Table of Contents

[1. Introduction - 3 -](#_Toc481172358)

[1.1 Problem drivers - 3 -](#_Toc481172359)

[1.1.1 Problem to be solved - 3 -](#_Toc481172360)

[1.1.2 Project Purpose - 3 -](#_Toc481172361)

[1.1.3 Project Objectives - 4 -](#_Toc481172362)

[1.1.4 Limiting the Project Scope - 4 -](#_Toc481172363)

[1.1.5 Outline of plan - 4 -](#_Toc481172364)

[1.2 Output Summary - 5 -](#_Toc481172365)

[1.2.1 Requirement Specification Document - 5 -](#_Toc481172366)

[1.2.2 Design Specification Document - 5 -](#_Toc481172367)

[1.2.3 Visualisation platform - 5 -](#_Toc481172368)

[1.2.4 Testing Documents - 6 -](#_Toc481172369)

[2. Literature Review - 7 -](#_Toc481172370)

[2.1 Initial Development - 7 -](#_Toc481172371)

[2.2 Java Data Structures - 7 -](#_Toc481172372)

[2.3 JavaScript - 7 -](#_Toc481172373)

[2.4 D3.js - 8 -](#_Toc481172374)

[3. Methodology - 8 -](#_Toc481172375)

[3.1 Software Development Model - 8 -](#_Toc481172376)

[3.2 Initial Planning - 9 -](#_Toc481172377)

[3.3 Information gathering and Requirement analysis - 9 -](#_Toc481172378)

[3.3.1 Interview with the client - 10 -](#_Toc481172379)

[3.3.2 Requirement specification - 10 -](#_Toc481172380)

[4. Implementation - 11 -](#_Toc481172381)

[4.1 Iteration 1 - 11 -](#_Toc481172382)

[4.1.1 Design Visualisation - 11 -](#_Toc481172383)

[4.1.2 Implementing the code - 12 -](#_Toc481172384)

[4.1.3 Testing - 12 -](#_Toc481172385)

[4.1.4 Stakeholder feedback - 12 -](#_Toc481172386)

[4.2 Iteration 2 - 13 -](#_Toc481172387)

[4.2.1 Design Visualisation - 13 -](#_Toc481172388)

[4.2.2 Improve business logic - 13 -](#_Toc481172389)

[4.2.3 Testing - 13 -](#_Toc481172390)

[4.2.4 Stakeholder feedback - 13 -](#_Toc481172391)

[4.3 Iteration 3 - 13 -](#_Toc481172392)

[4.3.1 Design Visualisation - 13 -](#_Toc481172393)

[4.3.2 Improve business logic - 13 -](#_Toc481172394)

[4.3.3 Testing - 13 -](#_Toc481172395)

[4.3 .4 Stakeholder feedback - 13 -](#_Toc481172396)

[5. Results - 13 -](#_Toc481172397)

[5.1 Iterations Results - 14 -](#_Toc481172398)

[5.1.1 Iteration 1 - 14 -](#_Toc481172399)

[5.1.2 Iteration 2 - 17 -](#_Toc481172400)

[5.1.3 Iteration 3 - 17 -](#_Toc481172401)

[5.2 Alternative Results - 17 -](#_Toc481172402)

[5.2 Complexity - 17 -](#_Toc481172403)

[5.3 Testing - 17 -](#_Toc481172404)

[5.4 Evaluation - 17 -](#_Toc481172405)

[6. Conclusion - 18 -](#_Toc481172406)

[APPENIDX A - 18 -](#_Toc481172407)

[APPENIDX B - 19 -](#_Toc481172408)

# 1. Introduction

## 1.1 Problem drivers

Organizations using Business Intelligence, use raw data to extract significant information to make decisions, spot business problems and boost their performance. The problem arises when these information are inaccurate and they lack quality, leading them to decrease efficiency in business and production. For such cases, a model should be carefully designed using validation techniques to ensure that the output will meet the appropriate criteria for each case study. Firstly, developers ensure that the conceptual model is valid by gathering information from the stakeholders and address the main problem. Then they build a computerised model to implement a solution for the problem. There might be cases when the model’s output behaviour is accurate but there will be times when the model is tested with different input it will produce inaccurate results. This causes confusion and requires the developer’s team to test their code again and again until they spot the main problem.

### 1.1.1 Problem to be solved

Knowledge engineering group (KEG) is a team of researchers, employed by the University of Brighton to work on large projects worldwide. The group consists a team of developers who were assigned a project to develop business intelligence platform for the Southeastern Railway. This project requires the use of raw data such as train timetables, stops and destinations. KEG’s analyst realised the rail company had many issues with their historic data as the trains had many delays or technical problems with their train fleet. A part of this project was to create a route finder that will be hosted to the rail’s company website. The developers using Java, they have already build a model that computerised the routes using the data provided by the rail company. Unfortunately, their code was not efficient and it produced inaccurate outputs. What irritated them the most was the time they needed to spend debugging the code to identify the problem. Their need was to find a different and most efficient way to understand and identify their problem as the deadline for their project was pressing them.

### 1.1.2 Project Purpose

The purpose for this project was to create a simulation model that illustrates clearly the whole implementation of their code and visualise the process how data are manipulated and extracted to their final form. By breaking down the code into different modules and visualising them in a different environment will give a clear understanding to the developers how their code executes using different data sets. The whole model should be re-programmed using JavaScript and D3.js. The code should be translated carefully to serve the main functionality as the original one. The main target for this project is to not identify and create a better version of the previous model but to develop a new environment that will visualise the process and to be tested using the many data sets provided by the company. This will give a clear insight to the group’s developers to identify potential mistakes caused on the operational process or even more lack of data quality. The new model will use these data for the input and it will produce a new output that it will be compared and tested with the correct results provided by the KEG’s staff.

### 1.1.3 Project Objectives

Objectives targeted as follows:

* Create an interactive and fully functional visualisation environment that receives a csv file and produces a new data set with the correct outputs
* Fully understand and translate Java code and data structures to JavaScript
* Understand the requirements and protocols provided by the KEG’s developers
* Visualise each component of code
* Test every module separately
* Test the system thoroughly and diminish any bugs

• Code the system in such a way so near future changes can be done effortlessly

### 1.1.4 Limiting the Project Scope

The scope of this project was to develop a visualisation environment to add data and illustrate the process until the expected output is extracted. In order to decide upon the functionality, the author researched extensively the available visualisation frameworks to find the best course of action. Using the data the rail company provided along with the research done and the frequent meetings with the developer’s team, enable the author to construct a plan using an agile software engineering model. Moreover, by examining the Java code, the author was able to decide and choose more technical stuff like the data structures required to enable the same functionality of the original code, combined with the visualisation components.

### 1.1.5 Outline of plan

**Research:** The first course of action was to collect as much information as possible from the KEG developers to identify their main issue and understand their code. Then the author researched available visualisation frameworks and decided the most suitable for this project.

**Design:** After collecting all the necessary information and deciding upon an agile model, a plan was created to design the basic model of visualisation which included the flow chart of the code transformed into modules. Moreover, functional requirements were created for the system and discussed with the director of Knowledge engineering group until an agreement was reached. JavaFx was used for the creation of the visualisation environment but was replaced with D3.js later when the team decided to host the platform on a web browser.

**Implementation**: The implementation started as soon as the team decided to use JavaScript libraries. The first stages of the implementation were to convert the Java code to JavaScript and use queries to store the output to MySQL. Later, all csv files were converted to JSON format to ease the process and avoid relational database systems.

**Testing:** The testing phase consists of two different parts. Firstly, each node on the visualisation platform must be checked it produces the same result as the original code. After making sure the operational process was functioning properly, train data were used to produce the routes for each train. Then it was compared with the KEG’s output to ensure both workflows were producing the same results. After all checks and tests are completed, KEG will be able to use the simulation model to identify mistakes and improve their code.

## 1.2 Output Summary

In this section of the project, the author will be talking about the major documents produced during the project.

### 1.2.1 Requirement Specification Document

Recipient: Knowledge Engineering group, author

File type : Microsoft Word Document

The first document that was created was the requirement specification document which was crucial in the development of the system since it was created to record and understand the client needs. Moreover, it can be used as a reference point for what the system needs to achieve.

This document contains the requirements specification document with a list of all the functional and non-functional requirements for the system approved by the director of KEG. These were developed after an extensive information gathering and analysis which included methods like interviewing the group members, using a questionnaire to identify their basic needs and understand the process taken to develop the original code for the route finder part of the system.

All the documents created can be found in **APPENDIX A**

### 1.2.2 Design Specification Document

Recipient: Knowledge Engineering group, author

File type : Microsoft Word Document

The second document created was to help even further with the development of the project and again act as a reference for what the system should look like.

This document contains designs of the visualisation as well as flow-charts diagrams that were used to structure the functionality of the code. These were derived from the information gathered by the author and the Java code of the route finder.

all the documents, drafts, diagrams created can be found in **APPENDIX B**.

### 1.2.3 Visualisation platform

Recipient: Knowledge Engineering group

File type : .rar file

This is the finished product that will be used from the group as their visualisation environment to check their code efficiency and execution process in each step, filtered based on each row on the data set and each route created. This folder contains a repository with the raw data, a repository with the original code and the new code written in JavaScript and a repository containing all the files needed for the visualisation including the data in JSON format.

Some of the outputs create are shown on **APPENDIX C**

### 1.2.4 Testing Documents

# 2. Literature Review

This section will detail what books, articles and resources the author had to read in order to be able to undertake this project. This section has been broken up to into four main categories which include Initial Development, Java data structures, JavaScript and d3.js.

## 2.1 Initial Development

Project planning and control was one of the modules the author attended during his university studies, therefore after researching development models online and reading books the author decided to work on Agile model. To do so the user reviewed all the module materials to find more information about development models and their advantages. After reading the book “Software Engineering” (Sommerville, 2010) the author found three development models that all suited his needs and were straightforward to understand. The Waterfall model which was eventually rejected, is the more standard development model used were the developers need to split the project into separate phases considering the fundamental process activities of specification, development, validation and evolution (Sommerville, 2010). An alternative option would have been the increment model which is based on the idea of developing an initial implementation, exposing this user comment and evolving it through several versions until an adequate system has been developed (Sommervile, 2010). This was then considered both by the author and the client due to the fact that constant communication was feasible as both were living in Brighton and they could plan regular meetings. At the end both agreed to use Agile model as the requirements of this project might change over the time. In contradiction with the waterfall model there was no need to have a strict plan as the development started early and by the regular meeting there was a feedback through the whole process. The quality of the software was improved using the extreme programming methodology by delivering a small version of the final product each week or month to the client and discussing changes or improvements.

## 2.2 Java Data Structures

During the research and design process the author spent a lot of time interviewing KEG’s developers to understand their code. The data structures that were used for the Java code were Lists and ListIterator. Again, the author had previous experience with data structures form his 2nd year module called Data Structures and Algorithms. To refresh his memory and make the most sensible decision the author spend time reading the Oracle’s documentation about the ListIterator interface and its methods. All the data in the Java code were manipulated using the methods from that interface, such as hasNext(), hasPrevious(), next() and previous(). After understanding these methods, the author tested the code by printing the results on the console to understand clearly their functionality.

## 2.3 JavaScript

The decision to choose d3.js to implement the visualisation for this project led the author to choose JavaScript as the programming language for translating the Java code. D3 is a library written in JavaScript so it will be much easier to combine the logic of the code with the visualisation environment. Because Java is different from JavaScript, there was a risk that the functionality of the system could not meet the requirements. To ensure this problem would not exists, the author researched ways to implement a class in JavaScript having the same functionality as the ListIterator interface. By reading posts in StackOverflow, there were many developers that have recreated this interface using plain JavaScript. After the long discussion the author had with the KEG’s director, they decided to not use this method, instead the author should create his own way to manipulate data in the list. By reading articles on StackOverflow on how to manipulate data inside lists or arrays the author concluded to use a simple for loop and create variables that will decide which objects in the list will be considered as current, previous and next.

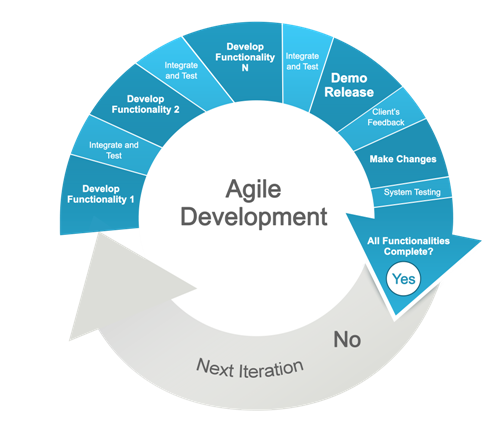
## 2.4 D3.js

D3 is a small library for great visualisations and its widely discussed as one of the best techniques to visualise almost anything. Like programming, d3 needs a lot of practise to understand its methods and get familiarised with it. The author during his research process started spending time studying Mike’s Bostock website which is the author of d3 - Available at : <https://bost.ocks.org/mike/> (Accessed: 27/10/2016). After understanding the fundamentals of this library, the author researched the official website of d3 Available at: <https://d3js.org> (Accessed: 10/11/2016) which provided an amazing documentation for all types of visualisations and their code. By using the code provided by these websites and d3’s GitHub profile, the author experimented with many types of visualisations and decided to use the template provided by Mike Bostock on collapsible trees. By using this tree, the code could be easily illustrated by breaking it down into modules and visualise them as tree nodes.

# 3. Methodology

This chapter will describe the methods used and work done by the author over the course of the project to achieve its purpose.

## 3.1 Software Development Model

The software development model that was used by the author is the Agile model. It is a model that satisfies the stakeholders through early and continuous delivery of valuable software. It welcomes changing requirements, even late in development. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. (Mike Beedle, 2002). Flexible panning, fast implementation and early product delivery is what defines scrum as agile, repeated in n number of iterations. For example, the author decided to split this project into 4 iterations which each consist of 6 phases: Requirement analysis, Planning, Design, implementation, Testing and stakeholder’s review. This model was ideal for this project as the main problem of the stakeholders was to visualise their code and the end-goal was not clearly defined. The requirements where changing during this process as the members of KEG could decide a new or modify existing features on the visualisation environment. The great communication between the author and the team was ideal for this model as both were living in the same city. One can ask “why not use the increment model then?”. The reason is that agile is both iterative and incremental. There is no need to follow any strict protocol or develop the features at first and then add functionalities to them in every iteration.

Due to the fact that this was the first full scale individual project the author had to tackle, there was a need for a more “agile” methodology. In principle, the Agile model states that the focus is to develop the full functionality of the product from the beginning and have continues feedback by frequently releasing demos of the product. The review from the stakeholders in each iteration was a critical part for the development of the end-product, as an undergraduate was supported with the appropriate guidance from professionals.

This principle was followed up to a point, however due to some particular factors like inexperience and changes in the requirements the project had to be adjusted. Therefore, in practice the iterations of the project, overlapped up to a point to provide the author with appropriate feedback, for example during implementation, problems are found and corrected or during the design phase some more requirements are identified and need to be included.

## 3.2 Initial Planning

Time is of the essence in a software development project so the first task the author did in preparation for the project was to develop a work plan. For a better representation of the time spend for each phase of the project, the author created a Gant Chart to allocate time for Requirements Analysis and Research and all iterations of the project. Furthermore, the plan included time taken off project during the examination period in January. However, the initial planning was not well thought and due to some other factors like the stakeholder’s slow response to provide the necessary data and miscalculating the time needed for translating the business logic to JavaScript, the initial work plan had to be adjusted. Some key changes included the re-allocation of every single End Date of the iteration. The first iteration was planned to start from the first week of December but the Java code and data were given in January so the new plan consisted of 3 iterations which they lasted 1 month each. The author spent that time to practice his visualisation code. Both the initial Gant Chart and the changes can be seen in **APPENDIX A, Project Definition Report**

## 3.3 Information gathering and Requirement analysis

This section will focus on the work done by the author to collect the necessary information for the development of the system requirements. The author had a respectable amount of knowledge with object oriented languages, therefore the stages involving communication with the client were more focused on specific requirements the group had about the Java code they provided to the author, their interaction with the current route finder application as well as their needs to visualise the whole process. All the original documents between the interaction between the author and the client can be seen in **APPENDIX A**

### 3.3.1 Interview with the client

In order to gather more information about the current application and general requirements, the author had an interview with the director of Knowledge engineering group as well with the developers. The director expressed his need to create a basic visualisation platform that visualises the operational process of their code that creates the routes for the trains. Their main requirement was to represent the exact rules as a filter or an SQL command. He mentioned that their existing code is not accurate and it produces wrong outputs. Instead of rewriting a better version of the existing logic, it was necessary to transform this logic into a visualisation environment that will be functioning according to the business protocols according to the client’s case study. The client did not provide enough information about the exact end-product as the features will be discussed and changed between the iterations, emphasizing that the main priority was to visualise the process. The author was responsible to design a new template that will be used as a simulation model of the current system.

After understanding the main requirements of the project, the author and developers had a long interview discussing the functionalities of the current route finder application. The first task was to understand the data and the expected output. The raw data was many csv files containing information about the train unit numbers and where they stop and start. In addition, the data contained information unnecessary for the routes. For example, some trains might have technical issues, or they must stop to depot destinations for repairs. The route finder application was written in Java and its main functionality was to remove unnecessary rows from the data or add a new column, pointing which rows were the first or last destination of a specific route.

The developers provided the author with 5 files, the java code, 2 csv files containing the raw data of 2 unit numbers and 2 csv files containing their correct output. The first file was the Java code which stored the data into a ListIterator and then it was manipulated inside a long while loop containing many else if statements with the business logic. The csv files with the correct output were given to the author to allow him to test the new system.

### 3.3.2 Requirement specification

The Requirements Specification was a document that was developed and updated frequently from the beginning of the first phase of the project. The requirements were separated into two sections; functional and non-functional. Functional requirements being statements of services the system should provide, how the system reacts to particular inputs, and how the system should behave in particular situations (Sommerville, 2010) and non-functional requirements being constraints on the services or functions offered by the system (Sommerville, 2010). A general description of the requirement is included along with date the requirement was written as well as the Priority of the requirement for the system divided into three categories (High, Medium, Low). The full document with both functional and non-functional requirements can be found in **APPENDIX A**

# 4. Implementation

## 4.1 Iteration 1

After the author interviewed the developers and identified the key requirements of the project, this iteration was the most crucial as the first demo was expected at the end of the cycle. Four files were created for this part of the project.

* demo.js
* logic.js
* index.html
* style.css

Furthermore, all data received form the stakeholders were converted to JSON format to be accessible by JavaScript and d3.

### 4.1.1 Design Visualisation

|  |
| --- |
| Figure 2: Draft design (Author) |

The first draft was designed on paper displaying shapes that represent methods or filters. (Figure 2, Author). Because the operational procedure is sequential, each shape will contain a unique functionality and some of them will have dependencies. For example, the first shape of the visualisation is a square. This square represents the method that reads data and makes the appropriate checks if the data are valid and doesn’t contain any useless characters. The circles could represent some filters that will receive data from the square as a parameter and filter them. This model will illustrate different shapes as different kind of operations. The users will be able to click at any shape and see the actual data that are manipulated at each phase.

Although this model does not meet all the requirements and doesn’t fit with the business logic, the author wanted to design the first demo to combine the logic and visualisation as the first prototype of something bigger. If the end-product was objectified, the author would characterise it as a large engine with different types of gears. The iterations’ purpose was to build a small gear model that will be used afterwards to create similar gears and combine them all together, until the engine is completed.

Section 5.1.1 discusses the actual design that was delivered to the client

### 4.1.2 Implementing the code

The implementation for this iteration contained 2 different stages. The first is the translation of the java file to JavaScript and the second is the visualisation process combined with the logic.

Starting with the translation, the author spent a lot of time reading the Java code to understand it’s structure. Firstly, KEG’s developers decided to store the data into a ListIterator of Strings, so each row of the dataset could be treated as a long string. All the columns were defined as variables on the global scope and with the help of an external class called FastTokenizer and the use of commas, the actual columns were extracted from each row. By using a while statement and the method hasNext(), the team was able to loop through the data and use the business logic as if statements to remove lines or to add a new column to assign the values of first or last. The challenge for the author was to recreate the same process described above into a similar structure in JavaScript to serve both the same functionality and support the visualisation environment of the new platform. After resolving the problem with the data structures for the business logic, the author broke down the Java code into modules and started rewriting the code to the logic.js file.

The full document with the structure of Java code can be found in **APPENDIX B**

Section 5.1.1 discusses the solution for this problem

For the visualisation, the author tried to visualise 2 procedures from the code, reading the data and filtering the data according to their destinations. The first shape will be a square and it will be responsible for reading the data. The second shape is responsible to filter the data and remove all rows containing the word Depot. When the user click the shape, a table will be displayed showing the actual data.

### 4.1.3 Testing

To ensure that the first demo was functioning properly, test data were created by the author to test the procedure. The first demo was launched and the user was able to click on the shapes and display the appropriate data on a table attached on the DOM interface. By using the Firbug extension on Firefox, the author could print results to the console to examine the data before and after the filtering procedure. Moreover, all the operations in the logic.js were commented with a given name by the author; the console was used to test each module of the code by printing the results and the name of each operation to ensure they are returning correct outputs.

### 4.1.4 Stakeholder feedback

Due to the time needed to rewrite the code to JavaScript, the demo did not support many features for the visualisation part. The stakeholders were satisfied with the prototype but they were expecting more features. In addition, there was no need to use different shape for each operation and at the end of the next iteration it was expected to visualise the whole process.

## 4.2 Iteration 2

After the meeting with the stakeholders, the author had to make a fast plan to meet the new requirements. While researching d3.js, the author experimented with many visualisation examples and the most suitable for this case was the collapsible tree. All modules from the code could be represented by a simple circle. All dependencies could be illustrated by creating children to the appropriate nodes. Moreover, the business logic now must be changed to functioning with the visualisation environment.

### 4.2.1 Design Visualisation

|  |
| --- |
| Figure 3: Daft design 2 (Author) |

By using the Java code structure in APPENDIX B, the author was able to rewrite the same structure into a JSON file. This file was used as the data for the collapsible tree. The end nodes of the tree will represent the business protocols which are remove line and add the value of FIRST or LAST to a new column. All nodes’ names will be the given names the author provided into the code using comments. The tree will start with all its nodes collapsed. The main idea is to develop a functionality to expand specific nodes while the code runs and display the results of the new and original data file in 2 different tables as can be seen in the figure 3. The html file should be spitted in 2 parts. The left part is responsible to host the d3 tree and display the necessary variables, like the current ***i*** and the second part is to display the results as tables.

### 4.2.2 Improve business logic

In order to achieve the functionality mentioned above, the code should be separated into different files containing the necessary functionality to achieve modularity and improve the testing phase. After designing the second demo for this iteration the author decided to break down the code into 3 new files:

|  |
| --- |
| Figure 1: treeData.json example (Author) |

* tree.js
* table.js
* logic.js

**tree.js**

After splitting the html file into different parts, the author started developing the collapsible tree and experimenting with the DOM. The first step was to create the new data that will be used to generate the tree. As mentioned before, by using the code structure that was created from the RoutesFinder application, the author created a new JSON file called treeData.json. The basic attributes of the data were the name, parent and children. Each object in the file will be represented as a tree node in the collapsible tree. The code will check if each node has children and it will link them together as children nodes using a diagonal in d3.js. During the implementation, the author decided to use a single variable to define the data as it will improve the performance of the application. To read the data from an external file, it is costly and decreases the complexity of the system as it must call the d3 class to find the data file and extract them to an array of objects. Instead by defining a new variable in the global scope, data can be accessed directly in any part of the application.

|  |
| --- |
| Figure 2: tree.js |

# Following the example of the Collapsible tree code (Mike Bostock, 2010) the author had to add some new functions and modifications to fit and combine the business logic with the visualisation. Firstly, some variables had to be defined on the top level like the dimension of the tree, the tree element from d3, the diagonal that will draw curved lines connecting the nodes and the svg element that receives the container of the DOM to place the new tree. All the elements of the tree (nodes and links) will be grouped together using the ***“g”*** attribute.Whenever the user interacts with tree, or the logic wants to modify the structure of the tree, all elements must be updated. For that reason, a new function was defined with the name update which receives the data of the tree and calculates the new position for each node.

|  |
| --- |
| Figure 3 tree.js (Author) |

# The main behaviour of the tree is to append and display all the circles and text as nodes on the DOM. In addition, the tree must collapse or expand specific nodes. To achieve this behaviour the update function should create specific variables for each process such as, nodeEnter, nodeExit and nodeUpdate. Starting by defining the nodes and links, the function will automatically calculate how many elements there are in the tree data and attach them on the tree as a node or a link. NodeEnter will be responsible to append a circle and its associated text in the “***g”*** attribute to display the nodes when the tree is created. NodeExit will be responsible to remove each element from the tree and nodeUpdate will transition the nodes whenever the tree is updated.

Every time the index.html is launched for the first time, the tree will automatically collapse all if its nodes. To achieve that, 2 functions should be created out of the update scope. These methods will receive a parameter as the node that the system wants to collapse and it will check if there are any children. For the collapse function, if the previous statement is true, the children will be stored to a temporary variable and then will be set to null. The same process will be repeated for any child that contains its own children. For the expand method the same process occurs but in the reverse way with the only difference, it must check where the children notes are being stored. These methods work because every time they are called they are changing the content of our data, so when the tree is updated it will not display the associated children nodes as the children variable was set to null. The expand method will be discussed later on this section.

# 

### 4.2.3 Testing

### 4.2.4 Stakeholder feedback

## 4.3 Iteration 3

### 4.3.1 Design Visualisation

### 4.3.2 Improve business logic

### 4.3.3 Testing

### 4.3 .4 Stakeholder feedback

# 5. Results

This section will cover in detail the results of this project, the outputs produced by the use of the methods from section 4. The author will then talk about the importance of each output produced and how significant these results were for the next phases of the project. It will follow a similar structure to section 4 stating the outputs and discussing the result in chronological order like the methods were presented before.

## 5.1 Iterations Results

### 5.1.1 Iteration 1

|  |
| --- |
| Figure 4: logic.js (Author) |

**Java to JavaScript**

By converting the data into a JSON format it was easier to manipulate them. Every row in the dataset was represented by an object. The columns were the attributes of that object and they could be accessed by simply using the key as the number of the row and the value as the number of the column. For that reason, the author decided to use a simple for loop to represent the ListIterator and replace the according line in the java code with the while(hasNext()). After the system read and checked the data, they were stored to a variable called ***ds*** (alias for dataset). ListIterator interface use pointers to jump between elements. Methods like next() or previous() return the element next to or before the current element. By defining 3 variables in the local scope called curr, prev and next, the next() and previous() methods of the interface will be replaced by these simple variables that will hold the value of ***i***, depending on their position. Moreover, methods such as hasNext() or hasPrevious() will be eliminated as prev takes the value of null when ***i*** equals 0 and next takes the value of null when ***i*** equals the size of the data set. Whenever the author wants to ensure that the previous or next element is not empty he can simply check if the value of prev or next is not null.

The remove method from the ListIterator interface could be replaced using the splice method by passing the associated variable that defines the position of the element that must be removed. Splice will automatically reduce the size of the length, so there is no need to worry if the index runs out of bounds, as every time the loop ends, the length of ***ds*** will automatically change and the appropriate values will be assigned to the local variables when the loop is accessed again.

**Testing**

During the testing phase, the author spotted many interesting mistakes that were caused by the new logic. To ensure that each part of the code was producing correct outputs, the author had to understand clearly how each block of code was executing in the RoutesFinder.java first. After commenting each block of code with a name, the author was drawing at a paper the procedure that was going to be executed to help him implement the new logic to the logic.js file. Then the outputs were printed to the console and they were compared with the correct outputs in the spreadsheet KEG provided to the author. An example could be seen in the figures below.

|  |
| --- |
| Figure 5: RoutsFinder.java (KEG) |

|  |
| --- |
| Figure 6: Execution Process (Author) |

Figure 5 illustrates the process to remove the current and the next element (if it exists) when the current destination is depot and the previous is not depot and contains the START value. It also checks if the element before previous contains the LAST value, if not it assigns the value to the end of the String. To understand how this works, the author draw boxes as the elements as it can be seen in the figure 6. The pointer is always located next to the current element when it starts, which now is selected. Whenever the pointer moves to the right or left, it selects the element that has been passed through. Now, when the else if statement executes the pointer moves to the left (it.previous()) and now is located left of the current element. It checks if there is an element to left and moves the pointer to the next element on the left. For example, in the figure 6, the pointer now will be located on the left of the previous element. At this point, it checks again if there is an element to left and if it doesn’t contain the value of LAST already, assign it at the end of the string. Then it moves twice to the right, placing the pointer next to the element with the caption Current which is the currently selected element. Now it removes it and checks if there is another element to the right. If this is true, the pointer moves to the right and removes the next element.

|  |
| --- |
| Figure7: logic.js (Author)] |

|  |
| --- |
| Figure 8: test.json (Author) |

As described above, after understanding the execution, the same block of code was re-written in the logic.js. The interesting part with the new logic is that whenever an element is removed, the local variables curr, prev and next are not updated. For example, in that case, when the current row is removed the next element, automatically becomes current. To remove 2 elements in a row, the author must call the splice method with the same variable twice. For this example, the variable is the curr. It’s like a stack of papers, if you remove the second paper, the third automatically becomes the second. To ensure the output was correct the author created test data (figure 8). After commenting all the previous blocks of code, the author was able to examine the results on the console as it can be seen in the figure above.

|  |
| --- |
|  |

By repeating the same process, the author managed to rewrite the whole RoutesFinder.java to JavaScript. After testing all outputs on the console, the author proceeded to create the basic visualisation.

**Simple Visualisation**

|  |
| --- |
| Figure 9 demo (Author) |

|  |
| --- |
| Figure 10 demo.js (Author) |

The first deliverable for the first iteration was basic and simple. The author created 2 methods that create the shapes. After reading and checking the data, the results were stored to the associated variables for each shape. The square was responsible to filter the data based on given values such as depot, shed and sidings. Furthermore, while creating the shapes, the onClick() method was added to attach the table with the data on the DOM when the user clicks the associated shape.

### 

### 5.1.2 Iteration 2

### 5.1.3 Iteration 3

## 5.2 Alternative Results

## 5.2 Complexity

## 5.3 Testing

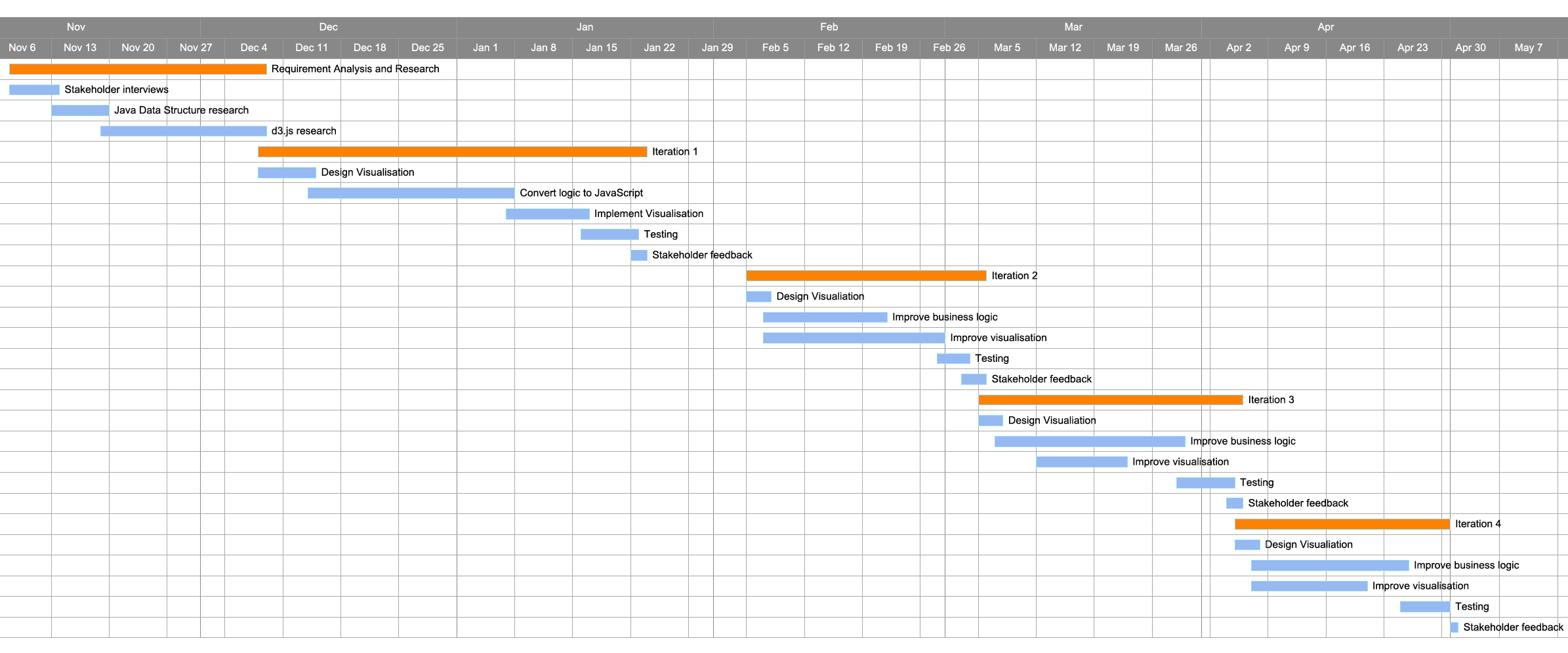
## 5.4 Evaluation

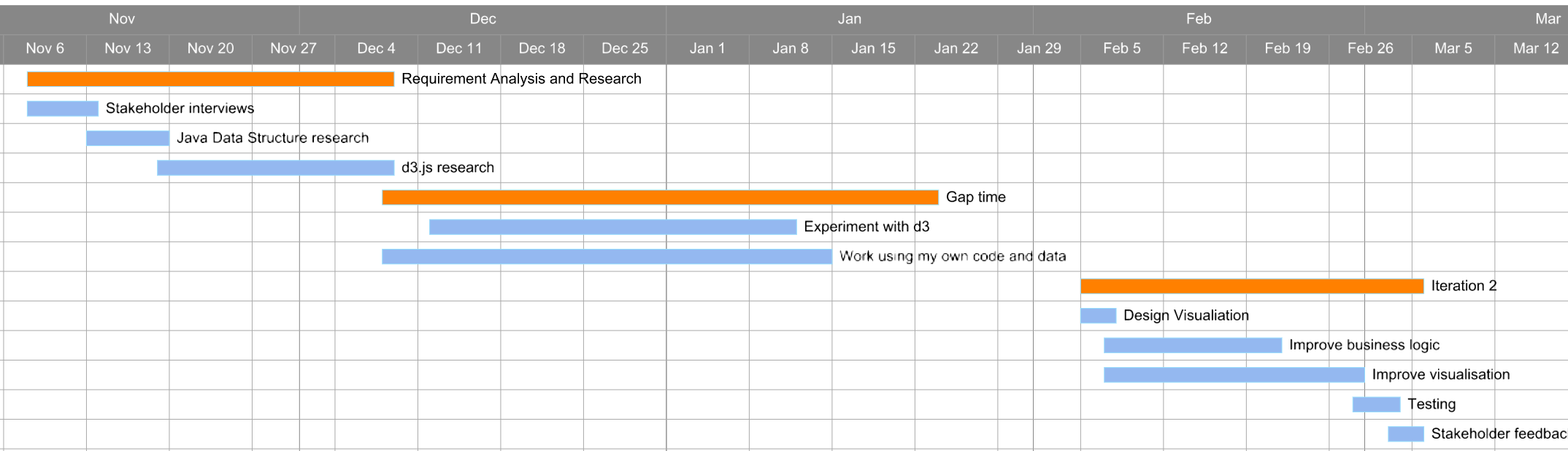
* Complexity problems
* The data list could have been converted to a list of strings like java

# 6. Conclusion

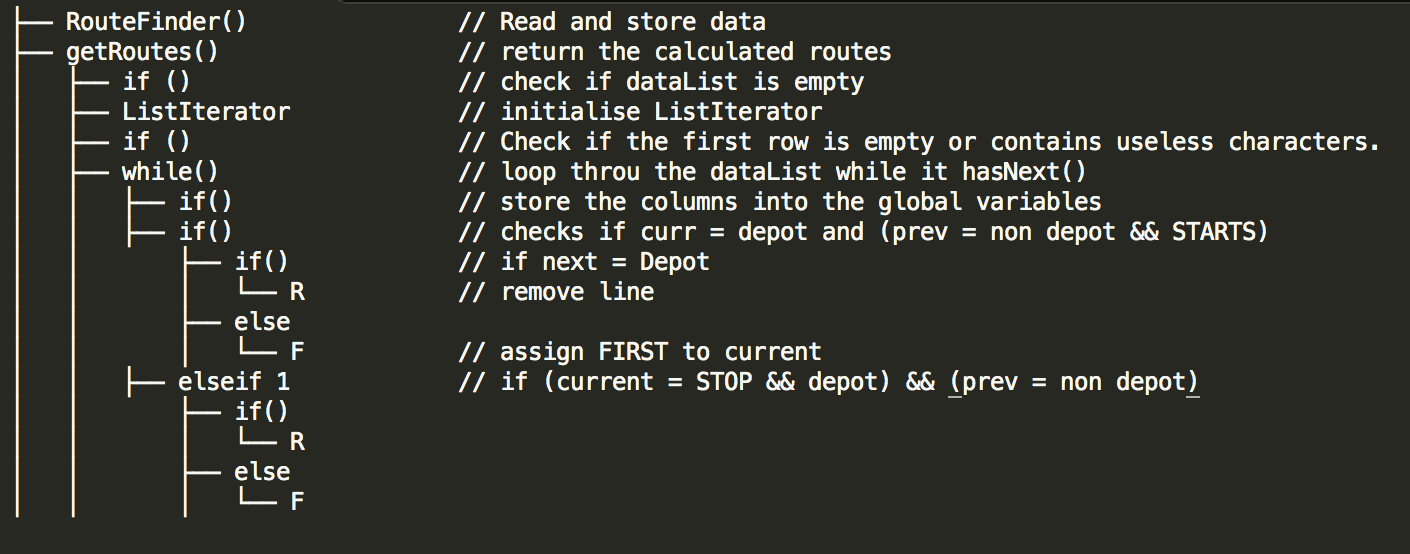
# APPENIDX A

Gant Chart





# APPENIDX B

**Java Code Structure**

Because the routes finder application contains 660 lines of code the author tried to structure the code in a simpler format to be more readable to help him understand it clearly and re-write the code into JavaScript.