UNIVERSITY of LIMERICK

OLLSCOIL LUIMNIGH

Driverless Taxi Scheduling

Name: Darren Stack

Student ID: 13129783

Year: 2017

Supervised by Dr. Patrick Healy

B.Sc. Computer Systems
Department of Computer Science and Information Systems

Acknowledgements

I would like to offer my sincere thanks and gratitude to my supervisor Patrick Healy who proposed this project. I wish to thank him for his time, help and ideas which were of great help to me and also helped improve the overall quality of the project.

Many thanks also to everyone in the Computer Science Department for teaching me the skills I never knew over the past four years. Each individual's guidance has been invaluable to me.

Finally thanks to my family and friends who have supported me throughout my years in college.

Declaration

I declare this project report is my own work. Other work such as quotes, definitions and images that are not my own have been referenced and acknowledged to the best of my knowledge. I have followed the University Of Limerick referencing guide *Cite it Right* (3rd Ed).

Table of Contents

1. Project Summary

The purpose of this project is to create an automated Taxi Dispatching System as well as a companion simulator that can test and run different Taxi Dispatching algorithms to determine how they compare to one another. An automated dispatch system is one that does not rely on the driver to pick their fare. This web application will give users the responsibility of creating their own unique simulator. Users can then interact with the simulator in their own way to see how the scheduling system reacts to taxi requests. Once the user has finished making requests, they will receive results that they can then interpret in order to find their optimum system.

An automated taxi dispatch system is a system that decides which taxi is most appropriate for picking up a passenger's request, based on an algorithm. This is to suit the needs of a taxi company, city planner or those with an interest in efficient travel. This is done by creating a system that can use the overall knowledge of all taxis to schedule them, in the best possible way. What constitutes the best taxi varies from algorithm to algorithm, but the most common choices are the taxi which can pick the passenger up the quickest and the taxi that can complete the fare the quickest. By allowing fares to be shared in one taxi, each of these choices take on an added complexity as the algorithm must provide an equally effective service for all fares. Other issues such as traffic congestion must also be addressed.

The web application is broken into two layers:

a) The client-side shows a display that allows for users to create, view and interact with their own simulator, and
b) The server-side contains algorithms that can process the user's input and return the most appropriate taxi to allocate for collection, based on that input.

A Graphical User Interface (GUI) allows the user to firstly enter details to specify the type of simulator they want. The simulator will require different parameters to be entered which will define the environment that the simulator is to be based in. This simulator will then be launched with these conditions.

The user can then start requesting taxi fares, whilst detailing their pickup and drop off locations. The simulator reacts to these individual requests and updates accordingly. If they're trying to simulate a surge of taxis in one area, that functionality is there. The idea behind this is to give the user the freedom to use their simulator as they want. The user can also specify other conditions, such as if the fare is willing to share with other fares as well as how many passengers will be travelling. By adding these details, the simulator can more accurately provide the best possible route depending on the algorithm applied.

Once the simulation is finished, the overall results for all fares are calculated including;

- average time waiting for a taxi
- average time spent travelling by each taxi, and
- average length of fare for that simulation.

The results are left up to the user to make sense of and come to their own conclusion of what algorithms worked best for them. Results are stored so that they can be looked at and improved upon in the future.

The languages used in this application were Java, JavaScript, Cascading Style Sheets (CSS) and Hypertext Markup Language (HTML). It makes use of the Google Maps Application Programmable Interface (API). The Java project management tool Maven, the Eclipse Integrated Development Environment (IDE) and MySQL were used in the development of this web application.

2. Introduction

In this section I will provide an overview of the project, and the key problem areas I wished to address while completing it. In addition, I plan to discuss the reasons why I chose this particular subject area, and what I hoped to achieve on a personal level from undertaking the project.

2.1 Overview

My goal for this project was to establish a web application that can be used to create and then display a working automated Taxi Scheduling System. This system would be able to take in requests then find and route taxis in the best possible way. Classifying the optimum route is not an easy task. There are many different variables to take into account like whether to look for the closest taxi to the request or the taxi which can complete the request quickest. Finding this information can also be affected by factors such as traffic or the passenger's preferences. It is because of this that the Scheduling System doesn't make the decision of what the best taxi is by itself.

The scheduling system allows for the users to test their system by requesting taxis individually or requesting groups of taxis. They can focus on what they think represents their world most accurately. For users to be able to see how the system reacts to these requests, a representation of the taxis needs to be displayed. The system would also need to be able to provide feedback through overall results.

My idea was for this application to be accessible for people looking at this scheduling problem from both a business perspective as well as on an individual basis. To cater for this wide variety of users, the idea was to have a front-end display which allows for a customisable simulator to be made and allows for comprehensive testing. This testing includes selecting different algorithms in order to see which suits their environment better. The front-end will then talk to a back-end that will be able to process all the data and update the simulator accordingly. When the simulator is finished and the results are shown to the user, it is then up to them to take what they want from the results. It was believed that simple input should be required so it is usable for any person.

The user can customise the simulator based on the number of taxis they want to have and what environment they want the simulator to run in. This is because an efficient Taxi Dispatch System in a large city will have different needs than in a smaller city or rural area. To simulate an event a user can request a surge of taxis in one condensed area or numerous individual requests from different parts of the simulator environment in order to represent an average day.

The web application server-side is used to represent each of the algorithms. It uses the taxis attributes such as their location and their destination and the origin and destination of the requested taxi fare to compute the best taxi to collect that passenger. If there's a current fare in a taxi then that passengers fare needs to be taken into account so it doesn't take too long. There's a number of different algorithms that each produce different results depending on that algorithm's strengths.

I used the front-end as Google Maps based display that could provide a visual aid to the user. JavaScript was needed in order to use a Google Maps display. The back-end also uses the Google Maps API to do all the calculations needed. The Google Maps

API is available to be used with Java and that is the language I chose for this project. All the development was completed using the Eclipse Web Tools Platform. The project was created with the help of Apache's Maven software which has a plugin in Eclipse.

2.2 Motivation

The reason I selected this project is that I wanted to offer a balance of the different technologies I've looked at during my studies, while also bringing in some new skills that I've had an interest in but have yet to study. Driverless Taxi Scheduling provided a realm where I could do all this in an area I found captivating and cutting edge. Driverless taxis are something that have only become a reality in the past few years. With it being a developing field, it has many unsolved challenges which gave me the opportunity to delve into many areas. The scheduling conflict is one of the most obvious. The simple question of where each individual taxi should go sounds like it has an easy answer. The more I thought about it, the more questions arose such as "How do you decide which taxi should go there?", "What if they're willing to share fares?" and "How do you decide which fare should merge with another?" There are numerous questions like these which all have subjective answers that I hoped I could analyse and provide constructive and useful data on, so that whoever used the application could interpret and arrive at the best decision based on their own requirements.

My reason for using a web application for this was to be provide a dynamic where a user could interact with the system in their own way. It also allowed for front-end back-end style communication to take place which separates the different concerns associated with creating my project.

I wanted to use Google Maps for the display which uses JavaScript. JavaScript was also a language I had never used but had heard much about so I wanted to obtain an understanding of it and this project provided an outlet for that. Google Maps was the most popular smartphone app in the world as of 2013 (Globalwebindex.net, 2013) so to be able to gain a better understanding of how it works would also be a beneficial experience.

To do the backend processing I decided to use the Java client library for the Google Maps API. I am comfortable with Java from my course and saw that with the back-end computation being so key, it would be a good idea to use a language I was comfortable with. The other options for Google Maps API client libraries were Python, Node.js and Go however I had no experience with these. The Google Maps API was chosen for this project as it had so much detail that could be used. I was looking for as much information as possible in this project and the Google Maps API provides all that.

2.3 Objectives

The Final Year Project represents the longest amount of time I would have spent on one project. It is because of this that I hoped to gain some new skills as well as experience from completing this project. There was room to grow in working in a professional manner and in picking up new development skills.

I saw an opportunity to manage a large scale project in which I was doing the work from beginning to end as a vital experience. This is something that is translatable to any workplace, from designing what I want this application to look like, to what functionality I would like it to have, to seeing who I wanted it to appeal to, and how I will fit each piece of functionality in a sensible manner. It was dealing with these dilemmas and other problems that I wanted to gain experience in problem solving.

I was interested in working in a domain I had only a surface level knowledge of. If I could create software tailored towards this environment then I would gain confidence into going into more unknown business sectors. The practices and pitfalls of being in a new area could be picked up and learned from respectively.

Throughout the course of the development of this project I would be also trying to do other college work simultaneously. Time management was something that I thought would be key to the quality level of the project. If I could learn to balance my time between different aspects of my 4th year course with a yearlong project then I feel I would be comfortable with balancing many different areas of work in the future.

In terms of particular tools and languages I wanted to gain a better understanding of, JavaScript was a language I wanted to use. It's the most widely used programming language for Stack Overflow users (Stack Overflow, 2016) and I believe that in itself shows the usefulness in learning the language. By the end of this project I wanted to be competent in JavaScript particularly in relation to Google Maps since that would be what I was focusing on.

A Java build automation tool that is used in professional developments is Maven. This was a tool that helps manage your Java project and helps split up the different areas to aid further extension. Due to its use in a professional context, I wanted to gain familiarity with this tool as it may be needed in the future. As for Java itself, I can continue my development of the language through this project. From working with the Google Maps API client for Java, I would be able to practice using a client API which are prevalent in today's software environment. I have used Eclipse many times over the duration of my course but was still unfamiliar with some of its most useful plugins but this was something I felt the need to rectify.

3. Research

Research represents a critical part of my Final Year Project. Before this project I had little knowledge of how regular Taxi Dispatching Systems work and what parameters they use to select what taxis go to what locations. Due to this fact, researching this information would be highly influential to how the final product would shape up. It is because of this that it would also be effecting the technology that I would be using in the project.

3.1 Taxi-Based Research

One current Taxi Dispatch System that triggered a lot of the ideas for the project was iCabbi. This is a company formed in Dublin that offer their cloud based dispatch system to other companies. They boast a quick response time and an automation system which are two of the most important goals for the system. It’s also a very flexible system which is why they can offer high quality to a variety of companies.

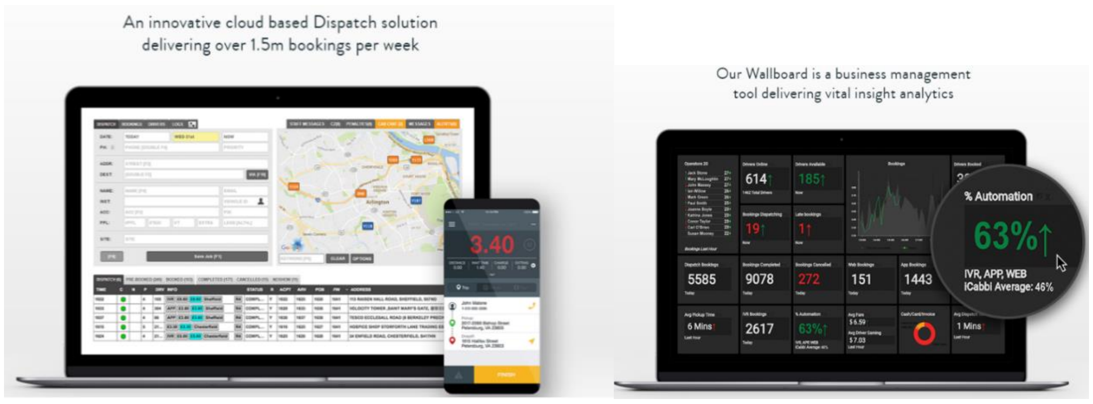

An innovative cloud based Dispatch solution delivering over 1.5m bookings per week
Our Wallboard is a business management tool delivering vital insight analytics
3.40
% Automation
63%↑
IVR, APP, WEB
iCabbi Average: 46%
614
185
19
1
5585
9078
272
151
1443
6 Mins
2617
63%
1 Mins

Figure 3.1: Screenshots of the iCabbi Simulation/ Results Page from (iCabbi, 2017)

I saw this flexibility as a quality that I could import into my own project to strengthen it. This led to the formation of a customizable Dispatch System in this project which will result in greater accuracy between different environments. Detailed analytics similar to iCabbi’s in Figure 3.1 will benefit the user when trying to see how they would improve their personal Dispatch System to suit their needs. Visually, I pictured my systems overview of the Taxi Dispatch System environment to be similar to the Google Map display in the Figure 3.1.

While iCabbi was very helpful with the overall structure of the system, I was still unaware of the finer details of the parameters to use when deciding which taxis should go where. The company Routific was a great source of information for this. This company “is a market-leading route optimization solution that helps delivery businesses save up to 40% on driving time and fuel” (Kuo, 2016). They published an article discussing different taxi practices and how effective each are. These practices ranged from the simplistic such as the nearest empty taxi picks up the fare, to the more sophisticated such as constantly re-optimizing the fleet of taxis. I was inspired by this to allow my system to allow multiple different algorithms in my product.

Another company that inspired me is Uber. Uber is the leading taxi app provider in 108 countries (Zaleski and Tartar, 2016). Uber published an article which looked at machine learning when it came to optimizing routes. The heuristics put in place for this machine learning were

- Average # of trips completed
- Average distance driven (on *and* off trips)
- Average driver earnings
- Total potential trips *lost*

As these analytics are seen as important for Uber, I felt that such data results should be given to a user in my system. While an estimate for trips lost would be difficult to quantify in a simulation like the one I am proposing, it could instead provide the average wait time for someone requesting a taxi. This is to let the user self-optimize instead of a machine optimizing. This article also showed that knowledge of the geographical area plays a large part in the creating a decent Taxi Dispatch System. According to their research

"Drivers who remain totally stationary between trips perform just as well as drivers who are constantly on the move or gravitating back toward a demand-density.", (Uber Global, 2014)

Their research conclude that drivers constantly moving randomly after dropping off a fare are much more inefficient than other drivers that are stationary or moving to an area of demand. Since this system is automated, it cannot rely on driver intelligence but can rely on the intelligence of whoever is designing the system. To cater for this, the user is able to tell taxis to move back to taxi locations or not move at all when the fare is completed.

Uber also implements a Surge feature in which the prices are changed when there is an increase in demand for a taxi. This led to the idea of being able to simulate a high demand of taxis in one area.

Separate research papers have been published that have looked at algorithms for a shared taxi service implemented in an area. A paper approached this problem as follows:

"In a shared-taxi system, the dispatch algorithm needs to find not only the best available vehicle among candidates, but also an optimal route with the newly updated schedules that avoids the violation of vehicle capacity constraints and the time window constraints of previously assigned passengers as well as the new passenger."
, (Jung, Jayakrishnan, and Park, 2013)

This approach was what this project modelled itself on. Another paper (Jung, Jayakrishnan, and Choi, 2012) broke down the logic of selecting the best possible taxi for a route.

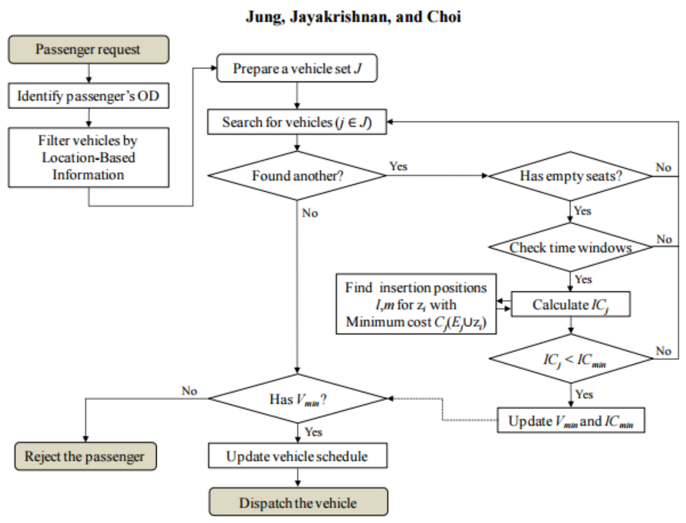

Jung, Jayakrishnan, and Choi
Passenger request
Identify passenger's OD
Filter vehicles by Location-Based Information
Prepare a vehicle set J
Search for vehicles ($j \in J$)
Found another?
Yes
No
Has empty seats?
No
Yes
Check time windows
No
Yes
Find insertion positions l,m for z_i with Minimum cost $C_j(E_j \cup z_i)$
Calculate IC_j
$IC_j < IC_{min}$
No
Yes
Update V_{min} and IC_{min}
Has V_{min}?
No
Yes
Reject the passenger
Update vehicle schedule
Dispatch the vehicle

Figure 3.2: Flowchart describing logic of picking taxi to dispatch from (Jung, Jayakrishnan, and Choi, 2012)

This method is made up breaking down all possible taxis into a smaller set that removes taxis that are too far away and should not be in consideration. Both of these papers put importance on the value of energy consumption also. When developing their algorithm they put importance on the distance covered by the taxi so that it doesn't run out of fuel (which in their case is electricity). They used a number of algorithms for their simulation and then compared there results based on factors such as profit, wait and travel times.

A separate paper (Ota et al., 2016) was influential to me as it looked at a "data-driven simulation framework that enables the analysis of a wide range of ride-sharing scenarios". With the simulation being data-driven, it allowed "historical trip data to be used to study the tradeoffs of different ride-sharing strategies". Using taxis and fares as separate entities meant that users could see statistics from both a taxi company view and a passenger view. This idea was appealing as it provided a tangible feedback to the user in order to measure different strategies

3.2 Software-Based Research

The majority of my research into the how the server-side Google Maps API works in terms of calculations of distances, times and routes was on the Google Developers webpages (Google Developers, 2016). These pages were useful in giving me many ideas of how to incorporate the Google Maps functionality into my project.

In terms of the scale of what I could hope to accomplish over the terms of my Final Year Project (FYP) I began to look at similar past FYP's to gain some sense of perspective. There was a ridesharing FYP (Kahiga, 2014) which looked at people meeting up with other people in order to create a carpool. This gave me some help in seeing what a passenger wants in a trip. Another FYP I looked at was an FYP that created a mobile application that helps find the nearest petrol station (O'Sullivan, 2013). It shared similarities with my project in that it also used the client-side Google Maps API. In seeing how this student approached using this API, I was able to integrate ideas into my own project.
This FYP was from 2013 and the improvements in the server-side Google Maps API since then are clear. The student in this FYP had to put a lot of effort in translating

requests into JSON format from Java and translate them back into Java from JSON. This was something that is bypassed by using a Java Client Library that Google provided. Where this FYP was most useful was for a Google Maps display. This app used code to allow markers to be shown to the user on a Google Map in the app based on locations. This was a feature that was a parallel to some of the features I would require in showing routes on my Google Map in a dynamic way. While I would be doing my implementation in JavaScript instead of Java due to my project being a web application instead of a mobile app, the business logic here was very valuable to me.

My main goal once I had most of the languages was how to structure my system to use them to their greatest potential. When looking at the server-side Google Maps API, it had a list of tools that could be used to incorporate it into a project. One of these tools was Apache Maven. I had not heard of this tool before but from my research it grabbed my interest as it was a professional standard tool that would help separate concerns in the development process.

4. Project Planning

This sections purpose is to show the work that was done prior to implementation. This includes choosing technologies to use, picking a development process and designing the system in the best possible way.

4.1 Technologies Used

Java:
The Java programming language was chosen due to its compatibility with the Google Maps API client libraries. Java can run on any Java Virtual Machine which means it is a highly portable language.

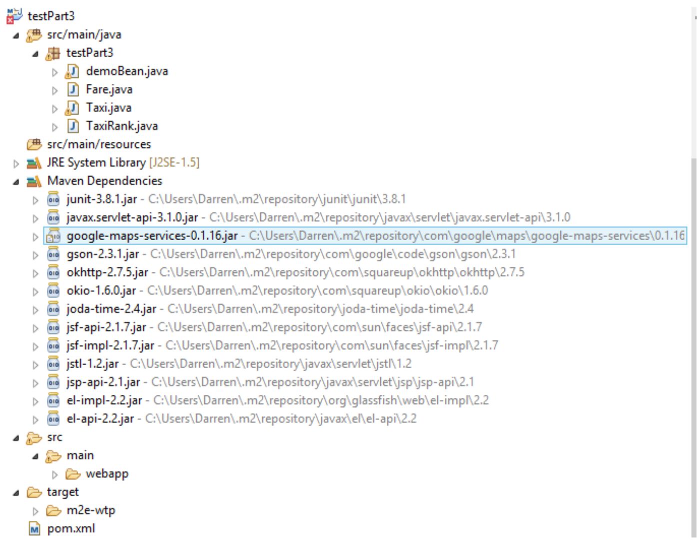
JavaScript:
This programming language is used predominately for web development. It has some similarities and was inspired by Java but has many differences in design. Google Maps is developed in JavaScript which made it an option in this project.

Eclipse IDE:
Development was done using the Eclipse Integration Development Environment (IDE) for Java EE. Eclipse has the advantage of having the ability to integrate with Maven projects. It gives the user an environment to create and compile their work while also giving helpful descriptions for warnings and errors.

Eclipse is primarily known for its Java IDE (Guindon, 2017) and as the system would be implemented using Java, Eclipse was seen as a favourable option. It also allows for web development which would cater to my needs for HTML and CSS development as each language would play a part in the product.

Java Server Faces:
Java Server Faces (JSF) is a Java based web application framework. It allows for GUI creation using JSF components. A Managed Bean can be registered to a JSF page. A Managed Bean is a Java Class that has business logic, getter and setter methods. This class can then be linked to different JSF components in the user interface.

Maven:
Apache Maven is a professional standard tool that helps separate concerns in the development process. It's a "tool that can now be used for building and managing any Java-based project" (Maven.apache.org, 2016). There's particular use in its dependency management feature which allows for the addition of API's into a project with ease.



```
testPart3
  src/main/java
    testPart3
      demoBean.java
      Fare.java
      Taxi.java
      TaxiRank.java
  src/main/resources
  JRE System Library [J2SE-1.5]
  Maven Dependencies
    junit-3.8.1.jar - C:\Users\Darren\.m2\repository\junit\junit\3.8.1
    javax.servlet-api-3.1.0.jar - C:\Users\Darren\.m2\repository\javax\servlet\javax.servlet-api\3.1.0
    google-maps-services-0.1.16.jar - C:\Users\Darren\.m2\repository\com\google\maps\google-maps-services\0.1.16
    gson-2.3.1.jar - C:\Users\Darren\.m2\repository\com\google\code\gson\gson\2.3.1
    okhttp-2.7.5.jar - C:\Users\Darren\.m2\repository\com\squareup\okhttp\okhttp\2.7.5
    okio-1.6.0.jar - C:\Users\Darren\.m2\repository\com\squareup\okio\okio\1.6.0
    joda-time-2.4.jar - C:\Users\Darren\.m2\repository\joda-time\joda-time\2.4
    jsf-api-2.1.7.jar - C:\Users\Darren\.m2\repository\com\sun\faces\jsf-api\2.1.7
    jsf-impl-2.1.7.jar - C:\Users\Darren\.m2\repository\com\sun\faces\jsf-impl\2.1.7
    jstl-1.2.jar - C:\Users\Darren\.m2\repository\javax\servlet\jstl\1.2
    jsp-api-2.1.jar - C:\Users\Darren\.m2\repository\javax\servlet\jsp\jsp-api\2.1
    el-impl-2.2.jar - C:\Users\Darren\.m2\repository\org\glassfish\web\el-impl\2.2
    el-api-2.2.jar - C:\Users\Darren\.m2\repository\javax\el\el-api\2.2
  src
    main
      webapp
  target
    m2e-wtp
  pom.xml
```

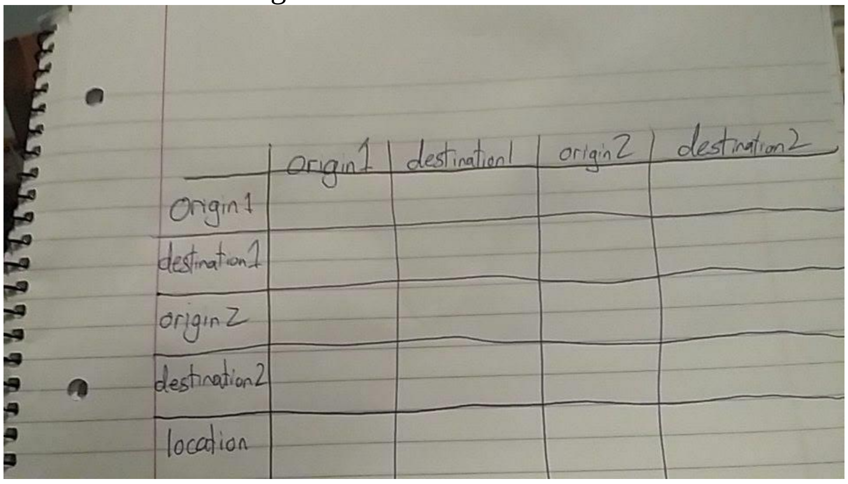
Figure 4.1: Maven project structure as shown in Eclipse

A Maven web application structure allows for different parts of a project to be segregated from another in order to manage it easier. For example the Java Code has its own area ('src/main/java' in Figure 4.1), the web pages are in a separate area ('src/main/webapp' in Figure 4.1) and the different API's being used are in another area ('Maven Dependencies' in Figure 4.1).

Google Maps Distance Matrix API:

This API requires a list of origins and destinations to be inputted from which it can then create a Distance Matrix between locations. All corresponding nodes in this matrix contain details such as the distance between locations, duration of time to travel between locations and duration of time in different traffic flows between locations. This API also has access to traffic trends so if a time and day is specified, the API can try and predict the most likely traffic for that time.

Once the details have been inputted for the Distance Matrix request, a result is given back of the travel details between any origin and destination. This data is stored in a matrix like the one shown in Figure 5.1.



	origin1	destination1	origin2	destination2
origin1				
destination1				
origin2				
destination2				
location				

Figure 4.2: Distance Matrix structure

So calculating the time taken for one valid route could be as follows:

Matrix[5][0] + Matrix[0][1] + Matrix[1][2] + Matrix[2][3]

Google Maps Directions API:
Through this API, users can create a request a route with an origin, a destination and waypoints if applicable. This request returns the result of the route which is broken down into legs and steps. Each route has one or more legs. The total number of legs depends on the number of waypoints. Each leg is then broken down into steps which indicate the different commands that need to adhered to in order to reach the next point in the route.

Waypoints seemed useful as the application would require the option of multiple fares sharing the same taxi. To do this accurately the route would need to change in order to incorporate the different pickup locations and drop off destinations. The Directions API also had the option of using traffic, unfortunately this was deprecated so it has a more inaccurate reading then the Distance Matrix.

MySQL:
This is an open source database tool that was used. It allows for customisable databases to be created. This includes customisable tables to suit the needs of the user's project. It requires installation of a MySQL server on the machine. This server and MySQL login details need to be available for any application requiring access to read/write to the database. Once this is setup the application can store and read data over numerous different sessions.

Selenium IDE:
Selenium is automated testing tool used for web applications. It can be used for recording tests then playing them back. It's available as an add-on to Mozilla Firefox. It allows for quick functionality testing to occur.

4.2 Development Process

The initial part of this project was figuring out the structure of development for the project. Due to a mixture of previous experience of working with Agile Development and with a lack of experience with long term projects, agile and particularly an iteration based approach was used. With this I have the option to change what features are implemented as the development progresses. The approach has the advantage of being flexible whereas the V-Model and Waterfall methodologies are more rigid and require more planning up-front. Planning all of the project at the start would represent risk for this system especially with my lack of initial knowledge on the fine details in this business area.

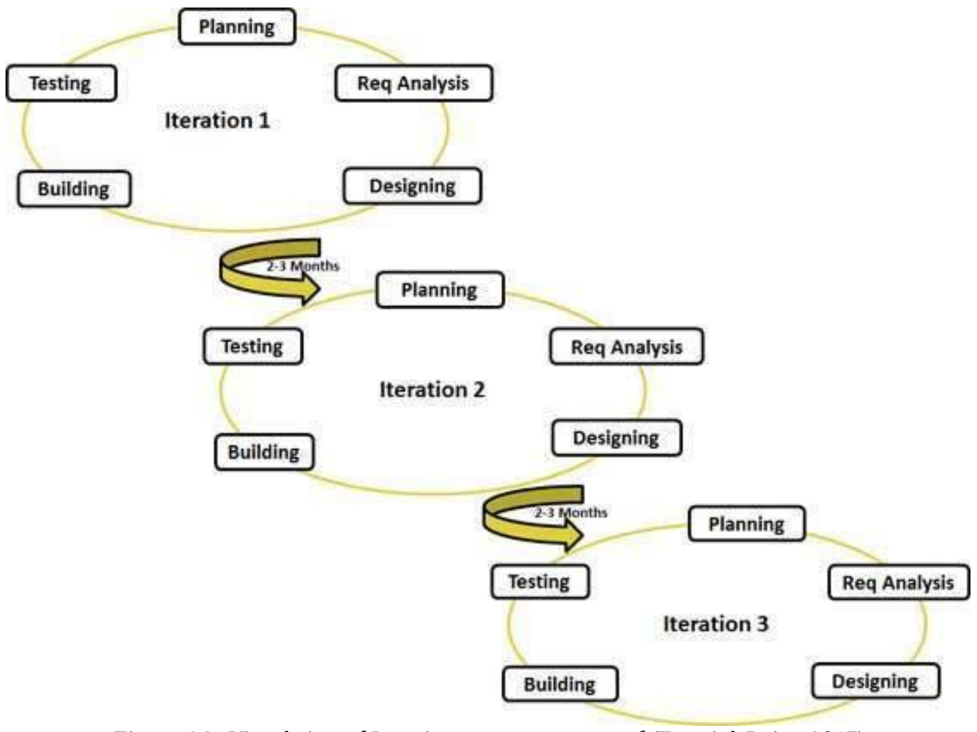


Figure 4.3: Visual view of Iterative process courtesy of (TutorialsPoint, 2017)

It is an iterative process where short term goals are defined, designed, created and finally tested. This process repeats and repeats as the system becomes more detailed. The requirements and design would be discussed between my supervisor and me. These periods of development would be short and frequent. It allows testing as the features are created which gives feedback on the status of the project as you go. As a result of this, it was decided an iteration based process would provide a better quality of product.

4.3 Requirements

The requirements of the system help specify what was going to be made. This depended heavily on the taxi-based research due to the lack of first-hand experience I had in this domain. Assessing this research led to list of potential requirements being created from the common elements of different commercial taxi dispatch systems as well as current research looking at new developments in the taxi sector. This list was then discussed by me and my supervisor in order to prioritise the requirements. This process was repeated for each of the four iterations of the system.

The list was broken into features that were felt to be must-haves, should-haves and could-haves. Must-haves were requirements that needed to be implemented into the system and if they were not, it would lead to a failure in quality in the system. Should-haves are not as critical to the project but if possible should be included. Could-haves are requirements that would improve the system if developed but would not be necessary in the final product. The iterations close to the beginning of the project focused more on the must-haves while the later iterations focused on the could-halves and should-haves.

The goals for this system were to make a realistic simulator that accurately simulate the movement of taxis in a range of different environments. The first iteration tried to establish a foundation through creating a stripped down version of the simulator talking to one basic algorithm.

Iteration 1:

1. Users must be able to view real-time simulator modelling taxis and fares.
2. Users must be able to interact with the simulator through adding new fares.
3. Users must be able to receive the best taxi for requested fare.

The second iteration increased the complexity of the simulator by allowing different algorithms to be used when finding the best taxi including two fares coming together into one taxi. It also allowed for the simulator to be customised to the user's needs.

Iteration 2:

1. Users must have options to use different algorithms for finding best taxi.
2. Fares should be merged together where possible.
3. Users must be able to customise number of taxis and location of taxis initially in simulator.

Once the foundation had been set, new features that increase the usability of the simulator could be added. This included automation of features that had already been implemented like addition of fares. It was important to give the user feedback on what went on in the simulation and allowing the simulator to be compared with other simulators with different parameters.

Iteration 3:

1. Users should be able to automate fare requests.
2. Users should have be able to get results at the end of the simulation.
3. Simulation results should be saved.
4. Simulator can be repeated.

The realism of the simulator was looked to be improved in the fourth iteration. Aspects of the taxi situation that do have a large influence on the habits of the taxi itself like traffic and the behaviour of the passenger were aimed to be replicated.

Iteration 4:

1. Simulator should allow for taxis to return back to initial location after fare.
2. Users should be able to specify time and date of simulator.
3. Shared Fares should be fair to both parties in shared fare.
4. Users should be able to specify traffic type in simulator

4.4 Architecture and Design

The Model-View-Controller architectural pattern was used for this project. This pattern splits the structure of the system. The Model is concerned with managing the data. The View establishes what the user sees and is based on the model. The Controller controls all interactions between the view and the model. The separation of concerns helps for flexibility as the project progresses which was a goal for this project.

JSF follows the MVC design pattern (Oracle, 2005). In terms of the overall project, the user interface is created using JSF components and this constitutes the view. The application data is placed in the Managed Bean which constitutes the model. The controller is the Faces Servlet as it handles all user interactions between the user and the UI.

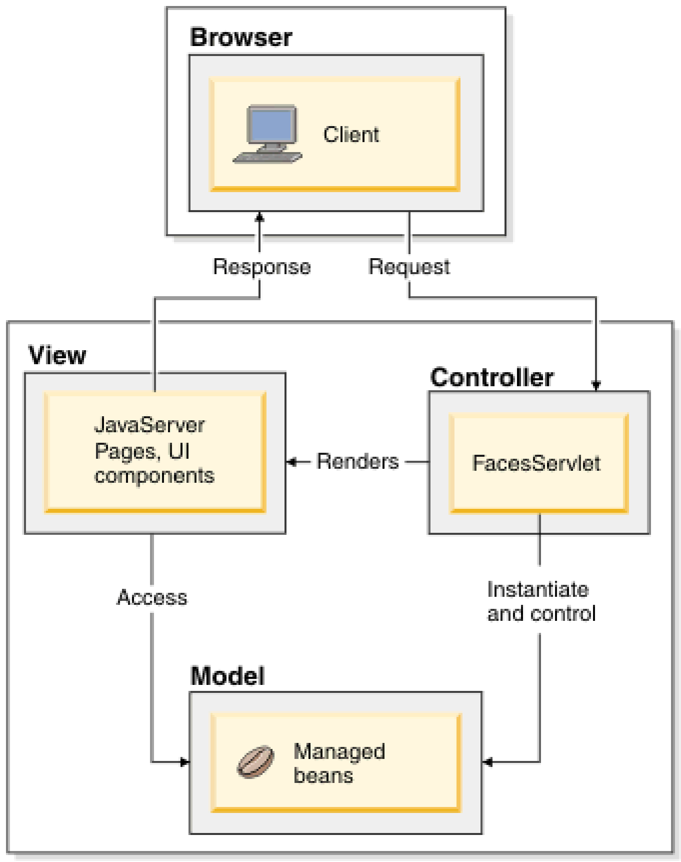


Figure 4.4: Visual description of Model-View-Controller structure in JSF (Taken from IBM.com, 2017)

The design process for this project broke down into a front-end representation of the simulator and a back-end algorithm side that would need to communicate with each other. The front-end concerned itself with visually representing the data that would be stored in the back-end. It also would have to update as changes occur to the back-end. A database is present that is used to store the data generated by the web application.

4.4.1 Front-End Design

From the research that was undertaken, I could see a visual representation was a popular approach to modelling taxis and fares. I saw it as a possible option for my system to be represented visually. This would also allow for more accurate testing of a system. As a result, taxis could be requested in ways where an expected result would be more obvious.

Google Maps was used to power many of the examples I had found. The reason for this was mainly due to the wide array of features they possess. The ability to dynamically add layers to a Google Map allows for a customisable view of the area.

Some relevant options for layers were polylines and a selection of shapes. This would allow for the system to model certain points for taxis and certain polylines for routes.

JavaScript is the language that Google Maps is developed in so the communication from the backend would need to be picked up there for it to be processed and mapped to the Google Map Display. Fortunately, due to JSF, a Managed Bean could be used in order to keep persistent data in the back-end. This data could then be accessed by the JavaScript functions stored in the front-end.

The front-end was partitioned into three different webpages. The first page is for customising the simulator, the second page is for interacting with the simulator, and the third is to show results from the simulator. Screenshots of these webpages can be seen in Figure 1, 2 and 3 in Appendix A.

Customise Simulator:

- For an accurate visual representation, the user must first provide the details they want in their simulator that will affect how it runs.
- This includes details of where the simulator will be located, how many taxis and the time and day for the simulator to run at.
- The behaviour of the taxis should also be specified by the user as it needs to be used consistently in the simulator.

Requests Page:

- The requests page allow for the user to get a view of the simulator as it is at the moment.
- It also allows for the user to then interact with the simulator through making requests, either automated or one offs and then watch how it reacts.
- Control can be provided through the running of the simulation through ‘Run’ and ‘Pause’ buttons.
- In order to provide ease of use for the user, the simulator updates the view in faster than real time.
- To give users an option to skip over watching the simulator and skip immediately to the results page, a ‘Finish Simulator’ button is displayed.

Results Page:

- The results are given both for the passenger centric statistics but also taxi and resource centric statistics.
- It gives the result of previous simulations in order to gather a comparison between simulators with different resources.
- Users have options to proceed from end of the simulator process.
- A ‘Redo Simulator’ option allows uses to recreate a previous a simulators requests but change resources.
- A ‘New Simulator’ option allows uses to start a completely new simulator.

4.4.2 Back-End Design

A back-end of the product would be needed to process input from the user as well as update the states of taxis and fares when asked. To achieve this, the back-end code is broken down into a second Model-View-Controller type structure. Entity classes were created to use the Model. This consisted of taking taxis and fares and turning them in to objects with relevant attributes. Other necessary objects were seen as taxi ranks, multiple fare requests and statistics for taxis, fares. In this scenario, the Managed Bean classes would act as control classes. This means that they would react from input made on the Front-End, interpret it and invoke the correct actions in the entity classes. The view in this case is the webpage itself as it represents the entities.

Managed Beans also have the ability to be ‘Request’, ‘Session’ or ‘Application’ scoped. This dictates how long the bean attributes and the bean itself will be stored. ‘Request’ means just as long for one HTTP request/response, ‘Session’ means just as long as the users’ HTTP session and ‘Application’ means for the applications lifespan. All Managed Beans in this product were ‘Session’ scoped as ‘Request’ scoped would provide many limitations moving from state to state. ‘Application’ was unnecessary due to the presence of a database which could store long-term results and requests which are what would be needed to be saved from the running of simulators. The ‘Session’ scope was useful for allowing the movement from the customising state to the simulator state.

This Managed Bean control class can receive calls update the simulator, which allows the taxis to be moved down its route a certain amount of time that is specified. It also checks if each active taxi to see if it has dropped someone off or if they’ve picked someone up as all these times need to be logged for results. If the user requests a fare, this control class can handles this request and check the appropriate taxis in order to find the one that suits the needs of the request.

4.4.3 Database Design

Tables needed to be created to store the relevant data from individual simulators. This work comprised of selecting the important details that were necessary to be saved. Four tables were created in all. The entity relationship diagram in Figure 4.5 shows how each table relates to the others.

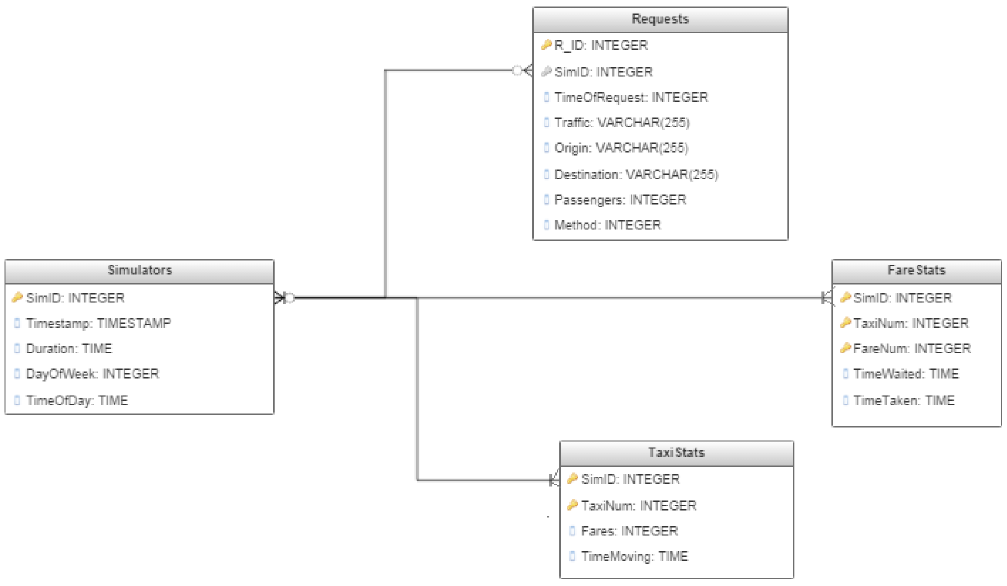


Figure 4.5: Entity Relationship Diagram of Database

Each table has to manage different data. The full tables can be seen in Appendix B. The descriptions are as follows:

Simulators:
It was necessary to give information to the user about the different simulations that have been already completed. The system time when simulator was created at is logged. How long the simulator lasted, the simulator start time and day are all useful details to know for each simulator. A unique ‘SimID’ value was given to each simulator and made a primary key. This ‘SimID’ value is used to identify the simulator.

Requests:
Each fare requested by the user is to be saved. This is because each request may need to be replicated at a later date. The simulator the request was used on is recorded. In order for the fare to be replicated correctly, the exact time in the simulator it was requested is necessary. Fare details set by the user must be saved as they make up the request.

TaxiStats:
Results for the simulator will be generated from the taxis in that simulator. The important details for each taxi are the number of fares each taxi took and how much time each taxi spent driving. The taxi number is combined with the simulation ID which creates a unique identifier for each taxi.

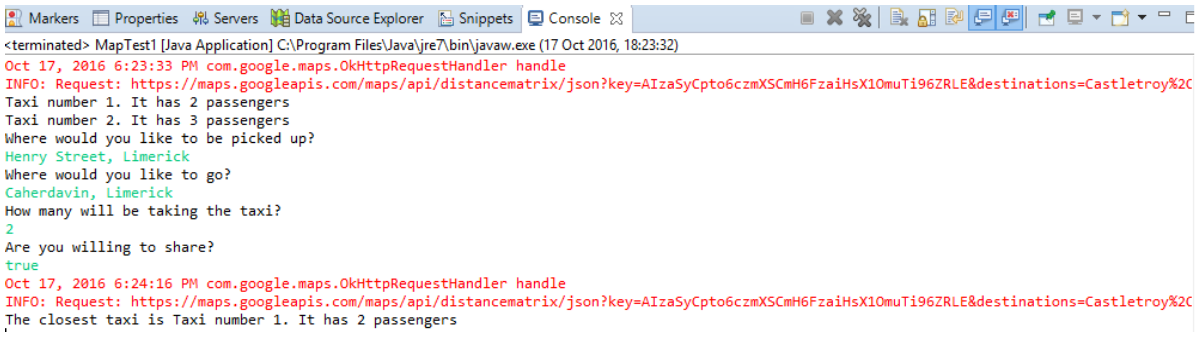
FareStats:
Results for the simulator will be generated from the fares in that simulator. To achieve this, columns were created for the amount of time the passenger waited for the fare and amount of time taken for the fare to be completed. A fare number was created which combined with the ‘taxiNum’ foreign key and ‘SimID’ foreign key columns, created a unique identifier for each fare.

5. Implementation

This section of the project covers the development of the system. The tools used in development will be defined and discussed. The key processes that shaped the dispatching system and the simulator are explained in detail over the course of this segment.

5.1 Prototyping

To get used to the different implementation technologies such as the Google API's as well as setting up projects using Maven, exploratory programs were created. This helped understand the limits of the technologies that were being used which in turn helped define the scope of the product being created.



```
<terminated> MapTest1 [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (17 Oct 2016, 18:23:32)
Oct 17, 2016 6:23:33 PM com.google.maps.OkHttpRequestHandler handle
INFO: Request: https://maps.googleapis.com/maps/api/distancematrix/json?key=AIzaSyCpto6czmXSCmH6FzaiHsX1OmuTi96ZRLE&destinations=Castletroy%2C
Taxi number 1. It has 2 passengers
Taxi number 2. It has 3 passengers
Where would you like to be picked up?
Henry Street, Limerick
Where would you like to go?
Caherdavin, Limerick
How many will be taking the taxi?
2
Are you willing to share?
true
Oct 17, 2016 6:24:16 PM com.google.maps.OkHttpRequestHandler handle
INFO: Request: https://maps.googleapis.com/maps/api/distancematrix/json?key=AIzaSyCpto6czmXSCmH6FzaiHsX1OmuTi96ZRLE&destinations=Castletroy%2C
The closest taxi is Taxi number 1. It has 2 passengers
```

Figure 5.1: Sample program using Google Distance Matrix API

These basic examples (Figure 5.1) utilised the Distance Matrix API and Directions API. Other API's were also used but were seen as not relevant to the goals of the project. From the example I was able to see how the origins and destinations could be applied to one or more taxi fares.

A separate program was focused on using the Google Map API for dynamically displaying a Google Map. Most of my learning was using JavaScript to create a map on a webpage, positioning it on Limerick (Figure 5.2) and then creating polylines to symbolize the origin and destination of taxis. From this exploratory program I could see the issue of realistically plotting the routes on the map as a straight line does not accurately show the route of a taxi.

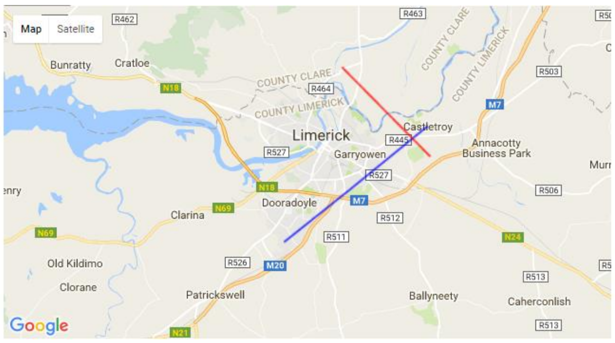


Figure 5.2: Sample view of taxi routes in a Google Map

Again this program helped me understand what some of the core issues might be when trying to implement the full product.

5.2 Implementation Process

In this section, I will break down my implementation process into the key features that shaped the system. The issues encountered will be discussed and the solutions explained.

5.2.1 Creating Simulator

The first interaction between the client-side of the project and the server-side is in creating the simulator as seen in Figure 1 in Appendix A. The user input is processed and used to start the simulator. This process entails initiating the variables that will be used throughout the duration of the simulator.

```
//sets time of sim to time specified in customization
            LocalTime timeOfDay = LocalTime.parse(startTime);
            int day = Integer.parseInt(dayOfWeek);
            LocalDate date = LocalDate.now();
            //cannot be the before current time
            date = date.plusDays(1);
            while(date.getDayOfWeek().getValue() != day){
                  date = date.plusDays(1);
            }

            timeOfSim = new DateTime(date.getYear(),date.getMonthValue(),
                        date.getDayOfMonth(),timeOfDay.getHour(),
                        timeOfDay.getMinute(), timeOfDay.getSecond());

            //sets up the taxis
            taxiList = new TaxiRank(Integer.parseInt(taxis) , startList ,
                               percentage);
            closestStatus = "";
```

Figure 5.3: Code initializing necessary variables for simulator

The start time for the simulator must be set as it affects the level of traffic congestion. Traffic will then affect the time it takes for taxis to move from point to point. A departure time is required for creating fares but a constraint with Distance Matrix API is that it needs the departure time to be in the future. Compensating for this, the nearest possible future date was found that fit the time and day that was entered by the user. This is visible in Figure 5.3.

```
public TaxiRank(int numTaxis , ArrayList<LatLng> taxiStartLocations,
                 String percentage) throws Exception{

      taxiRank = new ArrayList<Taxi>();
      int randNum;
      double percent = Double.parseDouble(percentage) / 100.0;
      for(int i = 0; i < numTaxis; i++){
            randNum = (int) (Math.random()*taxiStartLocations.size());
            Taxi t = new
            Taxi(4,i, taxiStartLocations.get(randNum).toString(), percent);
            taxiRank.add(t);
            }
      }
```

Figure 5.4: Code initializing all taxis

The code in Figure 5.4 shows how the specified taxis are created. The locations for the taxi are randomly chosen from a list of locations that were entered by the user when creating the simulator. The taxi number and the tolerance when picking up another taxi are all set in this code. A default capacity of four passengers was set for all taxis in this system.

This allows for a dynamic number of taxis to be created for each simulator and set in the correct locations as the simulator starts.

5.2.2 Displaying Simulator

It was decided that a visual display should be available so the user can see where all taxis are easily. A Google Maps display was decided on to display the current states of the taxis and their current fares. This means that the location of each taxi would have to be known and their current route. This data is stored in the 'Taxi' object in the back-end which could be accessed by the Managed Bean.

The taxi's route can be stored in the form of a DirectionsResult object which is part of the Directions API. This object has an attribute of an encoded polyline that translates the route into a displayable line on a client-side Google Map.

This data then needs to be accessible to the front-end for it to be able to display it. This was implemented through data tables in the front-end that would display the necessary information. The information is then taken from the data tables and accessed by JavaScript which creates circle icons to display the location of taxis and polyline attributes to display the current route of the taxi (Figure 5.5).

```
map = new google.maps.Map(document.getElementById("googleMap"),
                                                       mapProp);
var polyTable = document.getElementById("form:taxiPoly");
var i;
for(i = 0; i &lt; polyTable.rows.length; i++){
      var latlngStr
      =table.rows[i+1].cells[1].innerHTML.split(",");
      var lat = parseFloat(latlngStr[0]);
      var lng = parseFloat(latlngStr[1]);
      var taxiLoc = new google.maps.LatLng(lat,lng);
      var taxi = new google.maps.Circle({
                  center: taxiLoc, radius: 100,
                  strokeColor: "#0000FF",strokeOpacity: 0.8,
                  strokeWeight: 2,fillColor: "#0000FF",
                  fillOpacity: 0.4
            });
      taxi.setMap(map);
      markers.push(taxi);
      var polyString = polyTable.rows[i].cells[0].innerHTML;
      polyString = polyString.trim(); console.log(polyString);
      var polyline = new google.maps.Polyline({path:
      google.maps.geometry.encoding.decodePath(polyString),
                  strokeColor: colours[i],strokeOpacity: 0.8,
                  strokeWeight: 2,fillColor: colours[i],
                  fillOpacity: 0.35
            });
      polyline.setMap(map);
      polylines.push(polyline);
}
```

Figure 5.5: JavaScript code showing the how taxis and routes are displayed in simulator

This establishes the display side of the simulator and allows for communication of data between front-end and back-end.

5.2.3 Running Simulator

The simulator needs to run and update the taxis as they move along their routes. As these are imaginary taxis, there is no Global Positioning System (GPS) to track location, a substitute is required. This became the location variable in the taxi. To update the location, the DirectionsResult object for the taxi is used again. Each step in the route comes with an average time that step should take when driving and also the start and end location of each step. The system takes in the time elapsed since the last update as a parameter, so it can estimate what step the taxi should be at now and can update the taxi location to this location.

```
for (int i = currentLeg; i < r.legs.length && !foundPoint; i++) {
    l = r.legs[i];
    for (int j = currentStep; j < l.steps.length; j++) {
        if(legTime > legLengths.get(i)){
        //as long as it isn't the last leg
        while(i < r.legs.length - 1 &&
            legTime > legLengths.get(i) ){
            passengerUpdate(r.legs[i].endAddress);
            currentLeg = i + 1;
            currentStep = 0;
            travelTime+= (int) (legTime - legLengths.get(i));
            secondsWithoutMovement = (int) (legTime -
            legLengths.get(i));
            legTime = (int) (legTime - legLengths.get(i));
            //check if  remaining legTime smaller than next
            //leg. If not keep checking
            if(legTime < legLengths.get(i+1))
                foundPoint = true;
            i++;
        }
        j = l.steps.length;
        }
        else{
        stepMovement += l.steps[j].duration.inSeconds;
        // if they have moved step in allocated time than update
        // location
        if (stepMovement > secondsWithoutMovement) {
        // if taxi has moved to new step let the amount of
        // movement be reset
            if (currentLeg != i || currentStep != j)
                            secondsWithoutMovement = 0;
        //Arrived too early
        if (currentLeg != i)
            currentLeg = i;
        else{
            currentStep = j;
            location = l.steps[j].startLocation.toString();
        }

        foundPoint = true;
        i = r.legs.length;
        j = l.steps.length;
        }
        else if (currentLeg != r.legs.length - 1) {
        //in case taxi jumps too far and misses out on a place
        long timeToNextStop = (long) 0;
        for (int n = currentStep; n < l.steps.length; n++) {
            timeToNextStop += l.steps[n].duration.inSeconds;
        }
        if (secondsWithoutMovement > timeToNextStop) {
        i = r.legs.length;
        j = l.steps.length;
        }
        foundPoint = true;
        }
    }

    }
}
```

Figure 5.6: Code showing how the taxi is updated to find current location

Some limitations with this technique were that the Directions API times were not as accurate as the Distance Matrix API times. However the Distance Matrix did not break the times up to steps. To combat this, a check was put in place (Figure 5.6) that if the elapsed time is greater or equal to the Distance Matrix time of the leg. If so, the taxi should move to the end of that leg. In order to stop the taxi moving too quickly, checks are also in place to stop the taxi from moving to the next leg until the elapsed time is greater than the Distance Matrix time. This process helps give accurate times for the taxi trip.

This update code needs to be called automatically for the simulator to run smoothly. This was achieved through using JavaScript timers (Figure 5.7) to trigger the invocation of the code every 10 seconds. This time was chosen as it gave ample time for changes to be made to the states while also allowing for frequent updates to the user.

```
function setUp(data) {
      if (data.status === 'success') {
            myVar = setInterval(myFunction, 10000);
      }
}

function myFunction(data) {
      var button = document.getElementById("form:hdnBtn");
      button.click();
      update();
}
```

Figure 5.7: 'setUp' initiating timer to call 'myFunction' which will then invoke update code

After invoking the update code in the backend, there's also a call to re-render the taxis and fares in the Google Map display as they may have changed.

Once a taxi reaches the end of a leg, a call is made to check which fare should be picked up or dropped off. The time of this check is logged so that statistics can be collected.

```
public void passengerUpdate(String point) throws Exception{
      int  correctTime , offset;
      offset = legTime - legLengths.get(currentLeg);
      tempSecondsWithout = offset;
      correctTime = travelTime - offset;
      travelTime -= offset;
      //checks if stop was pickup or destination
      for(int i = 0; i < Fares.size();i++){
            if(Fares.get(i).checkForPickup(point , context)){
                  Fares.get(i).setPickup(correctTime);
            }
            else if(Fares.get(i).checkForDestination(point , context)){
                  //if destination then fare is removed
                  Fares.get(i).setDropoff(correctTime);
                  removeFare(Fares.get(i));
            }
      }
      secondsWithoutMovement = 0;
}
```

Figure 5.8: Code to check which fare is to be picked up or dropped off

A tricky part of implementing this feature was getting accurate pickup and drop off times. The core issue was that the system updates at three minute increments. This would lead to all statistic times being at these increments which would not be precise. To compensate for this, a 'legTime' variable was used to manage how much time had passed from the start of the leg. Using the calculated time from when the route was added, the correct time of pickup or drop off can be found as seen in Figure 5.9.

A further problem was what to do with the taxi once it had dropped off all its fares. My research had led to the two contrasting methodologies of keeping the taxi stationary or moving back to a busy area. The idea was to have the busy area be one of the initial taxi locations specified in the simulator creation stage (Section 5.3.1). The decision of which one to use can be made by the user in the Create Simulator stage (Figure 1 in Appendix A).

```
if(t.getIsActive() && t.isEmpty() && moveToHub && (!t.getTravelBack())){
      double min = t.getTaxiDistance(startList.get(0));
      int index = 0;
      for(int j = 1; j < startList.size();j++){
      if(t.getTaxiDistance(startList.get(j)) < min){
            min = t.getTaxiDistance(startList.get(j));
            index = j;
            }
}
t.moveToHub(startList.get(index).toString(), timeOfSim);
```

Figure 5.9: Finds closest taxi location

If the user wants the taxis to move back, the code in Figure 5.9 shows how the closest taxi location is found. To avoid more expensive Google API calls, the distance is calculated between the taxi's current location and each individual start location using the Haversine formula (Rosettacode.org, 2017). This formula finds the distance between two coordinates. As the movement of the taxi without fares was not as critical as a taxi with fares, accuracy was sacrificed in favour of efficiency. The closest start location is selected and a route is added to the taxi to move back. In the event that the user wants the taxi to be stationary after it is finished dropping off fares, this code is not called.

5.2.4 Finding Taxi for Fare

The reaction of the simulator to fares being added is key to this project. A 'Fare' object is created from the users input. This input includes the strength of traffic congestion for the fare which could be an 'Optimistic', ' Pessimistic' or a 'Best Guess' strength. This functionality was provided by the Distance Matrix API. The origin, the destination, number of passengers and the preferred type of pickup were also required. The fare is then looked at and depending on the type of fare, is evaluated with different methods. Whether the quickest pickup time or the quickest overall trip time was chosen is the initial check for the system as these two algorithms work differently.

After this check the system looks through an array of taxis to see how quickly they can deal with the fare. One of the challenges for adding fares was trying to cover the many different scenarios that may occur when requesting a taxi. These include empty taxis, active taxis willing to share, active taxis not willing to share, taxis with space and taxis without space.

```
if (empty){
      origins = new String[] {location};
      destinations = new String[1];
      destinations[0] = fare.getOrigin();

      DistanceMatrixApiRequest request =
      DistanceMatrixApi.getDistanceMatrix(context, origins, destinations);

      request.departureTime(fare.getTime());
      if(fare.getTrafficModel().equals("PESSIMISTIC"))
            request.trafficModel(TrafficModel.PESSIMISTIC);
      else if(fare.getTrafficModel().equals("OPTIMISTIC"))
            request.trafficModel(TrafficModel.OPTIMISTIC);
      else
            request.trafficModel(TrafficModel.BEST_GUESS);

      request.mode(TravelMode.DRIVING);
      DistanceMatrix matrix= request.await();
      shortest = matrix.rows[0].elements[0].duration.inSeconds;
      System.out.println("Taxi Num: " + number + "  Time: " + shortest);
      return shortest;
}
```

Figure 5.10: Code showing how empty taxi finds time for quickest pickup

If a taxi is currently without a fare, a distance matrix is created between the taxi's location and the pickup location, and another distance matrix is created between the pickup location and destination, if necessary. The time it takes is identified and stored temporarily, then compared to the current quickest time. If the new time is quicker, that taxi is considered the best taxi.

Taxis that do not have available space to add another fare or taxis that have a fare that is not willing to share with another fare can both use the same solution. In these cases, the time for the taxi to finish its current fare and then go to collect the new fare is calculated.

```
            tempLegs.clear();
            long distanceToEnd;
//distance to end of current route
            distanceToEnd = legLengths.get(currentLeg) - legTime;
            tempLegs.add(distanceToEnd);
            for (int i = currentLeg + 1; i < legLengths.size(); i++) {
                  distanceToEnd += legLengths.get(i);
                  tempLegs.add((long) legLengths.get(i));
            }
//get distance between final destination of current fares and new fare
            DirectionsRoute r = routeResult.routes[0];
            DirectionsLeg l = r.legs[r.legs.length-1];
            String[] origin = new String[]{l.endAddress};
            String[] destination = new String[]{fare.getOrigin()};
            DistanceMatrixApiRequest request =
DistanceMatrixApi.getDistanceMatrix(context, origin, destination);
            request.departureTime(fare.getTime());
            if(fare.getTrafficModel().equals("PESSIMISTIC"))
                  request.trafficModel(TrafficModel.PESSIMISTIC);
            else if(fare.getTrafficModel().equals("OPTIMISTIC"))
                  request.trafficModel(TrafficModel.OPTIMISTIC);
            else request.trafficModel(TrafficModel.BEST_GUESS);
            request.mode(TravelMode.DRIVING);
            DistanceMatrix matrix= request.await();
            long bridge =
            matrix.rows[0].elements[0].durationInTraffic.inSeconds;

            tempLegs.add(bridge);
            distanceToEnd += bridge;
            return distanceToEnd;
```

Figure 5.11: Code showing how unavailable taxi finds time for quickest pickup

Finding out if a taxi can merge two fares into one trip proved to be one of the more difficult problems in this project. When merging the routes of two separate taxi fares, the Taxi Dispatch system must find the optimal route in terms of time taken to visit all origins and destinations of both fares. It must do this without violating the following constraints.

- The taxi cannot visit the destination of a fare before visiting the origin of that fare.
- The taxi cannot visit an origin or destination that it has previously visited earlier in the route.

The system must find all valid routes and then compare the time taken for each valid route in order to find the quickest. The system has the ability to check if any point in the Matrix is an origin or a destination due to array lists of origin and destination points.

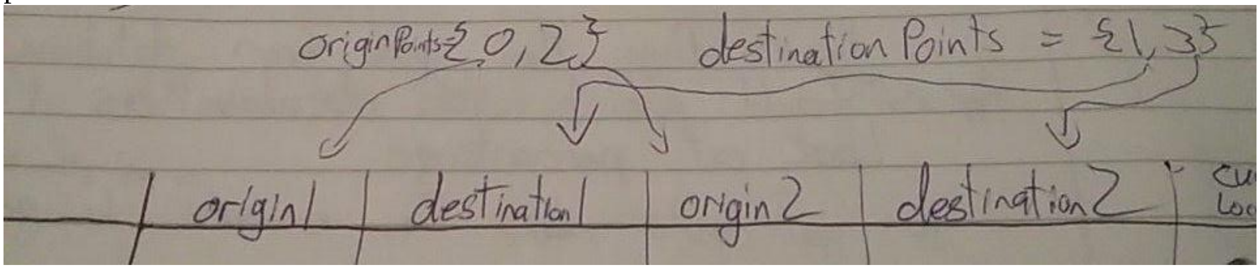
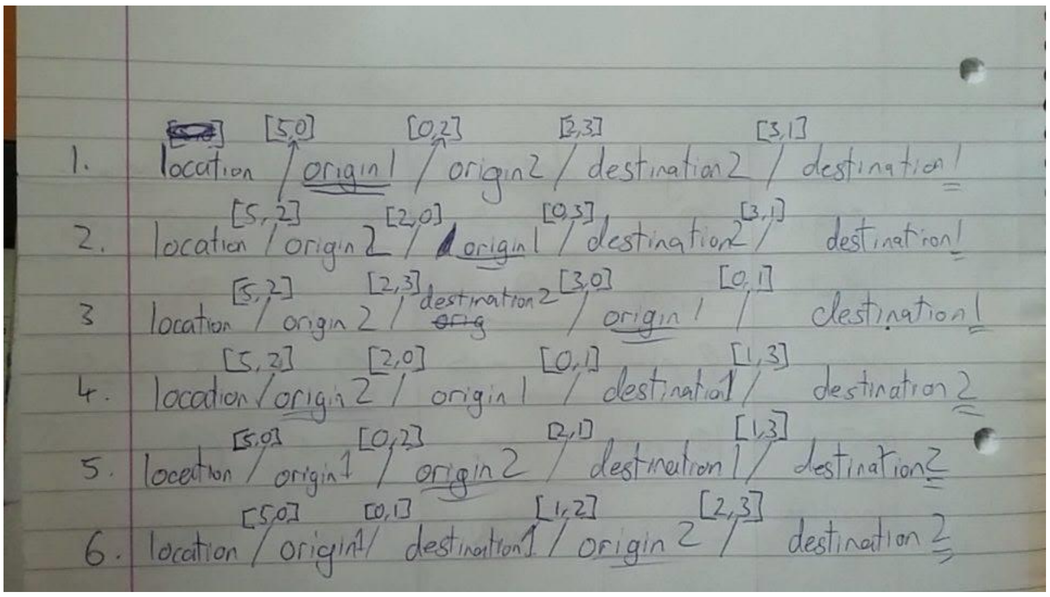


Figure 5.12: Sample setup for origin points and destination points

However, if an origin point has already been visited it is not added to the origin point's array list. This means that origin points = {2} could also be possible with destination points = {1, 3}. The distance matrix would still look like the one in figure 1. In this scenario when a taxi has already visited point 0 before the request to merge a route has come in, a sample valid route could be as follows:

Matrix[5][1] + Matrix[1][2] + Matrix[2][3]

The current solution looks at the order with the different destinations set as the last point to visit in the route, and the current location of the taxi as the start point of each route. The system then finds the corresponding origin of whatever destination is at the end of the route, and moves it through the different positions in between the first and last point. This is because the origin cannot be set to be visited after its corresponding destination. A sample search area is provided in Figure 5.13.



1. [illegible] [5,0] [0,2] [2,3] [3,1]
location / origin1 / origin2 / destination2 / destination1

2. [5,2] [2,0] [0,3] [3,1]
location / origin2 / origin1 / destination2 / destination1

3 [5,2] [2,3] [3,0] [0,1]
location / origin 2 / destination2 ~~orig~~ / origin1 / destination1

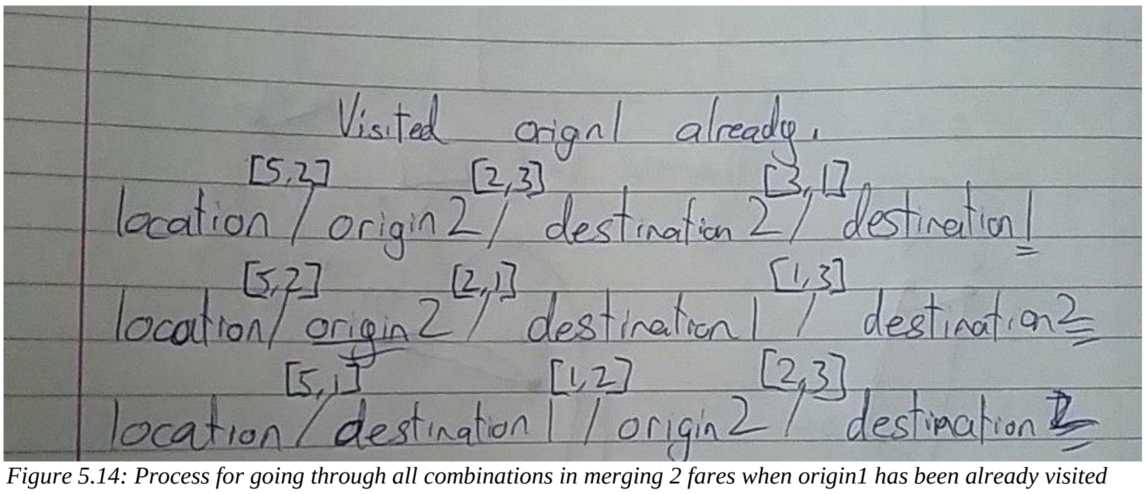
4. [5,2] [2,0] [0,1] [1,3]
location / origin2 / origin1 / destination1 / destination2

5. [5,0] [0,2] [2,1] [1,3]
location / origin1 / origin2 / destination1 / destination2

6. [5,0] [0,1] [1,2] [2,3]
location / origin1 / destination1 / origin 2 / destination 2

Figure 5.13: Process for going through all combinations in merging 2 fares

There's a check in place to see if an origin has already been visited before the route has been requested to merge. In this case the destination is put at the end of the route and the set positions are not changed so only one combination is looked at with that destination at the end. A sample search area can be seen in Figure 5.14.



Visited orign1 already.

[5,2] [2,3] [3,1]
location / origin2 / destination 2 / destination1

[5,2] [2,1] [1,3]
location / origin2 / destination1 / destination2

[5,1] [1,2] [2,3]
location / destination1 / origin2 / destination2

Figure 5.14: Process for going through all combinations in merging 2 fares when origin1 has been already visited before request

The limitations of this solution is that it only works for merging two fares together and no more.

Once this process has finished and calculated the fastest valid permutation of the origins and destinations, it is then measured against the current time that it was planned for the original route to take. This is done to ensure that there’s a limit to how much time will be added to the first fare in accommodating a second fare. This limit is specified in the creation of the simulator (Figure 1 in Appendix A).

5.2.5 Automating Requests

A feature was implemented to automate a number of fares instead of the user having to repeatedly request fares. There was a difficulty in getting the system to schedule when these fares would be called. The input required by the user was how many fares they wanted to automate and over what period of time the fares would be requested. The automated requests can have a set number of passengers for each or a random number for each fare. The other details required by the user are the same as those for a single fare such as origin, destination, method, willingness to share and number of passengers.

```
public void Multiply(){
      //simulate multiple routes at once
      int numberOfRequests = Integer.parseInt(multiple);
      int amountOfTime = Integer.parseInt(multipleTime);

      Multiples mult = new Multiples(numberOfRequests , amountOfTime,
                                     share , pickupPoint , dropoffPoint,
                                passNum , method , randomPass, traffic);

      multipleList.add(mult);
      closestStatus = "";
}
```

Figure 5.15: Multiple object used to automate multiple fares

These details are then used to create an object of Multiples type as seen in Figure 5.15. This object can then be used to create fares at the rate of:

numberOfRequests/amountOfTime

The rate is stored in the object. It is then added to an ArrayList that holds all of these Multiples objects. This is because they will need to be accessed at any time the fare is being requested.

```
      public void checkMultiples(int secondsMoved) throws Exception{
            int i =0;
            while(i < multipleList.size()){
                  Multiples m = multipleList.get(i);
            //gets the amount of requests to request during this update;
                  int portion = m.getPortion(secondsMoved);
                  for (int j = 0; j < portion; j++){
                        share = m.getWillingToShare();
                        if(Boolean.parseBoolean(m.getRandomPass()))
                              passNum = "" +((int) (Math.random()*4) + 1);
                        else
                              passNum = m.getNumberOfPassengers();
                        pickupPoint = m.getOrigin();
                        dropoffPoint = m.getDestination();
                        method = m.getMethod();
                        traffic = m.getTrafficModel();
                        Demo();
                  }
                  if(m.getAmount() == 0)
                        multipleList.remove(m);
                  else
                        i++;
            }
      }
```

Figure 5.16: Code to check multiple objects to see if they should call a fare

A method is then called at each update to see how many of the fare should be requested. This number of fares are then requested. If the full allocation of automated fares have been requested then the Multiples object is deleted from the Array List. This is due to previous inaccuracies of more than the specified number of requests being asked for.

5.2.6 Finishing the Simulator

In order to improve usability a ‘Finish Simulation’ button was created. This allows for the rest of the simulation to be finished immediately in the likely scenario that the user does not want to wait to see all taxis find their destination. All current fares have to be immediately finished. However a problem that occurs is that all fares have statistics of when the passenger is picked up and dropped off. It is because of this that the simulator has to be updated in order to finish these routes. There may be outstanding automated fare requests that have not been called yet either.

```
	public String Finish() throws Exception{
		Taxi t;
		int longestTimeLeft = 0;
		int timeLeft;
		//make sure all promised requests are made
		ArrayList<Integer> ratesOfrequest = new ArrayList<Integer>();
		while(multipleList.size() > 0){
			for(int i = 0;i < multipleList.size();i++){
				ratesOfrequest.add(multipleList.get(i).getRate());
			}
			//sorted list of rates
			Collections.sort(ratesOfrequest);
			//updates for each rate so all are called once
			Update(ratesOfrequest.get(0));
			for(int j = 1; j < ratesOfrequest.size();j++){
				Update(ratesOfrequest.get(j) -
					ratesOfrequest.get(j-1));
			}
		}
		for (int i = 0; i < taxiList.getTaxiAmount(); i++) {
			t = taxiList.getTaxi(i);
			timeLeft = t.finishUp();
			if(timeLeft > longestTimeLeft)
				longestTimeLeft = timeLeft;
			}
//if finished early, this tells you how long the simulation went on for
		totalTime += longestTimeLeft;

		taxiStatistics = new TaxiStats(taxiList , totalTime ,
				requestDetails , startTime , dayOfWeek);
		int SimID = taxiStatistics.getSimID();

		return "results.xhtml?simID" + SimID;
	}
```

Figure 5.17: Code to finish simulator

Firstly, all the rates of all automated fare requests are found and sorted as seen in Figure 5.17. They are taken one by one and the simulator is updated to satisfy each rate exactly. This is done repeatedly until all scheduled fare requests have been made. Once all fares have been allocated, each taxi must be immediately moved to their final destination.

```
      public int finishUp() throws Exception{
//simulator has been requested to finish up so route
//is fastfowarded to its end
            long distanceToEnd = 0;
            if(isActive){
                  DirectionsLeg l;
                  DirectionsRoute r = routeResult.routes[0];
                  for (int i = currentLeg; i < r.legs.length; i++) {
                        l = r.legs[i];
                        distanceToEnd += (int)legLengths.get(i) - legTime;
            //sets it up so there should be no offset so time is accurate
                        travelTime += (int)legLengths.get(i) - legTime;
                        secondsWithoutMovement += (int)legLengths.get(i) -
                                                      legTime;
                        legTime = legLengths.get(i);
                        currentLeg = i;
                        if(!travelBack)
                              passengerUpdate(l.endAddress);
                        secondsWithoutMovement = 0;
                        currentStep = 0;
                        legTime = 0;
                  }

                  isActive = false;
                  update(0);
            }
            System.out.println("DTE" + distanceToEnd);
            return (int)distanceToEnd;
      }
```

Figure 5.18: Code to move taxi to finish

The timestamps need to be recorded for the pickup and drop off. The durations are already calculated in the addition of the fare to the taxi. A loop looks at the time left in the current leg and the taxi is fast forwarded to the end of the leg where the times are logged. The next iterations of the loop concern themselves with running through all remaining legs and logging those times too.

The duration to the end of the longest remaining fare is added to the current duration of the simulator so it is accurate. On the conclusion of the simulator all statistics have to be logged to a database. This includes details about the simulator environment, taxi statistics, fare statistics and request statistics.

This data is then shown in a table format to the user. These results can be evaluated and judgement can be made. Other simulator results are also visible. The user can move through the results from other simulators (Figure 3 in Appendix A). However having individual simulator results can only mean so much to the user unless the simulator has something in common with the other simulator. This lead to the option of ‘Redo Simulator’ being given.

5.2.7 Recreating Simulator

To properly compare different taxi resources to see how they react to different environments, the simulator will have to be replayed with the same requests being made at the same time. This required the storage of each request with a timestamp of when in the simulation it was requested. Once the user inputs their new environment details the simulator runs through the simulation automatically skipping the ‘Requests’ page as the simulation requests are already defined.

```
	public String simAll() throws Exception{
		while(requests.next()){
		//goes through all requests for sim wanted to be replicated
		//and requests them at the same times with these new conditions
			Update(requests.getInt(3) - totalTime);
			Update(0);
			share = "" + requests.getByte(8);
			if(requests.getByte(8) == 0)
				share = "false";
			else
				share = "true";
			passNum = "" + requests.getInt(6);
			pickupPoint = requests.getString(4);
			dropoffPoint = requests.getString(5);
			if( requests.getInt(7) == 0)
				method = "false";
			else
				method = "true";
			traffic = requests.getString(9);
			Demo();

		}
		return Finish();
	}
```

Figure 5.19: Code calling all requests

Figure 5.19 shows that timestamps for each request that were made in the original simulator are found and ordered ascendingly. The simulation is updated to the point of each timestamp so that the taxi is in the position it should be at that time. The request is then made with the optimal taxi found. This process then restarts with the next timestamp.

The results page is then given with the new results visible alongside the old results. The user can then easily see what effect their changes have made to both the passenger and resource sides of the product.

6. Testing

Software Testing is a process that can be used to assess the quality of a system. This is done by specifying inputs and expected outputs and then evaluating these against what actually happens in the system. The advantages of this process is in finding bugs in the system. These can be identified and then fixed which improves the quality of the system. There are many different types of Software Testing and the following were influential in the testing of this system.

Manual Testing: The process of testing a system without using automated scripts or tools.

Automated Testing: The process of using tools to test a system. One such tool is Selenium which allows for record/playback features to test web applications.

Black-box Testing: A category of testing that does not look at the code when testing. The tester interacts with the system with their inputs and then compares their outputs to the expected outputs and uses this to evaluate whether it has passed or failed. It can be applied to regression, stress and integration testing.

Functionality Testing: It is focused on testing web applications in order to ensure it conforms to requirements.

Regression Testing: This method of testing is used after modifying a system to add new features. It involves testing the new features added to the system in order to show they have been implemented correctly. It also looks at features that have been tested earlier and tests them again to make sure that they still work as expected after the modifications have been made to the system.

Stress Testing: This can be used in order to find the limits of a system by applying unfavourable or extreme conditions to it. It looks at how effective the system operates under these conditions.

Integration Testing: This is a level up for Unit Testing which focuses on one unit. Integration testing puts multiple modules together and tests them. These collaboration tests are used to check if all modules are working together as expected.

White-box Testing: A category of testing that focuses on the code when testing. The test is aimed at covering the internal workings of the code. Branch coverage is a type of white box testing.

Branch Coverage: This method looks at logic decisions made in the code and goes through each possible option in order to check that each branch of code has been covered.

6.1 Testing the System

Due to the many moving parts of the system and the high amount of interaction between the different features, a substantial effort was made in testing the system to an acceptable level. Testing for this project consisted of iterative testing which occurred at the conclusion of each iteration of the product. Iterative testing has the advantage of being able to test the features as they are being added, which reduced the number of test cases being created. Once the project was complete, many sections had already been tested.

In order to log the testing, a test plan was used. A test plan is a document which details the objectives, techniques, features covered, test cases, pass/fail criteria for test cases and a list of what documents should be created from the test plan. In the next section I will discuss the Master Test Plan used for the final iteration product. The master test plan is a test plan that encompasses multiple levels of testing instead of just focusing on one.

6.2 Master Test Plan

The goal of this test plan was to provide a high standard of testing towards the final iteration of the Taxi Dispatch System and Simulator created. It aimed to achieve this through specifying what should be tested and how it should be tested while taking into account external factors that may affect the process.

A potential constraint when testing this system is that is a visual system. This provides an issue with automated scripting of tests, as the results will commonly have to be seen. The quota on Google Maps API requests represents another constraint on the how extensive the testing can be. Google Maps API has a limit to the number of requests made per day. Once the limit has been hit, the system can no longer use the API and thus cannot be used.

6.2.1 Features to be tested

- Taxi Amount Specification
- Taxi Location Specification
- Time Specification
- Taxi Request
- Multiple Taxi Request
- Simulation Control
- Bounds Specification
- Simulator Shutdown
- Simulation Redo
- New Simulation
- Results Creation
- Result Comparison

6.2.2 Approach

As it is the final iteration of the project and the final features have been added, regression testing is to be used. Regression testing was chosen as it allows for coverage of all features created up until the point of testing to be assessed equally alongside the new modules created in the previous iteration. Due to the small number of GUI elements in this application and its visual nature, manual testing will be primarily used on the simulator with automated tests created with the Selenium IDE to automate other UI tests.

The testing aims to focus on how individual modules worked together (Integration testing), and if the overall system worked as expected (Functionality testing). In addition to this, performance testing such as stress testing will be done specifically to see how it copes with a high number of requests.

6.2.3 Pass/Fail Criteria

The criterion for features to pass testing is for them to give the appropriate response to whatever action is applied, with said action being specified beforehand.

6.2.4 Deliverables

- Test Plan
- Test Cases
- Defect Log

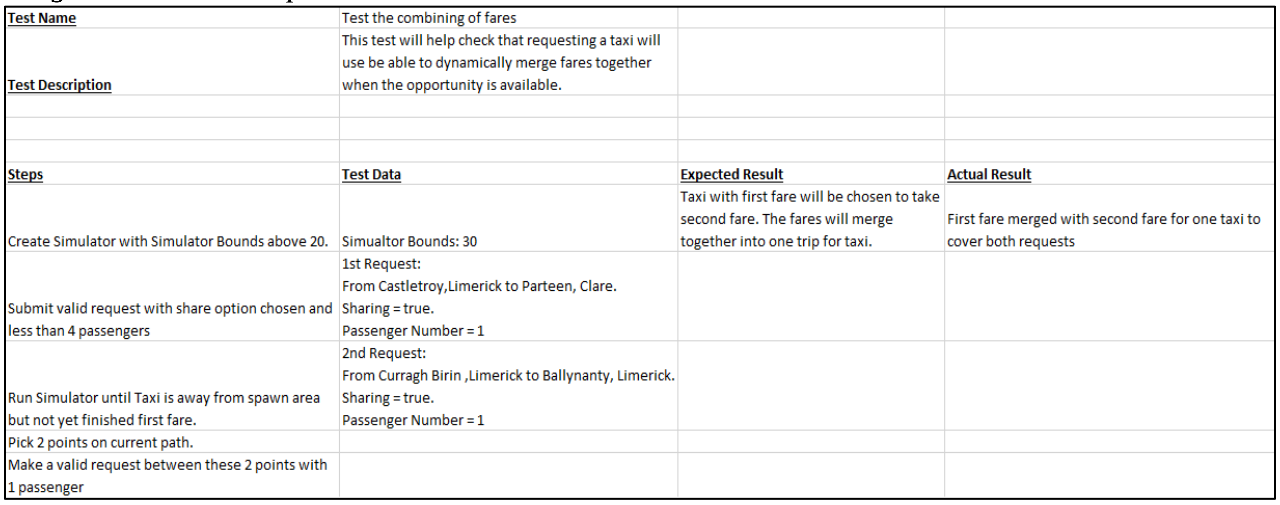
6.2.5 Suspension Criteria and Resumption Requirements

If any feature does not work as expected and the results are seen as below expectation, testing will be stopped in order to fix this feature.

Once the problem in the feature has been addressed, testing can restart. Due to the nature of the system, a full restart of the tests should take place as the changes may have affected other parts of the product.

6.3 Test Cases

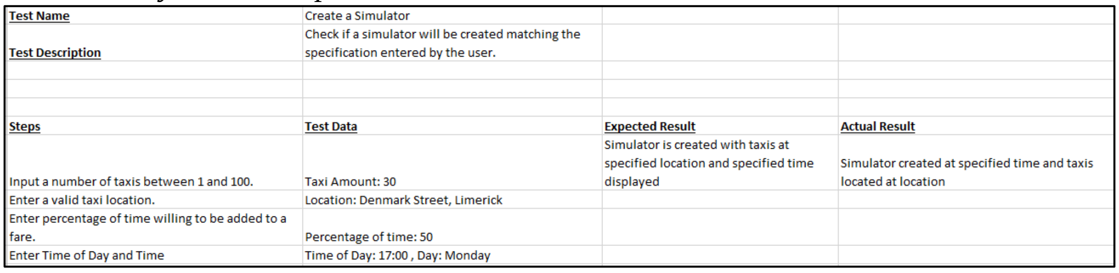
Integration Test Example:



Test Name	Test the combining of fares		
Test Description	This test will help check that requesting a taxi will use be able to dynamically merge fares together when the opportunity is available.		
Steps	**Test Data**	**Expected Result**	**Actual Result**
Create Simulator with Simulator Bounds above 20.	Simualtor Bounds: 30	Taxi with first fare will be chosen to take second fare. The fares will merge together into one trip for taxi.	First fare merged with second fare for one taxi to cover both requests
Submit valid request with share option chosen and less than 4 passengers	1st Request: From Castletroy,Limerick to Parteen, Clare. Sharing = true. Passenger Number = 1		
Run Simulator until Taxi is away from spawn area but not yet finished first fare.	2nd Request: From Curragh Birin ,Limerick to Ballynanty, Limerick. Sharing = true. Passenger Number = 1		
Pick 2 points on current path.			
Make a valid request between these 2 points with 1 passenger			

Figure 6.1: Integration Test for requesting and merging fares being tested.

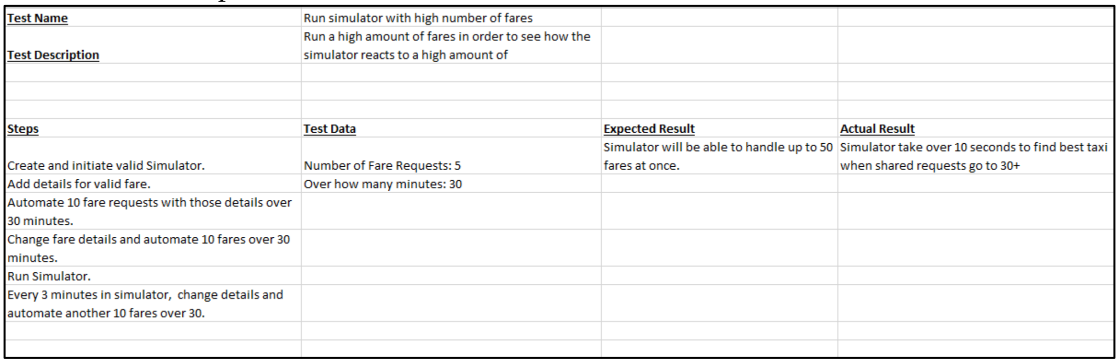
Functionality Test Example:



Test Name	Create a Simulator		
Test Description	Check if a simulator will be created matching the specification entered by the user.		
Steps	Test Data	Expected Result	Actual Result
Input a number of taxis between 1 and 100.	Taxi Amount: 30	Simulator is created with taxis at specified location and specified time displayed	Simulator created at specified time and taxis located at location
Enter a valid taxi location.	Location: Denmark Street, Limerick		
Enter percentage of time willing to be added to a fare.	Percentage of time: 50		
Enter Time of Day and Time	Time of Day: 17:00 , Day: Monday		

Figure 6.2: Functionality Test for creating Simulator tested

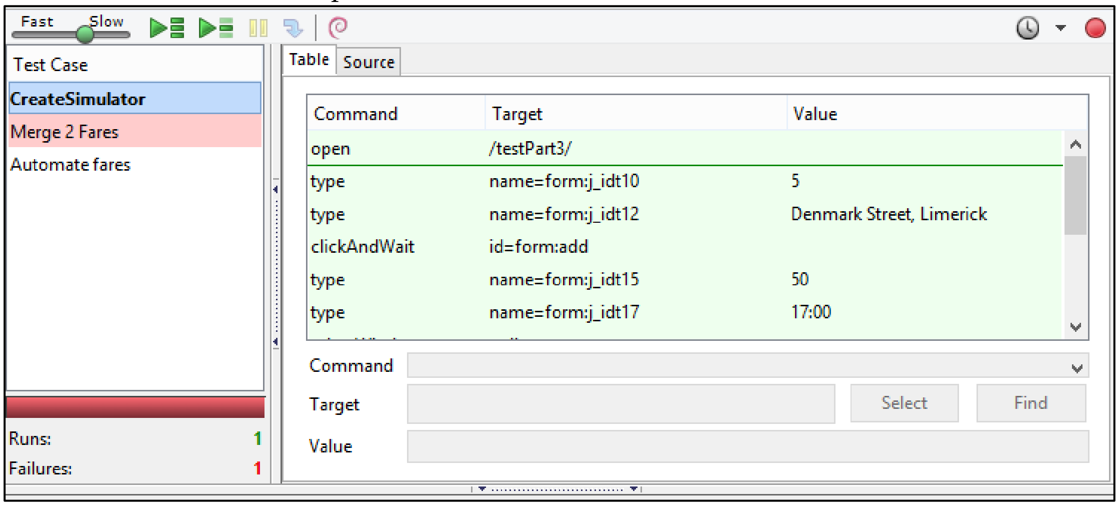
Stress Test Example:



Test Name	Run simulator with high number of fares		
Test Description	Run a high amount of fares in order to see how the simulator reacts to a high amount of		
Steps	Test Data	Expected Result	Actual Result
Create and initiate valid Simulator.	Number of Fare Requests: 5	Simulator will be able to handle up to 50 fares at once.	Simulator take over 10 seconds to find best taxi when shared requests go to 30+
Add details for valid fare.	Over how many minutes: 30		
Automate 10 fare requests with those details over 30 minutes.			
Change fare details and automate 10 fares over 30 minutes.			
Run Simulator.			
Every 3 minutes in simulator, change details and automate another 10 fares over 30.			

Figure 6.3: Stress Test for high number of fares being tested.

Selenium Test Suite Example:



Test Case: **CreateSimulator**, Merge 2 Fares, Automate fares

Command	Target	Value
open	/testPart3/	
type	name=form:j_idt10	5
type	name=form:j_idt12	Denmark Street, Limerick
clickAndWait	id=form:add	
type	name=form:j_idt15	50
type	name=form:j_idt17	17:00

Runs: 1
Failures: 1

Figure 6.4: Sample Test Suite in Selenium.

6.4 Test Results

As a result of conducting these tests I was able to identify issues and improve the system accordingly. One such example was in the results of the simulator. From functionality tests of different scenarios I could see that some results were inaccurate. This issue led to branch coverage in order to pinpoint where in the code this issue was arising so the problem could be fixed. In this instance it was due to the system not taking into account that the taxi may be dropping two fares off at the same location.

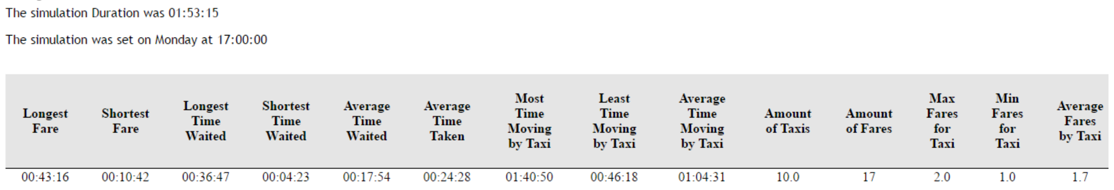
7. Evaluation

While testing showed that the product was working as expected, evaluating showed if what the system was doing was useful. This process was made up of looking at different simulators and different environments to see what results are given and then looking at differences. It was planned that these differences should give some business intelligence to the user. Finding the exact measurable differences that each parameter can make is not easy but by having one general simulator and one set of simulation requests, slight tweaks to individual parameters can show the effect that the parameter can have.

An example simulator was created for Limerick City for basic fares using locations like the University Of Limerick, Raheen Industrial Park, Shannon Airport and a selection of landmarks in the city itself like King Johns Castle, The Milk Market and Thomand Park, main streets like O'Connell Street and Henry Street were used as well. The requests varied from shared requests to not shared, heavy traffic to light traffic, and from full taxis to just one passenger.

7.1 Evaluation Tests

High Resources vs Low Resources

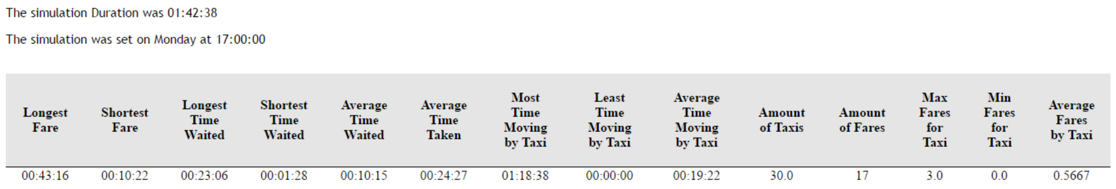


The simulation Duration was 01:53:15

The simulation was set on Monday at 17:00:00

Longest Fare	Shortest Fare	Longest Time Waited	Shortest Time Waited	Average Time Waited	Average Time Taken	Most Time Moving by Taxi	Least Time Moving by Taxi	Average Time Moving by Taxi	Amount of Taxis	Amount of Fares	Max Fares for Taxi	Min Fares for Taxi	Average Fares by Taxi
00:43:16	00:10:42	00:36:47	00:04:23	00:17:54	00:24:28	01:40:50	00:46:18	01:04:31	10.0	17	2.0	1.0	1.7

Figure 7.1: Low Resources Results



The simulation Duration was 01:42:38

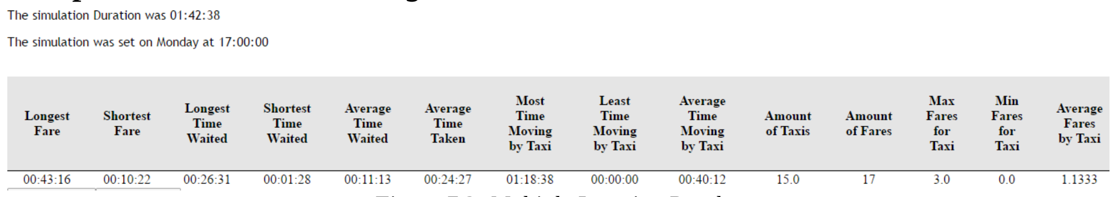
The simulation was set on Monday at 17:00:00

Longest Fare	Shortest Fare	Longest Time Waited	Shortest Time Waited	Average Time Waited	Average Time Taken	Most Time Moving by Taxi	Least Time Moving by Taxi	Average Time Moving by Taxi	Amount of Taxis	Amount of Fares	Max Fares for Taxi	Min Fares for Taxi	Average Fares by Taxi
00:43:16	00:10:22	00:23:06	00:01:28	00:10:15	00:24:27	01:18:38	00:00:00	00:19:22	30.0	17	3.0	0.0	0.5667

Figure 7.2: High Resources Results

The difference a high number of taxis has on the simulator is the 'Average Time Waited' is significantly lower than the equivalent figure for a low number of taxis. This would show that taxis are responding quicker. However the trade-off for this is that some taxis will not be used as much. This is seen in both the 'Average Time Moving by Taxi' and in all statistics focused on the amount of fares taken for a taxi. In these statistics, the low resource taxis are used for much longer than the high resource taxis.

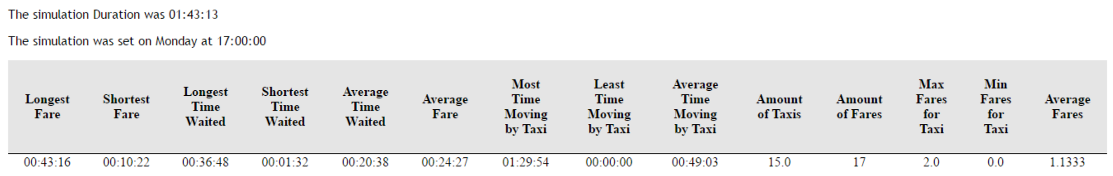
Multiple Taxi Locations vs Single Location


The simulation Duration was 01:42:38

The simulation was set on Monday at 17:00:00

Longest Fare	Shortest Fare	Longest Time Waited	Shortest Time Waited	Average Time Waited	Average Time Taken	Most Time Moving by Taxi	Least Time Moving by Taxi	Average Time Moving by Taxi	Amount of Taxis	Amount of Fares	Max Fares for Taxi	Min Fares for Taxi	Average Fares by Taxi
00:43:16	00:10:22	00:26:31	00:01:28	00:11:13	00:24:27	01:18:38	00:00:00	00:40:12	15.0	17	3.0	0.0	1.1333

Figure 7.3: Multiple Location Results


The simulation Duration was 01:43:13

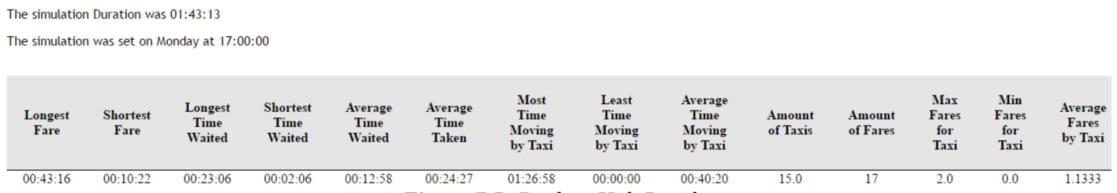
The simulation was set on Monday at 17:00:00

Longest Fare	Shortest Fare	Longest Time Waited	Shortest Time Waited	Average Time Waited	Average Fare	Most Time Moving by Taxi	Least Time Moving by Taxi	Average Time Moving by Taxi	Amount of Taxis	Amount of Fares	Max Fares for Taxi	Min Fares for Taxi	Average Fares
00:43:16	00:10:22	00:36:48	00:01:32	00:20:38	00:24:27	01:29:54	00:00:00	00:49:03	15.0	17	2.0	0.0	1.1333

Figure 7.4: Single Location Results

This comparison shows the benefit of having multiple location hubs instead of one solitary hub. With multiple locations the chances that you will have a taxi that can respond to a fare quicker and the chances that a taxi can move back to hub quicker are higher. All 'Time Moving' stats and 'Average Time Waited' results strengthen this claim. The disadvantage with spreading your resources is if there is a surge of taxis in one area, there is more chance of overflow causing taxis from the other hubs to travel over which can add to wait time.

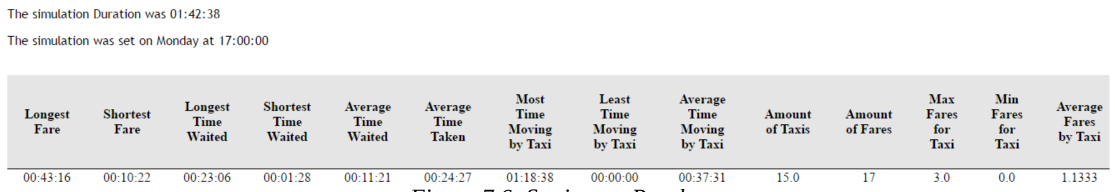
Back to Hub vs Stationary


The simulation Duration was 01:43:13

The simulation was set on Monday at 17:00:00

Longest Fare	Shortest Fare	Longest Time Waited	Shortest Time Waited	Average Time Waited	Average Time Taken	Most Time Moving by Taxi	Least Time Moving by Taxi	Average Time Moving by Taxi	Amount of Taxis	Amount of Fares	Max Fares for Taxi	Min Fares for Taxi	Average Fares by Taxi
00:43:16	00:10:22	00:23:06	00:02:06	00:12:58	00:24:27	01:26:58	00:00:00	00:40:20	15.0	17	2.0	0.0	1.1333

Figure 7.5: Back to Hub Results


The simulation Duration was 01:42:38

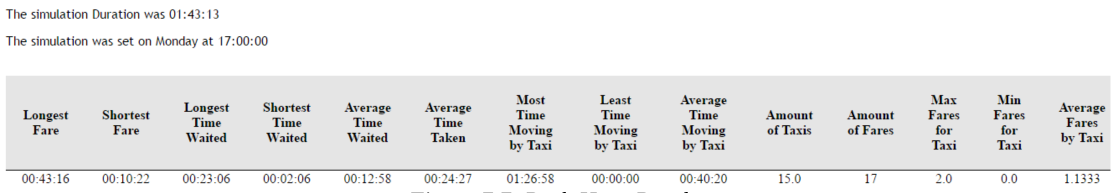
The simulation was set on Monday at 17:00:00

Longest Fare	Shortest Fare	Longest Time Waited	Shortest Time Waited	Average Time Waited	Average Time Taken	Most Time Moving by Taxi	Least Time Moving by Taxi	Average Time Moving by Taxi	Amount of Taxis	Amount of Fares	Max Fares for Taxi	Min Fares for Taxi	Average Fares by Taxi
00:43:16	00:10:22	00:23:06	00:01:28	00:11:21	00:24:27	01:18:38	00:00:00	00:37:31	15.0	17	3.0	0.0	1.1333

Figure 7.6: Stationary Results

The key difference between leaving taxis stationary once they have no active fares and having taxis move to the closest hub once they have no active fares is in the 'Time Moving' statistics. The stationary taxis spent less time moving in this relatively short simulation, although only 3 minutes less on average. In my research of stationary versus back to hub, there is no clear advantage between either schools of thought. This is reflected in these results. The time waited is slightly in favour for the stationary taxi. It's also interesting to note that taxis moving back to hubs have a more even distribution of fares judging by the 'Max Fares for Taxi' result.

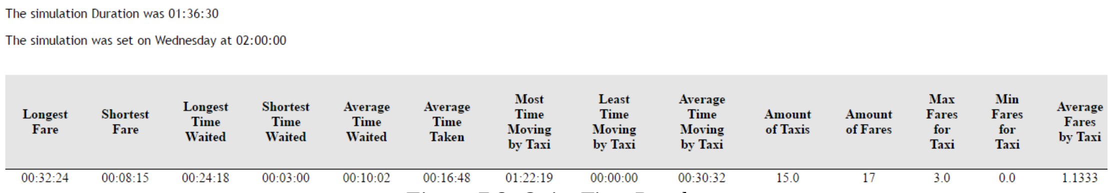
Rush Hour vs Quiet Time


The simulation Duration was 01:43:13

The simulation was set on Monday at 17:00:00

Longest Fare	Shortest Fare	Longest Time Waited	Shortest Time Waited	Average Time Waited	Average Time Taken	Most Time Moving by Taxi	Least Time Moving by Taxi	Average Time Moving by Taxi	Amount of Taxis	Amount of Fares	Max Fares for Taxi	Min Fares for Taxi	Average Fares by Taxi
00:43:16	00:10:22	00:23:06	00:02:06	00:12:58	00:24:27	01:26:58	00:00:00	00:40:20	15.0	17	2.0	0.0	1.1333

Figure 7.7: Rush Hour Results


The simulation Duration was 01:36:30

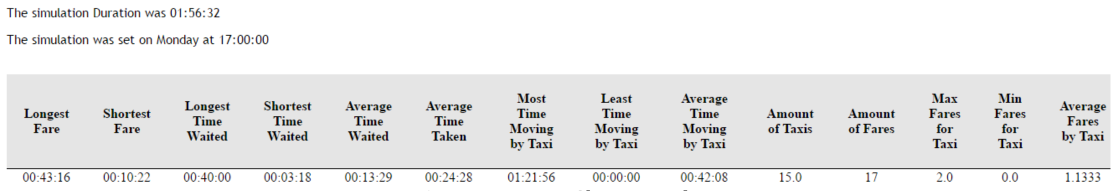
The simulation was set on Wednesday at 02:00:00

Longest Fare	Shortest Fare	Longest Time Waited	Shortest Time Waited	Average Time Waited	Average Time Taken	Most Time Moving by Taxi	Least Time Moving by Taxi	Average Time Moving by Taxi	Amount of Taxis	Amount of Fares	Max Fares for Taxi	Min Fares for Taxi	Average Fares by Taxi
00:32:24	00:08:15	00:24:18	00:03:00	00:10:02	00:16:48	01:22:19	00:00:00	00:30:32	15.0	17	3.0	0.0	1.1333

Figure 7.8: Quiet Time Results

In the case of comparing peak traffic times and quieter times, the obvious variances are between the duration of the fares. In quiet times the fares are completed quicker (‘Time Waited’/’Longest Fare’/’Shortest Fare’ statistics) and from that taxis spend less time moving (‘Time Moving’ statistics) than they would in a rush hour scenario.

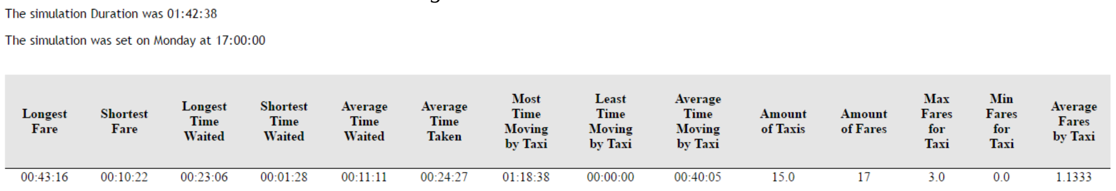
High Share Tolerance vs Low Share Tolerance


The simulation Duration was 01:56:32

The simulation was set on Monday at 17:00:00

Longest Fare	Shortest Fare	Longest Time Waited	Shortest Time Waited	Average Time Waited	Average Time Taken	Most Time Moving by Taxi	Least Time Moving by Taxi	Average Time Moving by Taxi	Amount of Taxis	Amount of Fares	Max Fares for Taxi	Min Fares for Taxi	Average Fares by Taxi
00:43:16	00:10:22	00:40:00	00:03:18	00:13:29	00:24:28	01:21:56	00:00:00	00:42:08	15.0	17	2.0	0.0	1.1333

Figure 7.9: Low Share Results


The simulation Duration was 01:42:38

The simulation was set on Monday at 17:00:00

Longest Fare	Shortest Fare	Longest Time Waited	Shortest Time Waited	Average Time Waited	Average Time Taken	Most Time Moving by Taxi	Least Time Moving by Taxi	Average Time Moving by Taxi	Amount of Taxis	Amount of Fares	Max Fares for Taxi	Min Fares for Taxi	Average Fares by Taxi
00:43:16	00:10:22	00:23:06	00:01:28	00:11:11	00:24:27	01:18:38	00:00:00	00:40:05	15.0	17	3.0	0.0	1.1333

Figure 7.10: High Share Results

When the share tolerance is high, there is more chance of getting a quicker taxi which is evident in the ‘Time Waited’ categories. The distribution of taxis is more even for low share taxis.

7.2 Evaluation Results

What was gathered from each of these tests is that changing any of the individual parameters when creating the simulator, has the potential to alter the results of the simulator itself. In a real world case, the user will not just be changing one parameter but many which gives many combinations of parameters, and thus many different possible results from just one simulator. It must also be taken into account that many different fare requests can be made in a totally new simulator also. This again provides a new context to work in where changing parameters will affect the simulator differently than it did in this example.

8. Conclusion

In the conclusion I aim to show what has been achieved from this FYP. I intend to look at how this project could be improved on in the future.

8.1 Reflection on Project

What was hoped to be achieved in this report was to help describe the complete process of how I created my Final Year Product. What I aimed to cover was:

- The research needed to be done to be able to address a problem in the taxi domain.
- The first steps in creating a system.
- The importance of addressing challenges early on in the process.
- The methodology of work that was followed in this project.
- How the product was implemented.
- The techniques that were used to assess the product.

I believe that these aspects are the key factors in giving a tangible understanding into why this project was made and why it was done in this fashion.

8.2 Scope for Future Work

There are many different areas to be considered in taxi scheduling. In terms of this project, some areas that I believe should be further analysed are multiple shared fares, optimising the search and looking at large scale versions of this product.

A limitation in this project is that it only allows for two fares to be merged together. In the future, a solution could be created that would allow for more fares to be added into one taxi trip in an efficient manner. This would enhance the quality of the project by giving yet another option for quicker travel.

There is currently no support for estimating the amount of money generated from a simulator. While there are difficulties in estimating precisely how much a fare will cost, it would improve the realism of the simulator. To add to this, more taxi categories could be included, similar to Uber’s system. Their system breaks down to “Economy” cars which are cheap and “Premium” cars which are more expensive (Uber.com, 2017). Cars of different sizes can be requested and this could be implemented into this FYP system as well.

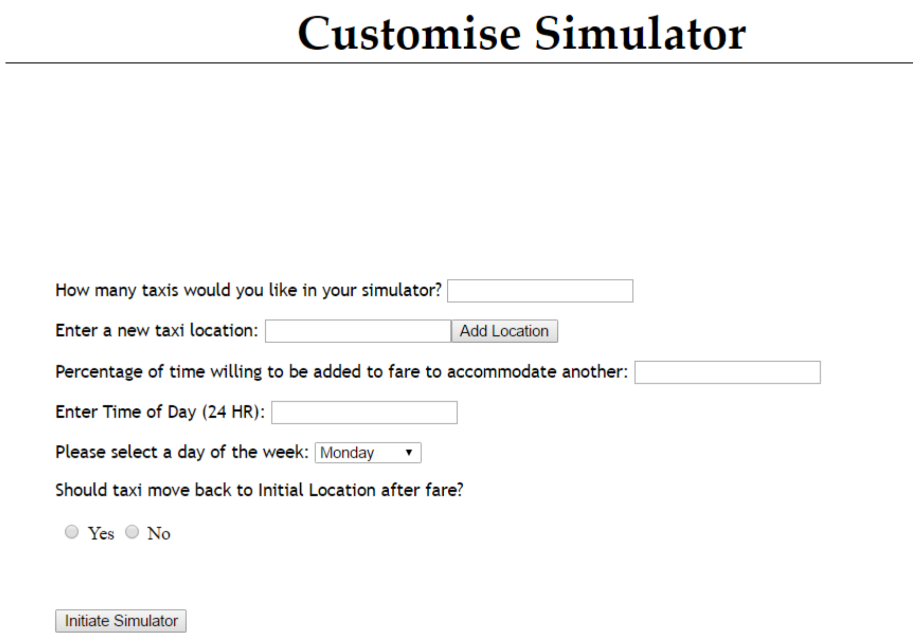
Many other simulators implied a grid function when searching for taxis in order to search through them faster. However this wasn’t an option unless there was a standard environment to run in which was not the goal of this project. I investigated the creation of a dynamic grid that could use the nearest current taxi as a benchmark for searching for taxis. Unfortunately this was not done to a sufficient standard to include in this project, but hopefully a version could be created that could rule out taxis that will not have a chance of being picked as the best taxi. The benefit of this would be faster response times and allow the system to cater for more demand.

Catering for more demand is a high level goal for this project in the future. The ability for this product to cope with large amounts of actual taxi data and then to be able to produce the same realistic results would allow for this to be a very useful system. Automated fares could be randomised to pick either random origin or destination points in a realistic manner would enhance the system tenfold. The improvements could lead to such real world benefits as reduced traffic congestion thus reduced Carbon Dioxide Emissions.

8.3 Conclusion Summary

I thoroughly enjoyed working on this project. It involved developing and learning a wide variety of skills, whilst encountering a new problem domain. It was a relatable idea which I could see in use in everyday life. I have learned much from completing this project and I believe it will stand to me in my coming endeavours.

Appendix A



Customise Simulator

How many taxis would you like in your simulator?

Enter a new taxi location: Add Location

Percentage of time willing to be added to fare to accommodate another:

Enter Time of Day (24 HR):

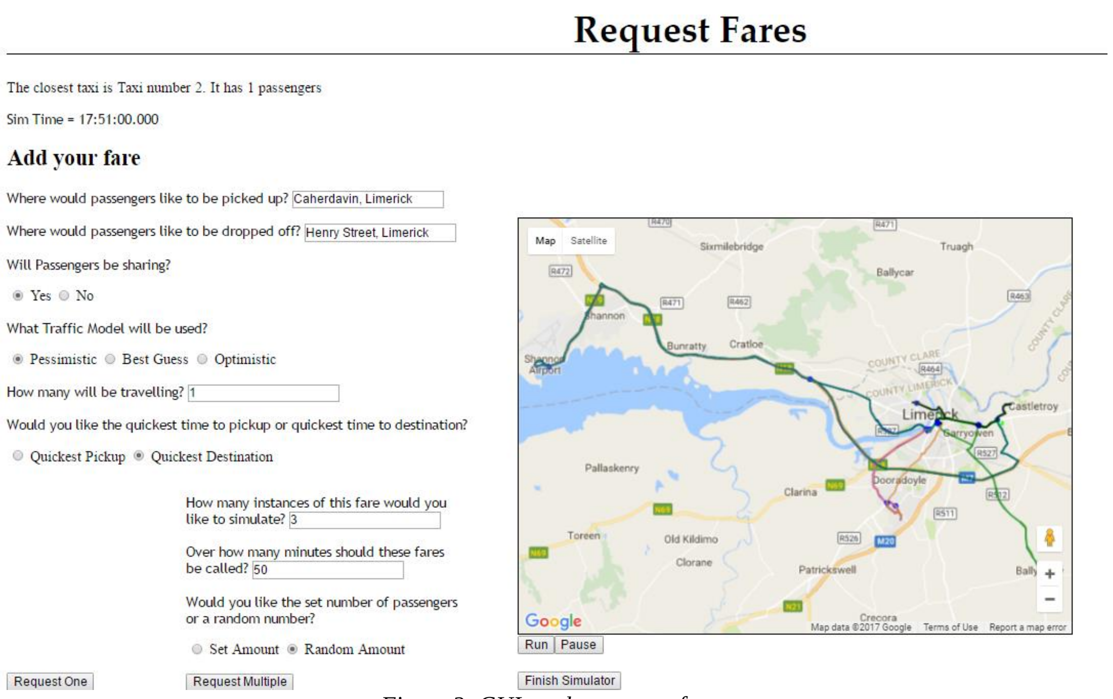
Please select a day of the week: Monday

Should taxi move back to Initial Location after fare?

Yes No

Initiate Simulator

Figure 1: GUI used to specify Simulator Environment



Request Fares

The closest taxi is Taxi number 2. It has 1 passengers

Sim Time = 17:51:00.000

Add your fare

Where would passengers like to be picked up? Caherdavin, Limerick

Where would passengers like to be dropped off? Henry Street, Limerick

Will Passengers be sharing?

[✓ Yes] No

What Traffic Model will be used?

[✓ Pessimistic] Best Guess Optimistic

How many will be travelling? 1

Would you like the quickest time to pickup or quickest time to destination?

Quickest Pickup [✓ Quickest Destination]

How many instances of this fare would you like to simulate? 3

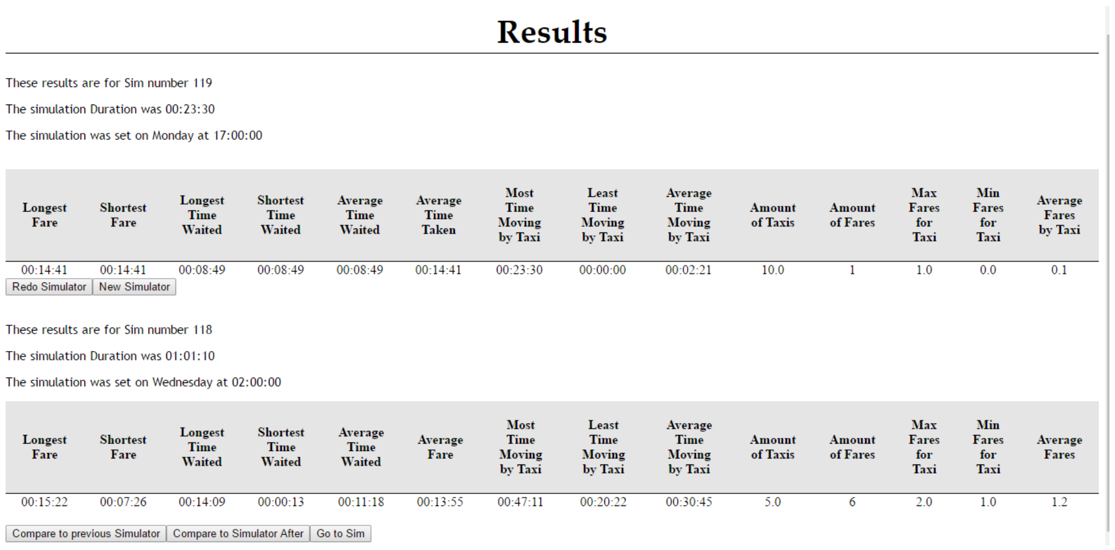
Over how many minutes should these fares be called? 50

Would you like the set number of passengers or a random number?

Set Amount [✓ Random Amount]

Request One Request Multiple

Figure 2: GUI used to request fares



Results

These results are for Sim number 119

The simulation Duration was 00:23:30

The simulation was set on Monday at 17:00:00

Longest Fare	Shortest Fare	Longest Time Waited	Shortest Time Waited	Average Time Waited	Average Time Taken	Most Time Moving by Taxi	Least Time Moving by Taxi	Average Time Moving by Taxi	Amount of Taxis	Amount of Fares	Max Fares for Taxi	Min Fares for Taxi	Average Fares by Taxi
00:14:41	00:14:41	00:08:49	00:08:49	00:08:49	00:14:41	00:23:30	00:00:00	00:02:21	10.0	1	1.0	0.0	0.1

These results are for Sim number 118

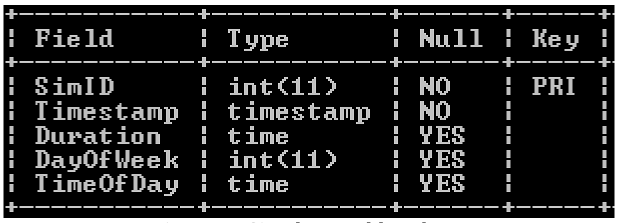
The simulation Duration was 01:01:10

The simulation was set on Wednesday at 02:00:00

Longest Fare	Shortest Fare	Longest Time Waited	Shortest Time Waited	Average Time Waited	Average Fare	Most Time Moving by Taxi	Least Time Moving by Taxi	Average Time Moving by Taxi	Amount of Taxis	Amount of Fares	Max Fares for Taxi	Min Fares for Taxi	Average Fares
00:15:22	00:07:26	00:14:09	00:00:13	00:11:18	00:13:55	00:47:11	00:20:22	00:30:45	5.0	6	2.0	1.0	1.2

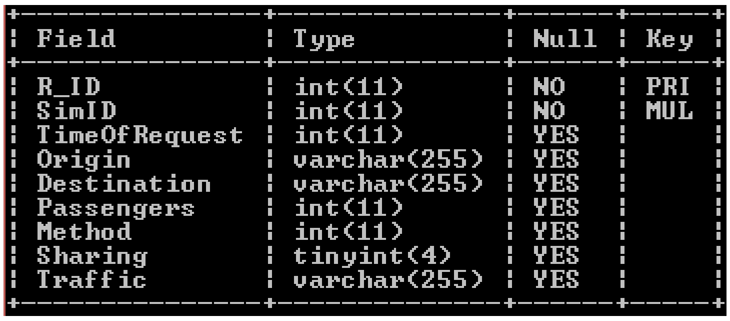
Figure 3: Webpage used to display results of simulator

Appendix B



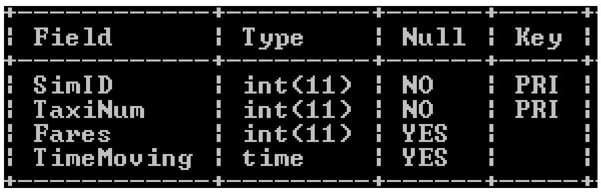
```
+------------+-----------+------+-----+
¦ Field      ¦ Type      ¦ Null ¦ Key ¦
+------------+-----------+------+-----+
¦ SimID      ¦ int(11)   ¦ NO   ¦ PRI ¦
¦ Timestamp  ¦ timestamp ¦ NO   ¦     ¦
¦ Duration   ¦ time      ¦ YES  ¦     ¦
¦ DayOfWeek  ¦ int(11)   ¦ YES  ¦     ¦
¦ TimeOfDay  ¦ time      ¦ YES  ¦     ¦
+------------+-----------+------+-----+
```

Figure 4: Simulator table schema



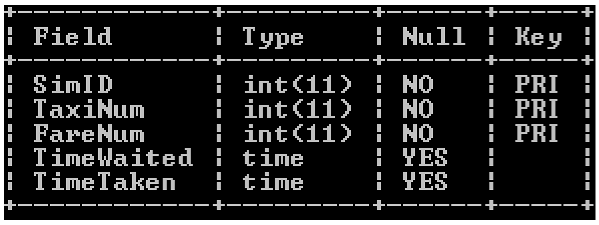
```
+---------------+--------------+------+-----+
¦ Field         ¦ Type         ¦ Null ¦ Key ¦
+---------------+--------------+------+-----+
¦ R_ID          ¦ int(11)      ¦ NO   ¦ PRI ¦
¦ SimID         ¦ int(11)      ¦ NO   ¦ MUL ¦
¦ TimeOfRequest ¦ int(11)      ¦ YES  ¦     ¦
¦ Origin        ¦ varchar(255) ¦ YES  ¦     ¦
¦ Destination   ¦ varchar(255) ¦ YES  ¦     ¦
¦ Passengers    ¦ int(11)      ¦ YES  ¦     ¦
¦ Method        ¦ int(11)      ¦ YES  ¦     ¦
¦ Sharing       ¦ tinyint(4)   ¦ YES  ¦     ¦
¦ Traffic       ¦ varchar(255) ¦ YES  ¦     ¦
+---------------+--------------+------+-----+
```

Figure 5: Requests Table Schema



```
+------------+---------+------+-----+
¦ Field      ¦ Type    ¦ Null ¦ Key ¦
+------------+---------+------+-----+
¦ SimID      ¦ int(11) ¦ NO   ¦ PRI ¦
¦ TaxiNum    ¦ int(11) ¦ NO   ¦ PRI ¦
¦ Fares      ¦ int(11) ¦ YES  ¦     ¦
¦ TimeMoving ¦ time    ¦ YES  ¦     ¦
+------------+---------+------+-----+
```

Figure 6: TaxiStats Table Schema:



```
+------------+---------+------+-----+
¦ Field      ¦ Type    ¦ Null ¦ Key ¦
+------------+---------+------+-----+
¦ SimID      ¦ int(11) ¦ NO   ¦ PRI ¦
¦ TaxiNum    ¦ int(11) ¦ NO   ¦ PRI ¦
¦ FareNum    ¦ int(11) ¦ NO   ¦ PRI ¦
¦ TimeWaited ¦ time    ¦ YES  ¦     ¦
¦ TimeTaken  ¦ time    ¦ YES  ¦     ¦
+------------+---------+------+-----+
```

Figure 7: FareStats Table Schema

References

Guindon, C. (2017). *Eclipse desktop & web IDEs.* [online] Eclipse.org. Available at: http://www.eclipse.org/ide/ [Accessed 26 Mar. 2017].

Globalwebindex.net. (2013). *Top global smartphone apps, who's in the top 10.* [online] Available at: https://www.globalwebindex.net/blog/top-global-smartphone-apps [Accessed 4 Jan. 2017].

Google Developers. (2016). *Google Maps Web Service APIs | Google Developers*. [online] Available at: https://developers.google.com/maps/web-services/ [Accessed 4 Jan. 2017].

Ibm.com. (2017). *JSF MVC Pattern*. [image] Available at: https://www.ibm.com/support/knowledgecenter/SSRTLW_9.6.0/com.ibm.etools.jsf.doc/topics/cmvc.html [Accessed 26 Mar. 2017].

iCabbi (2017). *Example Dispatch System and Analytics*. [image] Available at: http://www.icabbi.com/ [Accessed 4 Jan. 2017].

Jung, J., Jayakrishnan, R. and Choi, K. (2015). A Dually Sustainable Urban Mobility Option: Shared-Taxi Operations With Electric Vehicles. *International Journal of Sustainable Transportation*, p.150930131156001.

Jung, J, Jayakrishnan, R. and Park, J.Y. (2013). Design and Modeling of Real-time Shared-Taxi Dispatch Algorithms. Proceedings of the Transportation Research Board's 92th Annual Meeting, Washington, D.C.

Kahiga, S. (2014) *Rideshare for UL Staff – Website Development*, unpublished thesis, (B. Sc.), University of Limerick.

Kuo, M. (2016). *Taxi Dispatch Algorithms: Why Route Optimization Reigns*. [online] Routific. Available at: https://blog.routific.com/taxi-dispatch-algorithms-why-route-optimization-reigns-261cc428699f [Accessed 26 Mar. 2017].

Maven.apache.org. (2016). *Maven – Introduction*. [online] Available at: https://maven.apache.org/what-is-maven.html [Accessed 4 Jan. 2017].

O'Sullivan, K. (2013) *Refuelling Optimizer*, unpublished thesis, (B. Sc.), University of Limerick.

Ota, M., Vo, H., Silva, C. and Freire, J. (2016). STaRS: Simulating Taxi Ride Sharing at Scale. *IEEE Transactions on Big Data*, pp.1-1.

Rosettacode.org. (2017). *Haversine formula - Rosetta Code*. [online] Available at: https://rosettacode.org/wiki/Haversine_formula [Accessed 10 Apr. 2017].

TutorialsPoint, (2017). *Agile Development Lifecycle*. [image] Available at: https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm [Accessed 26 Mar. 2017].

Schalk, C. (2005). *Introduction to Javaserver Faces - What is JSF?*. [online] Oracle.com. Available at: http://www.oracle.com/technetwork/topics/index-090910.html [Accessed 26 Mar. 2017].

Stack Overflow. (2016). *Stack Overflow Developer Survey 2016 Results*. [online] Available at: http://stackoverflow.com/research/developer-survey-2016 [Accessed 4 Jan. 2017].

Uber.com. (2017). *Ride with Uber – Tap the Uber App, Get Picked Up in Minutes*. [online] Available at: https://www.uber.com/en-IE/ride/ [Accessed 26 Mar. 2017].

Uber Global. (2014). *Optimizing a dispatch system using an AI simulation framework*. [online] Available at: https://newsroom.uber.com/semi-automated-science-using-an-ai-simulation-framework/ [Accessed 4 Jan. 2017].

Zaleski, M. and Tartar, M. (2016). *Uber Is Now the Most Popular Taxi App in 108 Countries*. [online] Bloomberg.com. Available at: https://www.bloomberg.com/news/articles/2016-08-23/uber-is-the-most-popular-ride-hailing-app-in-108-countries [Accessed 4 Jan. 2017].