a particular path is blocked.

Multiple routes could be very simply added to the application because of the way the code has been designed. The view would need a very small tweak to change the section currently drawing a route, to instead loop through the provided routes drawing each in a different colour. The RoutePlannerState class could then be modified to carry this list of routes to the view.

Updating the view to display the routes as outlined above leaves only one problem for this extension: how to determine what these multiple routes should be? One possible way to determine the different routes would be to use the concept of Disjoint paths. Princeton University defines Edge Disjoint paths as paths which “have no edge in common” [24]. Using disjoint paths would be a relatively simple way to provide multiple routes. Implementing the disjoint method of multiple routes would involve creating a copy of the graph. Then once the shortest path had been found, likely using the existing algorithm, these edges could be removed from the graph and the algorithm could be repeated until new paths could be found.

There are several problems with using disjoint paths to create the multiple routes. The first problem is that within buildings typically some edges have to be taken for example the path out of a room with one door will always involve using that door. If the definition of disjoint path was applied to the graph as it is built at the moment it would almost never find an alternative to provide to the user.

Another way would be to use the tags that would be included during the route modifiers section to use different criteria to find the multiple routes. An example of using these tags would be to provide the shortest path in a traditional timing sense, the path which spent the smallest amount of time on stairs and the path that spent the smallest amount of time outside. Creating multiple paths based on different criteria the user may be interested in would provide a more meaningful choice.

Implementing an algorithm to provide these different routes should be fairly simple and would only require editing a small section of the model. To change the model to deal with the new functionality would simply involve changing the section in which it uses the edge weights while calculating the shortest path. To find the path which spends the least amount of time on stairs would mean using zero as the edge weight for most edges and only using the standard edge weights for the edges which involve stairs. A more lenient system could also be developed which simply applies a stronger weighting to using stairs than not using stairs. By only applying a stronger weighting system instead of only weighting those edges on stairs the algorithm would produce a path which would still use stairs if it meant avoiding a ten minute detour around the building.

## Reaching More Platforms

Phonegap?

## Release and Updates

Best to talk about being encorporated into other applications rather than being a standalone application. Still a good idea to look into the process of releasing and updating on the two big app stores.

## Adding Location Services

Finding the wifi locations and triangulation.

# Future Applications

There are many potential uses for the application both in its’ current state, provided the additional data can be obtained, and even more if some of the progress alluded to in the future work section is completed.

## Without Extensions

As it is currently the application could be useful in a wide variety of circumstances to many different departments within the University of Strathclyde and potentially any institution mapped indoors within google maps.

### Recruitment

The University of Strathclyde, and indeed many other University Campuses, faces an issue when trying to attract new students. Although campus maps are provided and signs are placed throughout the University it can still be difficult for prospective students to find the rooms events are being held in. Making this application widely available to prospective students would enable them to find the event’s far more easily. In addition to this it may even make them more comfortable joining the University if they feel they will be able to navigate around it easily. Finally, it will also reduce the workload placed on the university as the extra directional signs will no longer have to be posted.

### Conferences

As with any visitor’s conference attendees will not be expected to be familiar with the campus. This can make it difficult for them to find their way around. By supplying a version of the mobile application created during this project navigation could be made far simpler. A room number could simply be given to the attendees and with the application it could be assumed they would easily be able to find the room.

### Deliveries

The application is easily envisioned in a delivery setting. Again this is largely as the University doesn’t tend to have one obvious and easy to find front desk. As a result, deliveries can potentially be made throughout the university. This can cause serious problems as the people making the delivery may not be familiar with the campus making it difficult to find the correct location. The “avoid stairs” option may also be particularly useful if the delivery is difficult to manoeuvre. TODO Ask Derek about it?

## With Extensions

By altering the application very slightly and improving upon certain aspects there are even more usages that can be envisioned easily.

### Disability Services

By adding the ability to avoid hills or other such obstacles as well as the already implemented “avoid stairs” feature, the application becomes very powerful in allowing mobility impaired users to navigate around the university. Currently, navigating around large campuses such as the John Anderson campus at Strathclyde can be very difficult for students, staff or visitors if they can’t use stairs or climb steep hills.

At the University of Strathclyde this is partially solved by a combination of the DisabledGo application discussed in the Possible Competition section and helpers who will guide the student. By using this application with the small improvements suggested the guides could far more easily discover these routes before starting and even check them while they follow the path given. TODO make it more obvious you carried out an interview

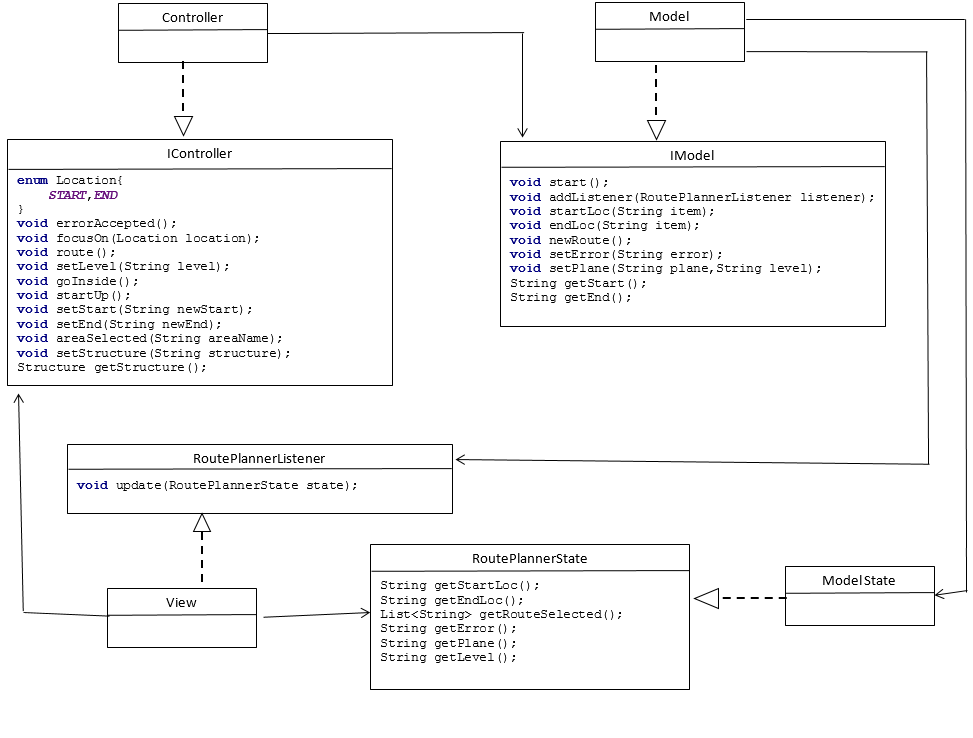
### Fire Safety

Within Non-Domestic properties in Scotland, and indeed most of the United Kingdom, it is the responsibility of those in control of the premises to provide evacuation plans for all of the people in the building [24] [25]. This is typically carried out through the creation of a Personal Emergency Egress Plan (PEEP) for individual disabled visitors or the creation of standard PEEPs in busy public areas. For a large campus this potentially means detailing the quickest route out of a building for every room in every building under a variety of different conditions such as avoiding lifts or stairs. This could potentially take a long time however with a very small modification to allow the application to route to any external location the fire safety officer could potentially create these far quicker. By adding an additional option to allow the user to select and area to avoid as well this would essentially create dynamic PEEPs constantly available on the users’ phone.

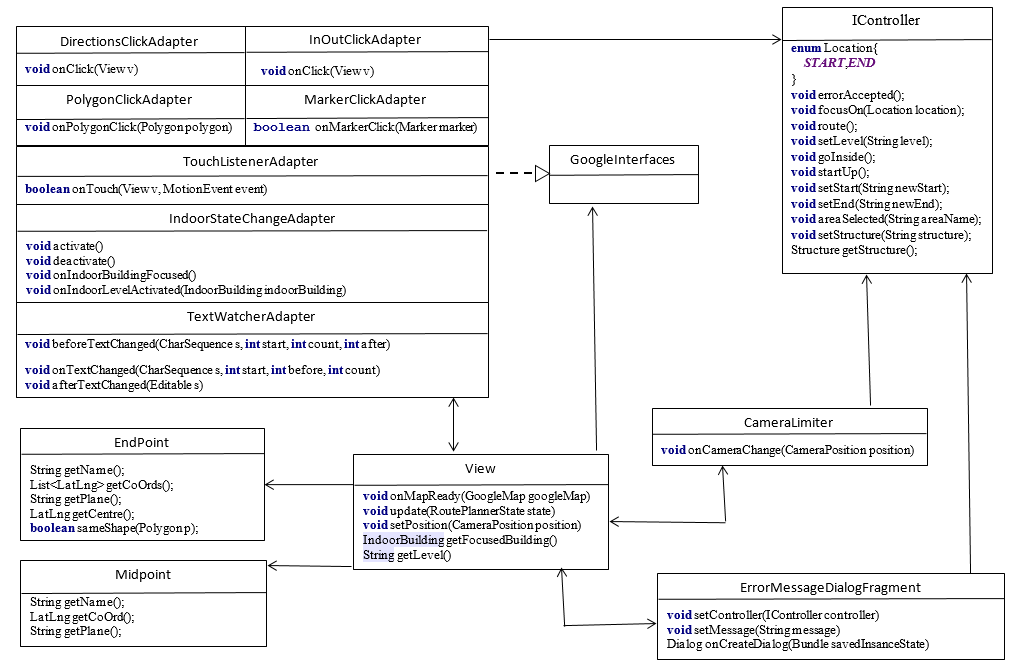
# Appendices

## Appendix 1 Strathclyde Disability Adviser Interview

## Appendix 2 Initial High Level UML Diagram

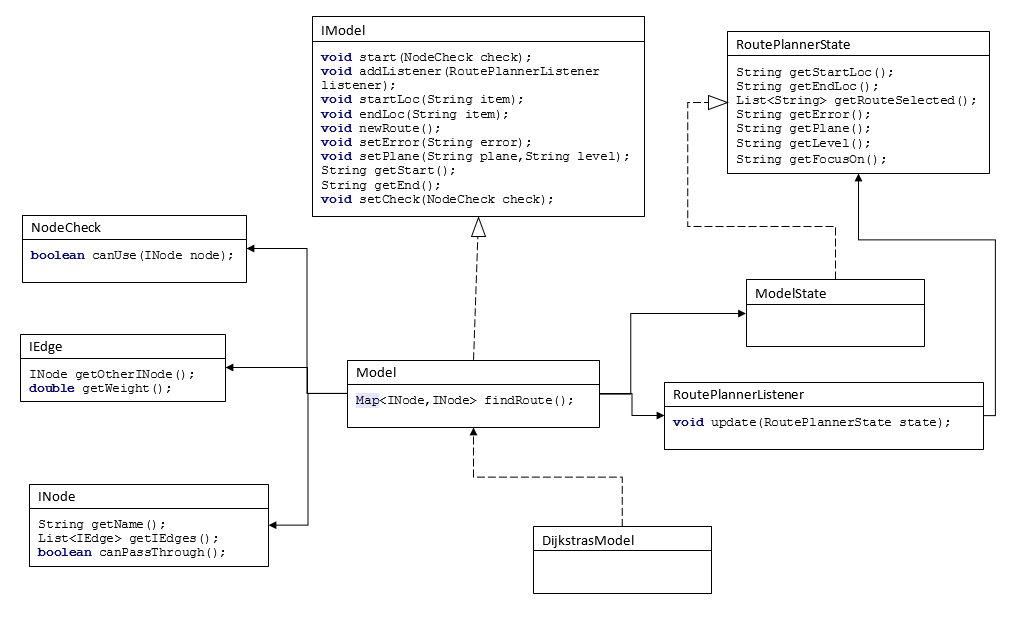


## Appendix 3 View UML Diagram

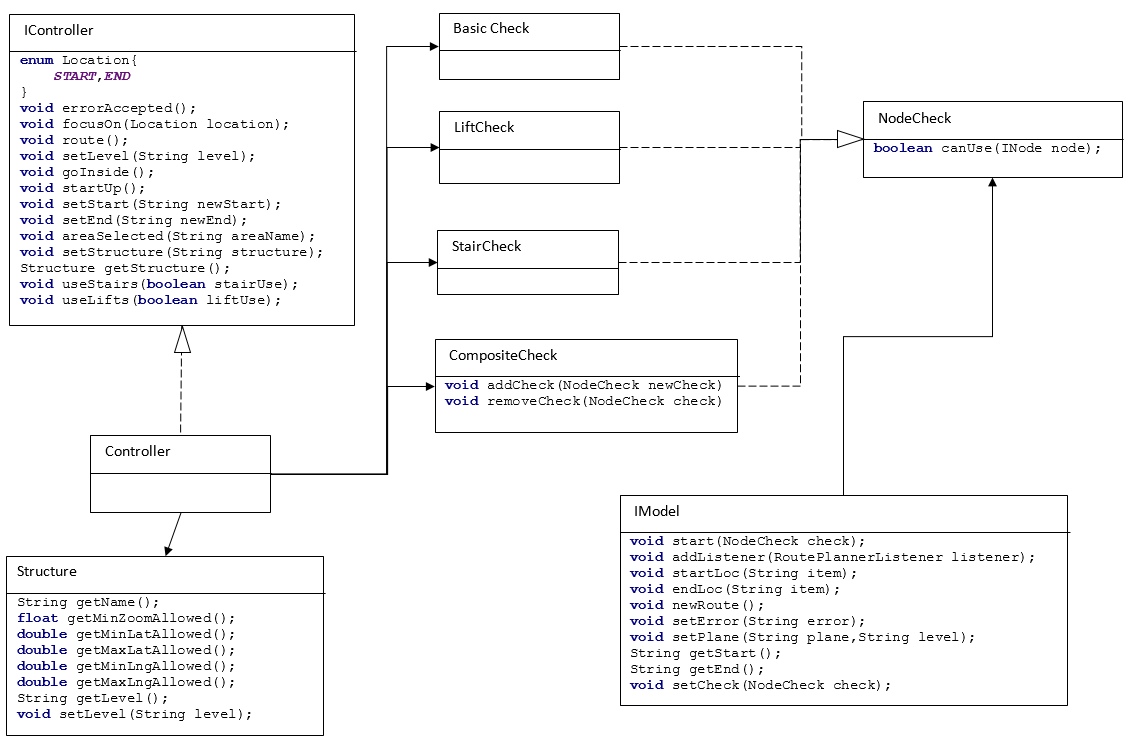


Note: Each of the adapters have their own google interfaces however they were treated as a block to simplify the diagram

## Appendix 4 Initial Model UML Diagram



## Appendix 5 Initial Controller UML Diagram



## Appendix 6 Final UML Diagram

## Appendix 7 User Guide

## Appendix 8 Installation Guide

# References

|  |  |
| --- | --- |
| [1] | P. Viswanathan, “about tech,” about, 15 December 2014. [Online]. Available: http://mobiledevices.about.com/od/additionalresources/a/Native-Apps-Vs-Web-Apps-Which-Is-The-Better-Choice.htm. [Accessed 08 March 2016]. |
| [2] | w3schools, “w3schools,” [Online]. Available: http://www.w3schools.com/html/html5\_geolocation.asp. [Accessed 8 March 2016]. |
| [3] | A. De Rosa, “envatotuts+,” 8 August 2014. [Online]. Available: http://code.tutsplus.com/tutorials/html5-network-information-api--cms-21598. [Accessed 8 March 2016]. |
| [4] | Google, “Location Strategies,” [Online]. Available: http://developer.android.com/guide/topics/location/strategies.html. [Accessed 8 March 2016]. |
| [5] | International Data Corporation, “Smartphone OS Market Share, 2015 Q2,” [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp. [Accessed 8 March 2016]. |
| [6] | Google, “Google Maps SDK for iOS,” [Online]. Available: https://developers.google.com/maps/documentation/ios-sdk/. [Accessed 8 March 2016]. |
| [7] | The Bing Maps Team, “New Bing Maps iOS SDK,” Microsoft, 5 May 2011. [Online]. Available: https://blogs.bing.com/maps/2011/05/05/new-bing-maps-ios-sdk/. [Accessed 8 March 2016]. |
| [8] | Google, “Google Maps Android API,” [Online]. Available: https://developers.google.com/maps/documentation/android-api/. [Accessed 8 March 2016]. |
| [9] | R. Brundritt, “Bing Maps Android SDK available on Codeplex,” 31 March 2011. [Online]. Available: https://blogs.bing.com/maps/2011/03/31/bing-maps-android-sdk-available-on-codeplex/. [Accessed 8 March 2016]. |
| [10] | 4 March 2016. [Online]. Available: https://www.google.co.uk/maps/dir/Glasgow/London,+UK/@53.6628863,-4.4369792,7z/data=!3m1!4b1!4m13!4m12!1m5!1m1!1s0x488815562056ceeb:0x71e683b805ef511e!2m2!1d-4.251806!2d55.864237!1m5!1m1!1s0x47d8a00baf21de75:0x52963a5addd52a99!2m2!1d-0.1277583!2d51.507350. |
| [11] | M. Rouse, “TechTarget,” September 2005. [Online]. Available: http://searchnetworking.techtarget.com/definition/GNSS. [Accessed 7 March 2016]. |
| [12] | E. K. a. C. Hegarty, “GPS Overview,” in *Understanding GPS: Principles and Applications*, Artech House, 2005, pp. 3-4. |
| [13] | GPS.gov, “GPS Accuracy,” U.S. Air Force, 18 September 2014. [Online]. Available: http://www.gps.gov/systems/gps/performance/accuracy/. [Accessed 7 March 2016]. |
| [14] | T. Bowler and M. Wall, “BBC News,” BBC, 26 August 2014. [Online]. Available: http://www.bbc.co.uk/news/business-28779524. [Accessed 7 March 2016]. |
| [15] | European Space Agency, “European Space Agency,” 18 December 2015. [Online]. Available: http://www.esa.int/Our\_Activities/Navigation/The\_future\_-\_Galileo/What\_is\_Galileo. [Accessed 7 March 2016]. |
| [16] | K. M. Pesyna, Jr, R. W. Heath, Jr and T. E. Humphreys, “GPS World,” 2 February 2015. [Online]. Available: http://gpsworld.com/accuracy-in-the-palm-of-your-hand/. [Accessed 7 March 2016]. |
| [17] | iPosi, “iposi.com,” 2014. [Online]. Available: http://www.iposi.com/tech-talk/the-technology. [Accessed 7 March 2016]. |
| [18] | D. Vaughan, “Geographic Location,” in *Windows Phone 7.5 Unleashed*, Sams Publishing, 2012, pp. 529-552. |
| [19] | J. Yang, A. Varshavsky, H. Liu, Y. Chen and M. Gruteser, “Accuracy Characterization of Cell Tower Localization,” UbiComp '10, New York, 2010. |
| [20] | Federal Communications Commission, “Accurate Location Detection 911 Help SMS App,” 2015. |
| [21] | S. J. Vaughan-Nichols, “ZDNet,” 16 November 2011. [Online]. Available: http://www.zdnet.com/article/how-google-and-everyone-else-gets-wi-fi-location-data/. [Accessed 9 March 2016]. |
| [22] | T. Cormen, D. Balkcom and K. A. C. C. Team, “Representing Graphs,” [Online]. Available: https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs. [Accessed 9 March 2016]. |
| [23] | TranSafety, Inc, “Study Compares Older and Younger Pedestrian Walking Speeds,” *Road Engineering Journal,* no. October, 1997, p. 1, 1997. |
| [24] | Safer Scotland Scottish Government, “Practical Fire Safety Guidance: The Evacuation of Disabed Persons from Buildings,” [Online]. Available: http://www.gov.scot/Resource/0040/00402451.pdf. [Accessed 22 March 2016]. |
| [25] | HM Government, “Fire Safety Risk Assesment Means of Escape for Disabled People,” [Online]. Available: https://www.gov.uk/government/uploads/system/uploads/attachment\_data/file/422202/9446\_Means\_of\_Escape\_v2\_.pdf. [Accessed 22 March 2016]. |

[Figure 1 Static Routing Example [10] 8](#_Toc446416796)

[Figure 2 Dynamic Routing Example 9](#_Toc446416797)

[Figure 3 Cell Tower Triangulation 11](file:///C:\Users\kbb12\Documents\Uni\Project\UniversityRoutePlanner\Report.docx#_Toc446416798)

[Figure 4 Outdoor Location Example 12](#_Toc446416799)

[Figure 5 Indoor Location Example 12](#_Toc446416800)

[Figure 6 Example Graph 13](file:///C:\Users\kbb12\Documents\Uni\Project\UniversityRoutePlanner\Report.docx#_Toc446416801)

[Figure 7 Example Adjacency Matrix 14](#_Toc446416802)

[Figure 8 Observer Design Pattern UML 17](file:///C:\Users\kbb12\Documents\Uni\Project\UniversityRoutePlanner\Report.docx#_Toc446416803)

[Figure 9 UML Representation of Adapter Pattern 18](#_Toc446416804)

[Figure 10 UML Representation of Template Pattern 19](#_Toc446416805)

[Figure 11 UML Representation of Strategy Pattern 19](#_Toc446416806)

[Figure 12 UML Representation of Composite Pattern 20](file:///C:\Users\kbb12\Documents\Uni\Project\UniversityRoutePlanner\Report.docx#_Toc446416807)

[Table 1 Comparison of Native and Web Applications on Key Issues 5](#_Toc446416808)

[Table 2 Comparison of Android and iOS Applications on Key Issues 7](#_Toc446416809)

[Table 3 High Level Communications 21](#_Toc446416810)