School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

Train Crew Schedule Information System

Harry Patrick Duce

Submitted in accordance with the requirements for the degree of
BSc Computer Science

2014/2015

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Deliverable 1	*Report*	*SSO (03/06/2015)*
Participant consent forms	*Signed forms in envelop*	*SSO (03/06/2015)*
Deliverable 2	*Software Application Code*	*Emailed to supervisor and assessor (02/06/2015)*
Deliverable 3	*Data Sets*	*Emailed to supervisor and assessor (02/06/2015)*
Deliverable 4	*User manual*	*Emailed to assessor and supervisor (02/06/2015)*

Type of Project: Exploratory Software (ESW)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student) _____

Summary

A train planners role when processing a train crew schedule is to minimise costs whilst meeting the constraints of the process. Train crew scheduling involves handling masses of data relating to the train journeys that have already been defined and the potential crew shifts that are available to be added to the schedule. The data is funnelled through an algorithm so that a potential schedule can be formed. This project aims to provide train planners with a user interface that enables them to visualise and navigate through the data of a potential train crew schedule, whilst allowing them to modify a schedule where they see fit in an effort to aid them in minimising costs.

Acknowledgements

I would like to thank my project supervisor Dr Raymond Kwan for all the advice and expert knowledge he provided throughout the project. I would also like to thank Professor Kristina Vuskovic for the feedback provided during the progress meeting.

Table of Contents

Abbreviations

Phrases that are used commonly throughout the report have been abbreviated too:

- RO – Relief opportunity
- WP – Work piece
- WS – Work spell
- WPF - Windows Presentation Foundation

1. Introduction

1.1 Problem Statement

Train crew scheduling is a complex task that involves organising potentially thousands of possible crew shifts into a vehicle schedule every day (Opcom, c2001, p1). For a Train Operating Company the cost of crew shifts is often the most expensive part of operating costs therefore train crew scheduling is a very important part of the planning of railway operations process (Laplagne, 2008, p4). The data that details a vehicle schedule and the available crew shifts for a specific day comes in the form of two plain text files that can stretch up to tens of thousands of integer values long. A train planner needs to be able to navigate through this data to process a crew schedule from the resources available that minimises costs.

1.2 Project Aim

The purpose of this project is to design and develop a software application that makes visualising and navigating through the large amounts of data involved in a train crew schedule an easy process that aids in minimising costs by creating an interactive visual schedule.

1.3 Project Plan

The basic plan for the project is as follows:

- Perform background research to gain an understanding of the subject area. This will begin by finding background reading and gathering relevant literature and then studying the alternative solutions along with the two data sets that are involved in train crew scheduling. The minimum requirements and possible extensions will be devised during this process
- Devise a sensible work schedule based on the time available and the minimum requirements devised for the project
- Applying knowledge gained from the background research to create a basic plan for the application that is to be produced. This process will involve experimenting with interface designs, deciding upon functionality and creating a basic class structure for the application
- Research for appropriate software tools to develop the application based on the basic blueprint already created

- Begin development of the designed application following the work schedule
- If time allows, attempt possible extensions
- Evaluate the final application

1.4 Minimum Requirements

The minimum requirements for the solution of this project are:

1. Read and prepare the data that describes a daily train crew schedule into a sensible class structure
2. A software system which uses the prepared data to create a visual display of a train crew schedule that is contained within a graphical user interface
3. Allow navigation through the train crew schedules data via user interaction with the visual display
4. Enable the user to modify the schedule e.g. by adding and deleting shifts

1.4.1 Possible extensions

- The application lets the user save the crew schedule to a file
- Implement 3D graphics into the visual schedule display
- Improve an algorithm used to generate crew schedules

1.5 Selected Methodology for this Project

Development methodologies define a framework that is used to form a plan of action for a software project. There is a wide array of methodologies, each with their own characteristics that make them suitable for some projects and not others. Choosing a methodology for a project influences which, how and in what order processes are performed, therefore the choice can potentially negatively impact the quality of the solution which means it is sensible to consider multiple options. (CENTERS for MEDICARE & MEDICAID SERVICES, 2008, p1)

1.5.1 Waterfall

This method divides the project into sequential stages:

1. Background investigation and requirements definition
2. System design
3. Design implementation
4. Testing and maintenance

There is little scope for moving backwards through the process and if a stage is conducted inadequately the subsequent activities will have to accommodate the deficiencies. Consequently, the Waterfall method is most suitable to longer term projects that have the

capacity and flexibility to make provision for full completion of each stage prior to commencing the next. (Maheshwari and Jain, 2012, p286)

1.5.2 Prototyping

A methodology that promotes user involvement by dividing a project into segments where prototypes are created and presented to them. This enables frequent user feedback so that the solution being created constantly adheres to the customer needs. If there is little interaction with the user then the benefits of this methodology are not taken full advantage of. (Maheshwari and Jain, 2012, p287)

1.5.3 Iterative Waterfall Development

This was created with the purpose of being a faster and more flexible version of the Waterfall model, splitting the software development into incremental stages which are performed in iterations. Each iteration receives feedback from the previous stage to improve and add functionality to the solution. (Maheshwari and Jain, 2012, p287)

1.5.4 Selected Methodology

The minimum requirements defined for this project are generic and will not be well understood until developing the user interface is well underway. Therefore a flexible methodology is required which rules out the Waterfall method. There will be little to no chance of interacting with a train planner, who is the intended user of the solution which nullifies many of the benefits of prototyping. This leaves the iterative methodology as the most suitable for the project.

1.6 Relevance to Degree

The second year Software Engineering module took students through the motions of a how to perform a software project. With this being a software project, a lot of the knowledge gained from the module can be applied to creating the plan/structure of the structure. Another second year module, Graphical User Interfaces, taught students the general process of creating a user interface using popular languages and GUI frameworks.

1.7 Project Schedule

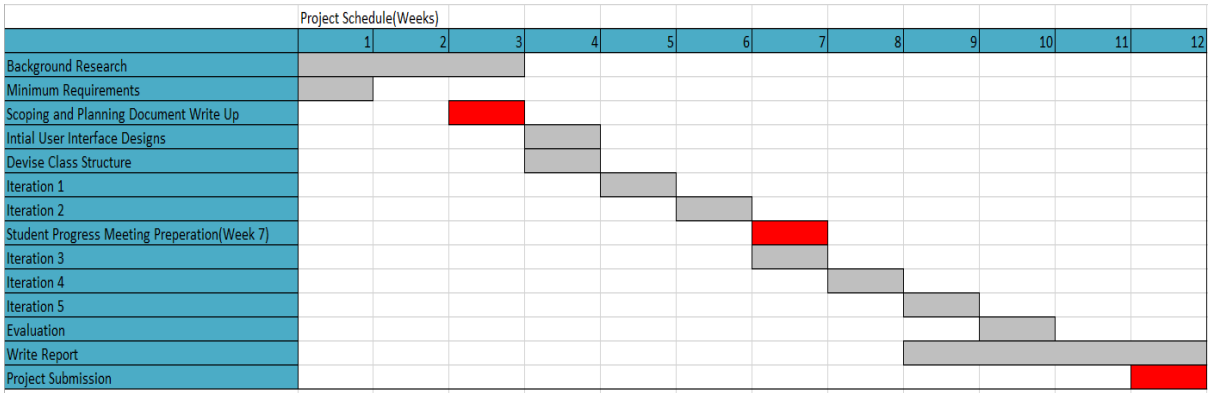


Figure 1 Gantt Chart of initial schedule that was drawn up in week 1. Red cells represent deadlines.

The schedule chart was made at the start of the project with purpose of listing all tasks to be undertaken and sequencing them appropriately so that the project will be completed in the required timescale.

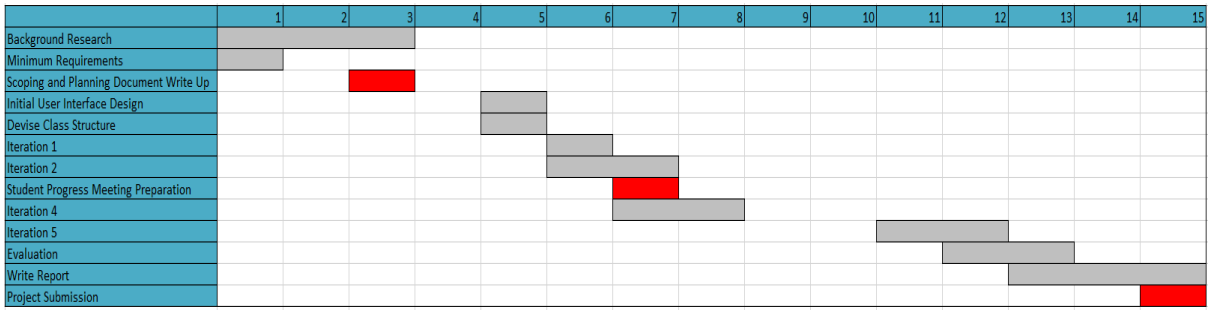


Figure 2 Gantt Chart of revised schedule that was made once the extension was granted for the project

1.8 Structure of Report

The report is split into sections that explain the project as it took place:

- **Background Research** – Details the reading and review of materials that took place throughout the project
- **Design of Application** – Explains the process of designing a user interface and class structure for an application using acquired knowledge from background reading and past studies. Also explains the reasoning behind why certain software development tools were decided upon
- **Implementation** – Describes the development process of the designed software application
- **Evaluation** – Describes the evaluation processes that were performed to gauge an understanding of the success of the solution

2. Background Research

2.1 Introduction

This section outlines the background information that was gathered and studied to help gain an expert understanding of the problem. Definitions of key terminology used throughout the project are outlined, an alternative solution is studied and development tools are discussed within.

2.2 Planning of Rail Operations

There are four general stages involved in the process of planning of rail operations:

1. **Timetabling** – The initial stage in the planning of rail operations where a timetable is created that is compiled of services and trips (Laplagne, 2008, p3)
2. **Vehicle Scheduling** – The second stage of planning where all train units available are allocated so that each service and trip is covered (Laplagne, 2008, p3)
3. **Train Crew Scheduling** – This process allocates each piece of work in the vehicle schedule a crew from the crew shifts available to create a train crew schedule (Kwan, 2015). A train crew schedule is only valid if every WP is covered by a shift (Albers, c2009, p38)
4. **Rostering** – This stage occurs in rail operations planning defines the type of work for each member of every crew (Albers, c2009, p12)

2.2.1 Defining Terminology

- **Vehicle Block** – Represents a section of consecutive pieces of work performed by a train unit over a certain timeframe (Laplagne, 2008, p13)
- **Vehicle Schedule** – The set of all vehicle blocks within the scope of a specific timeframe
- **Relief Point** – A location within a vehicle schedule where is it possible for a crew to be relieved and another to take over (Albers, c2009, p38)
- **RO** – A specific time and location within a vehicle block where it is suitable for a crew to be relieved and another crew to take over (Kwan, 2009, p47)
- **WP/Piece of Work** – The summary of all work performed between 2 consecutive ROs within a vehicle block. Therefore within a vehicle block:

  Number of WPs = Number of ROs - 1

- **Shift** – Defines all of the work that a specific crew is to perform within a schedule

- **WS** – These are the portions of time within a shift that a crew will spend actually working, splitting the shifts up by breaks. A shift can be made up of multiple WSs
- **Train Crew Schedule** – A vehicle schedule that has been allocated shifts to cover the WPs performed by the train units
- **Candidate Shift** – A shift that is eligible to be added to a vehicle schedule the vehicle schedule in question
- **Overcovering** – A WP is overcovered when more than one shift in a crew schedule is set to cover it. The ultimate aim is to produce a crew schedule where every WP is covered by one crew exactly, however a schedule is still valid even with overcovering (Excess crew members can ride the vehicle as passengers) (Laplagne, 2008, p15)

2.2.2 Train Crew Scheduling

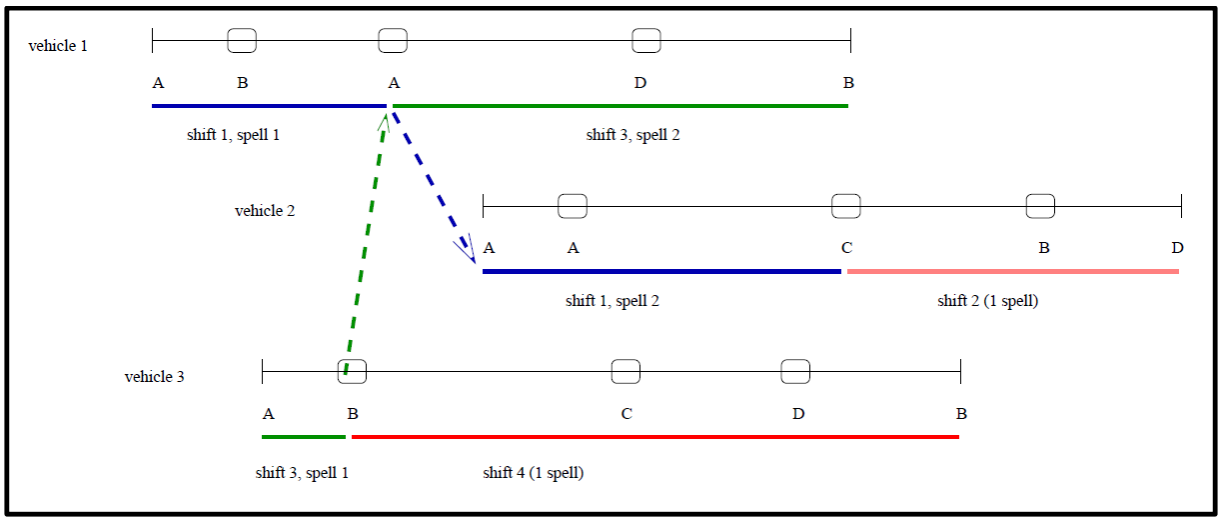


Figure 3 A representation of a train crew schedule (Laplagne, 2008, p15)

The train crew schedule above consists of three vehicle blocks. The horizontal black lines represent the timeline of the trips and services performed by each vehicle. The square shapes placed along the black lines each represent a RO. The ROs split the black lines up into WPs. A label is attached to each RO that displays the location value. The coloured horizontal lines each represent a WS, the colour defines what shift the WS belongs too. There are four shifts and six WSs being displayed. The coloured lines stretch along the vehicle lines visualising where and when the crew will start and end their WSs. This specific train crew schedule is valid because every WP is covered by a shift. The train crew scheduling process does not need to take into account what specific jobs crew members are to perform.

2.3 Data Files Format

The data that the software system must take as input are two plane text data files that are the output of a commercial train crew scheduling system (Kwan, 2015). They each contain integer values exclusively and have specific formats that need to be understood to be able correctly read the data into an application. The two files come with extensions .trains and .shifts.

2.3.1 Trains File

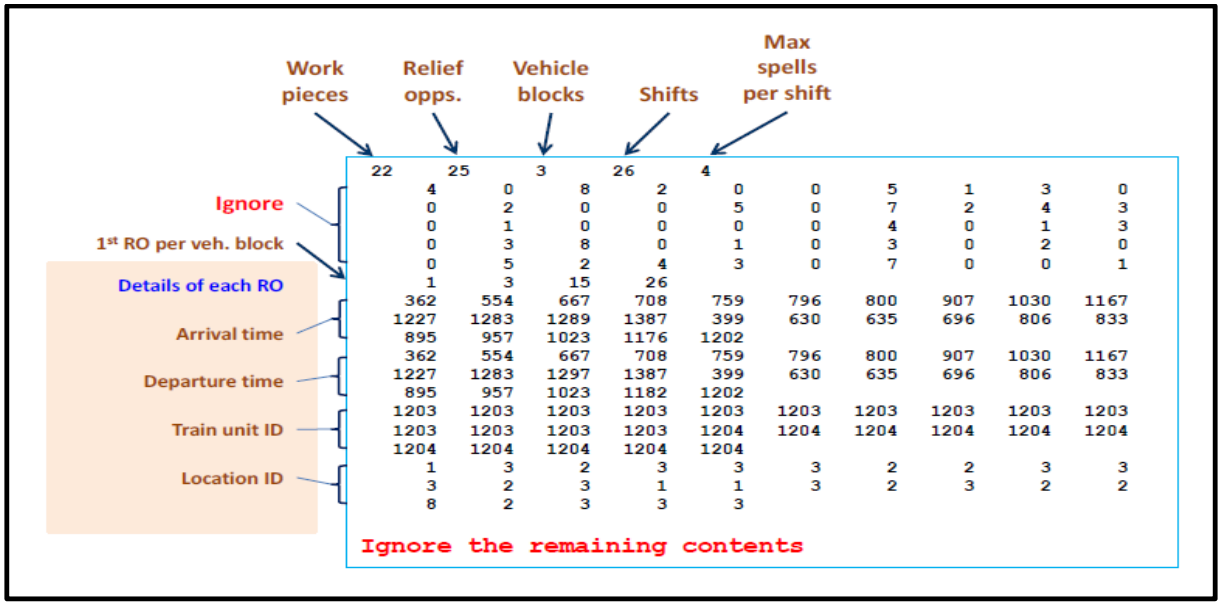


Figure 4 Sample layout illustrating the format of a Trains file and identifying which data is not relevant. This illustration is taken from a material supplied by the project supervisor.

This file supplies the data that details the vehicle schedule. From this sample we can deduce that the schedule is split into three vehicle blocks; the first containing two RO's, the second has twelve and the third has eleven. Arrival and departure times are given for each RO along with the location identification. The location value will refer to a specific place in the world e.g. Location ID 1 = Liverpool and Location ID 3 = London. The identification of the specific train vehicle that will arrive at each RO is also included in the file as the Train unit ID.

2.3.2 Shifts File

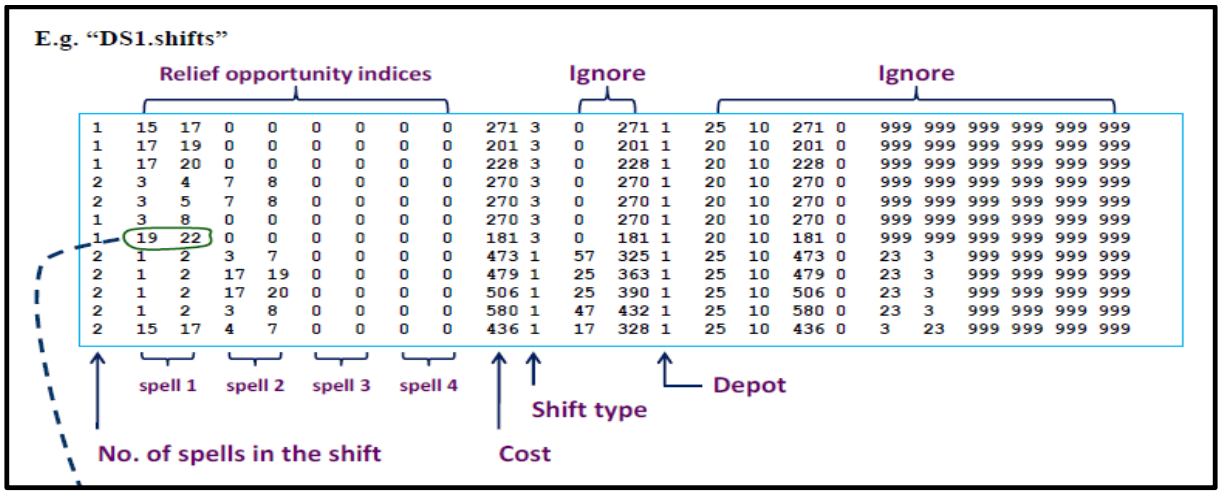


Figure 5 Sample layout illustrating the format of a Shifts file and identifying which data is not relevant. This illustration is taken from a material supplied by the project supervisor.

This file contains data for all the shifts that are available to be allocated onto the vehicle schedule that is detailed in the corresponding trains file. Each line of integers represents an individual shift, therefore:

Number of candidate shifts = Number of lines in file

The number of WSs in each shift is specified, along with the ROs that each WS occurs between. The cost of each shift is defined by the amount of minutes spent working and the shift type value determines what that particular shift is classified as and the depot value refers to the location that the crew has to report to, often at the start and end of a shift, via its identification number. The Depot value corresponds to the location values given in the .trains file.

2.4 Alternate solution

2.4.1 Introduction

During the first project supervision meeting, Raymond Kwan presented and discussed a Python program titled GreedyHeuristics-2.py (GreedyHeuristics). This is an alternative solution to the train crew scheduling problem due to it being a command line program that offers no user interface. He instructed that this program should be used as a template for reading in and preparing the data from the plain text files. He also suggested that the algorithms used within this program could be used in the solution as a means for generating an initial train crew schedule. Therefore a thorough understanding of the program was a necessary prerequisite.

2.4.2 Program Description

This is a command line program written in Python. A basic outline of the steps this program takes from start to finish are as follows:

1. Takes a Trains file and Shifts file as input at run-time and reads in and stores the data according to the file formats
2. RO, WP and shift objects are created iteratively, with their properties being assigned values from the data. They are aggregated into three separate lists based on their class type
3. An algorithm is then executed, using the three lists from step 2 to generate a train crew schedule which is stored as a list of shift objects. Contained within the Python code are three separate algorithms that all returns a train crew schedule, to switch between what algorithm is executed, the developer must comment in the code they want
4. The generated train crew schedule is then outputted as a string that displays the number of shifts in the schedule, and the ID values of each shift

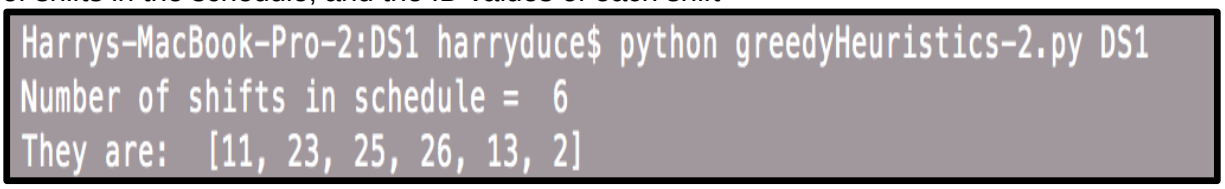


Figure 6 A screen shot of the program being executed from the command line.

Figure 6 presents the execution of GreedyHeuristics when the two sample files displayed in **figures four and five** are passed as input. The output tells the user the size of the crew schedule that has been generated in terms of the number of shifts allocated. The identification number of each shift in the schedule is also presented.

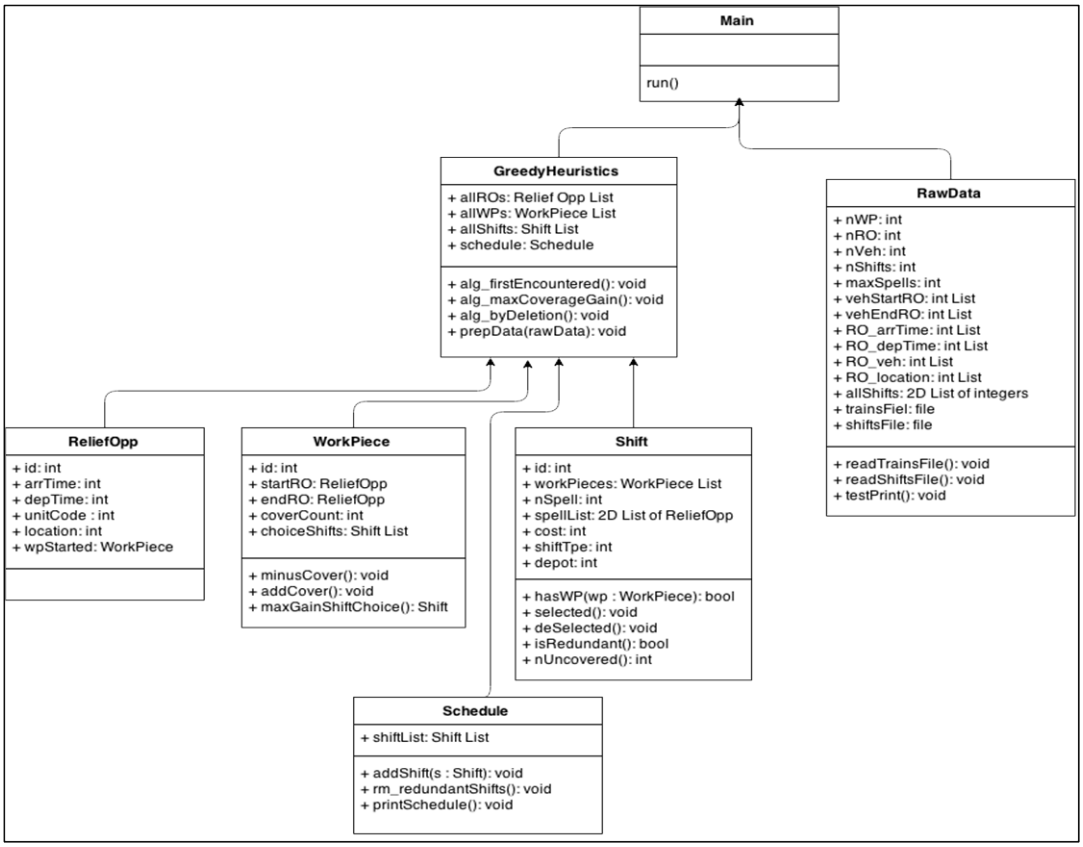


Figure 7 Class Diagram detailing the class structure of GreedyHeuristics.

2.4.3 Critique

Evaluating GreedyHeuristics led to the following conclusions:

Pros:

- Reads in and stores the raw data from the files as sensible property values
- Creates useful objects that represent important components of a train crew schedule, e.g. ROs, WPs and shifts. These objects have the potential to be used for further functionality and not just generating a train crew schedule in the form of a list of shifts

Cons:

- The output presents a small fraction of the information that is being processed within the program. For a user to gather further information on the schedule produced they would be forced to refer to the data files
- There is no functionality that allows the user to manually modify the generated train crew schedule

2.5 Programming Languages

The minimum requirements defined in **1.4** formed an outline of what programming practices would have to be used during the creation of the final solution:

- Reading in file data (requirement 1)
- Object orientated programming (requirement 1)
- Graphical User Interface programming (requirement 2)

Therefore these were the characteristics that were used as filters when deciding upon the initial list of programming languages that would be most suitable for the project.

From the first year of the University of Leeds Computer Science degree C++ is taught and it was used as one of the programming languages that introduced students to object orientated programming. In the second year of the course the Graphical User Interface module is taught using C++ and the Qt framework. Due to the knowledge and skills gained from the degree course, C++ was the most suitable programming language to be chosen for the project based on the minimum requirements.

3. Design of Application

3.1 Introduction

This section of the report describes the design process of the project in the order in which it happened. A ‘from the ground up’ approach was taken, with a basic class structure for the application being devised first and then the initial interface design being created based upon its capabilities. The purpose of this process was to take the knowledge gained from the background research that had been collected and use it to design an effective solution for the problem stated in section **1.1**. Creating the design also gave better understanding as to what software development tools would be best suited for creating the solution.

3.2 Basic Class Structure

As is stated in section **2.4.1**, the GreedyHeuristics program could be used as a basic template for reading and preparing the files data. Therefore the class structure in **Figure 7** was the starting point for the basic class structure design for the project solution. However, changes were made for reasons listed below:

- Two new class types have been introduced called *Trains* and *Shifts*. The purpose of these is to keep a clear separation between the raw data read in from the Trains file and the Shifts file
- The *GreedyHeuristics* class type has been renamed to *PrepareData*. This small change has been made because the name *PrepareData* better represents the functionality of the class in terms of importance of achieving the solution
- The *Schedule* object is to be instantiated within the Main Window instead of *PrepareData*. This is to reduce the gap between the user interface and the train crew schedule data

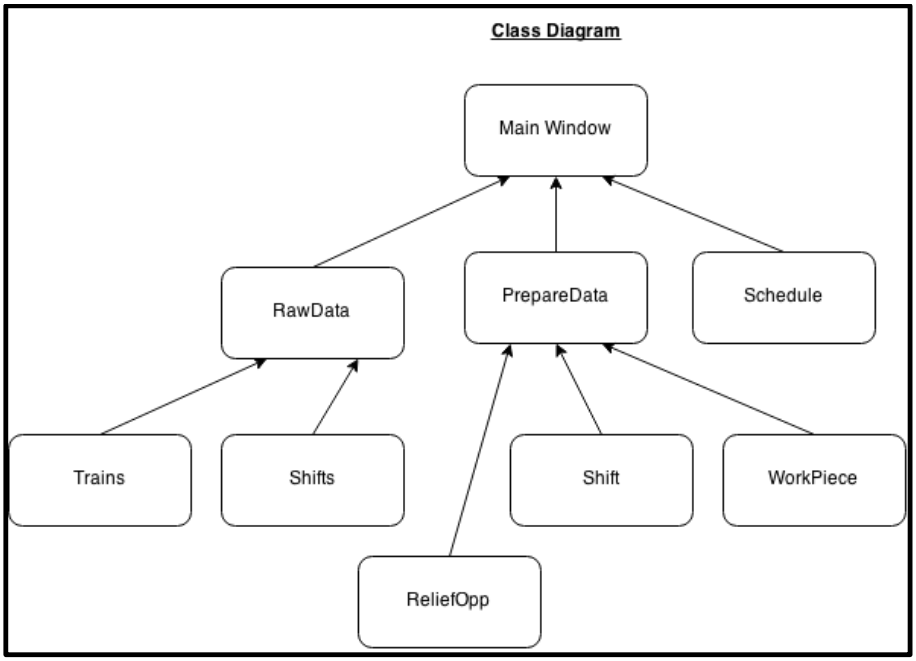


Figure 8 Simple drawing displaying the basic class structure

3.3 Visual Display of Schedule

3.3.1 Considerations

Throughout the design and development of the user interface, the following questions were considered:

- **Is the display easy to read and understand?** - It is important that the interface is self-explanatory and makes logical sense so that the user can visualise and navigate through the data as effectively as possible
- **Is the data being displayed relevant?** - Will the data that the interface is presenting actually aid a train planner
- **Is the display visually pleasing?** - Though not as important as the other questions, a visually displeasing interface could potentially discourage users in their work

3.3.2 Initial Schedule Model

As is discussed in section **2.2.1**, a vehicle schedule is a set of vehicle blocks compiled together and a train crew schedule is a vehicle schedule that has been allocated shifts to cover the WPs. Therefore a train crew schedule can be visually formed by iteratively creating vehicle blocks and grouping them together. A vehicle blocks size is defined by the start time of the first RO and the finish time of the last WP. By stacking vehicle blocks on top of one another, with each starting at the same longitudinal coordinate, the difference in length would be easily visualised. However, in terms of a daily schedule, this method would give no indication as to what time of day the vehicle blocks components take place. To achieve this,

each vehicle block is placed inside a container that has a constant width. Each horizontal unit of space inside a container represents an amount of time. The left and right end of the containers define the time frame that the train crew schedule takes place in. A vehicle blocks position along the x-axis of the container is decided by the start time of the first RO. Then if each container is directly stacked on top of one another, this should give the overall schedule a global timeline and enable the user to visualise what times the vehicle block components happen in relation to each other.

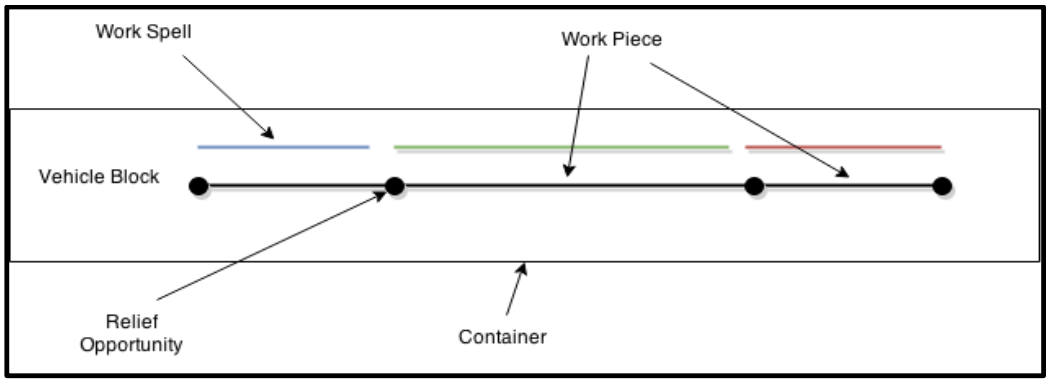


Figure 9 Initial vehicle block design

Figure 9 was the first blueprint for a vehicle block. The concept was that the nature of the shapes (e.g. the colour and size) representing each component should give information about the data behind it:

- **RO** – As is discussed in section **2.2.1**, a RO represents a specific time. A small circle is used to give the effect that this is a small point in time. The colour black is used for both ROs and WPs to express that these are constant values within a train crew schedule. They do not change
- **WP** – A horizontal line is used to display a stretch in time. The length of the line indicates the duration of the WP. As stated above, the colour black is used to express that this is a constant component in the train crew schedule
- **WS** – For the same reasons as a WP, a horizontal line is used. WS's have contrasting colours to represent that they are part of a different crew shift

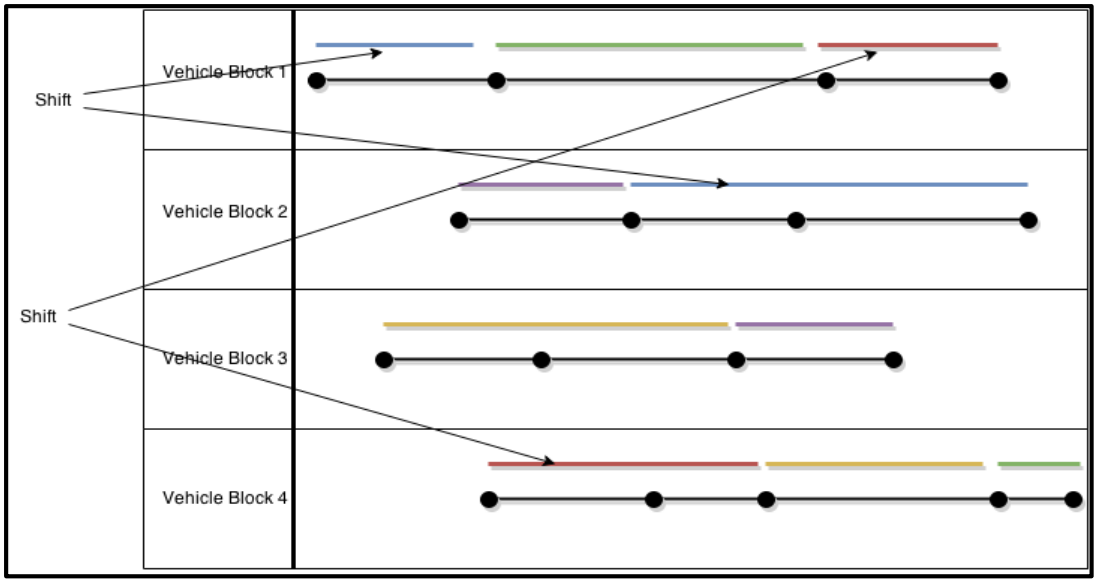


Figure 10 Design displaying stacked vehicle blocks forming a schedule

As is discussed above, a train crew schedule can be visually formed by stacking vehicle blocks directly on top of each other, this is shown in **Figure 10**. Listed below are some examples of what can be determined from the data being displayed in **Figure 10** that would be useful for a train planner:

- Which WPs and vehicle blocks a shift covers
- The number of RO's and WP's within each vehicle block
- The ascending order of vehicle block start times

What cannot be determined from the data being displayed are the actual time values that each component represents, for example, the actual start time of vehicle block 1.

3.3.2.1 Visual Timeline

As is discussed in the above section, the left and right end of the containers define the period of time that the entire train crew schedule takes place within. Therefore for the time values of each component to be determined by the user through visual representation, a timeline that displays the time frame of the vehicle schedule is required. A sensible time frame for a daily schedule was taken to be 26 hours, starting at 00:00 and ending at 02:00 the next day, the extra two hours added on to cover for WPs that run over into the early hours of the morning. However, upon reviewing the data files, there are WPs with durations as short as five minutes. Displaying these in a visual schedule that covers twenty six hours when the actual timeframe of the schedule is less whilst there is limited screen space would have resulted in compacted components, making the interface difficult to read.

If the width of the visual vehicle block container was a known, constant value, e.g. 100 units, and the earliest start time and the latest finish time of the entire schedule could be

determined from the file data, e.g. start time 0200 and end time 1200, then it would be possible to calculate the amount of time that each horizontal unit in the container represents:

(Finish time – Start Time) / Width = Amount of minutes per unit

(1200 – 200) / 100 = 10 minutes per unit

With these values it would then be possible to:

- Assign the left end of the container with the earliest start time value and the right end with the latest finish time value
- Position components longitudinally within the vehicle block by converting their start time values to units of the container
- Create a visual timeline illustrating the time intervals along the schedule

This method would more efficiently pack the visual components of a vehicle block into the container by basing the timeframe on the actual data values from each data set.

3.3.3 Displaying RO and WP Data

The schedule design (**Figure 10**) along with a visual timeline displays some of the data behind each component in the schedule, however there are properties that cannot be distinguished, e.g. the unit code of the vehicle at each RO, the location of each RO and the cost of a shift. These are all values that need to be accessible to a train planner, but cannot all be directly added to the visual schedule because there would not be enough space and it would become unreadable due to the vast amount of numbers that would be scattered across each vehicle block. Therefore alternative methods had to be devised and explored:

1. A network of combo box styled lists that enables the user to select and view specific components and their properties from lists of identification values, e.g. a drop down list containing all ROs by their ID value.
    - Using this method would force the user to take their attention away from the visualised schedule and instead spend time scrolling through lists of integer values until they find the specific component they were looking for. Therefore the goal became looking for methods where user interaction with the visual schedule would reveal more data

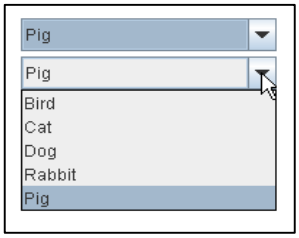


Figure 11 Generic Combo Box (Jaksmata, 2008)

2. The next possibility was to let the user select a vehicle block, which would open a new window containing a blown up version of that vehicle block. In the newly available space the property values would be listed next to each component
    - This method would create interaction between the user and the visual schedule by enabling them to select vehicle blocks. It would also aid in visualisation of the data because it would be drawing the data into the vehicle blocks display. However, this method was not preferred. A typical use case for a train planner using a piece of software of this nature would involve them navigating to find a value for a specific component, not an entire vehicle blocks worth

3. This design idea came about whilst contemplating how a user could select a vehicle block in part two above. The basic principle was that every basic component within the visual representation of the schedule would be selectable e.g. the RO circles, WP lines and shift lines that are drawn in **Figure 10**. After selecting a component the corresponding object stored in the application is retrieved and the property values are displayed via table format in a relatively small new window
    - This method would mean that all the data used to form the visual schedule is accessible via observing and interacting with the initial schedule interface. It achieves this whilst also not having to makes any differences to the initial design discussed in **3.3.2**

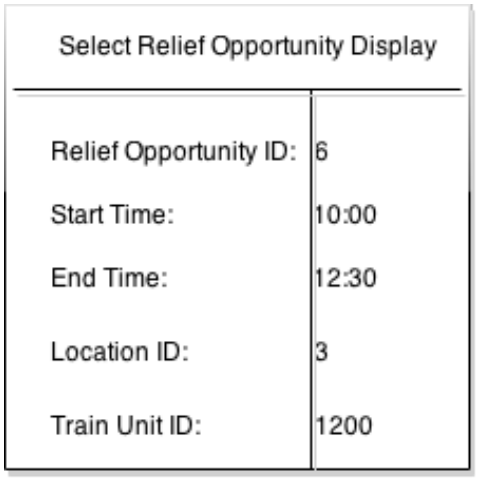


Figure 12 Example design of window displaying selected components properties

With the third of the three methods being preferred over the others, the next stage was to find software that allows the manual positioning of selectable shapes that are created via data objects and placed within a visual container. As is discussed in section **2.5**, based on

the minimum requirements C++ with the Qt framework was the most suitable candidate for the programming language to be used in the implementation. Therefore the search for the desired software functionality began with the different Qt Containers that enable developers to have control over visual objects within a form (Qt, c2015). Upon reviewing each container type, it was concluded that none would be suitable due to the lack of being able to position objects absolutely within them. This led to searching for alternative GUI frameworks that did offer such functionality.

3.3.3.1 Windows Presentation Foundation

Windows Presentation Foundation (WPF), is a presentation system used in Windows-based applications for rendering graphical user interfaces. Discussed below are potentially useful properties of WPF that could be used together to achieve the desired selectable shapes functionality:

- **Control Class –** Defines the properties and methods shared between all components that use a *ControlTemplate* to determine their graphical presentation (Microsoft, c2015, *Control Class*)
- **Canvas Class -** A container control, derived from the **Panel Class**, that allows developers to position controls within the container absolutely using coordinate values (Microsoft, c2015, *Introduction to WPF*; Microsoft, c2015, *WPF Container Controls Overview*)
- ***Shapes* namespace -** Gives developers access to a library of shape classes, e.g. *Ellipse*, *Rectangle* and *Line*. Each of these classes have properties that enable customisation of the shapes, e.g. *Width* property for the *Rectangle* class (Microsoft, c2015, *System.Windows.Shapes Namespace*; Microsoft, c2015, *Rectangle Class*)
- **Data binding –** Allows for graphical elements to be bound to data from within the applications data storage (Microsoft, c2015, *Data Binding Overview*)
- ***ListBox* class -** Inherits from the *ItemsControl* Class that is used to present a list of selectable items. The *ListBox* class has a property called *ItemsPanel* that allows the developer to define the **Panel Class** that its items will be drawn on to. *ListBox*'s *ItemTemplate* property means the visual structure of the items can be customised. The *ItemsSource* property is the collection of data values that is contained by the *ListBox* (Microsoft, c2015, *ListBox Class*; Microsoft, c2015, *ItemsControl.ItemTemplate Property*; Microsoft, c2015, *ItemsControl.ItemsPanel Property*)

So in theory, if a *ListBox* could be bound to a list of WP objects, its *ItemsPanel* customised to be a Canvas and its *ItemTemplate* set as a rectangle shape that's width can be modified based on a property of a WP. Then the goal of having selectable shape items can be achieved.

3.3.3.2 Experimentation with WPF

The idea expressed above was implemented using C# and WPF. The code used to create and customise the *ListBox* that contained the circle objects is displayed in **Figure 13** below. The collection of *Circle* objects that was to be visually displayed was bound to the *ListBox* in the application logic using its *ItemsSource* property.

```
<!--Experimentation With ListBoxs-->
<ListBox Name="CirclesListBox" BorderThickness="0" Height="50">
    <ListBox.ItemContainerStyle>
        <Style TargetType="{x:Type ListBoxItem}">
            <Setter Property="Canvas.Left"
                    Value="{Binding positionValue}"/>
            <Setter Property="Canvas.Top"
                    Value="5"/>
        </Style>
    </ListBox.ItemContainerStyle>
    <ListBox.ItemsPanel>
        <!--Defines the ListBox's ItemsPanel property-->
        <ItemsPanelTemplate>
            <Canvas/>
        </ItemsPanelTemplate>
    </ListBox.ItemsPanel>
    <!--Defines the ListBox's ItemsTemplate property-->
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Ellipse Fill="Black" Width="12" Height="12"/>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Figure 13 Code used to create and customise the *ListBox* containing circle items. Comments are included to outline what was discussed in 3.3.2.2.1.

The results were a success as can be seen in **Figure 14** which displays the graphical output.

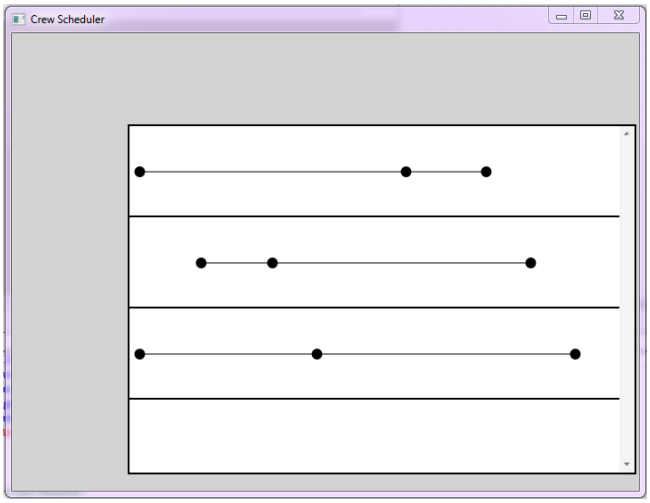


Figure 14 Screen cap of *ListBox* with *Canvas* template being executed.

3.3.4 Main Window

The main window of the application will contain the visual train crew schedule, including the timeline. The selectable components that compile together to make the visual schedule enables access to all the data that describes the train crew schedule via observing and interacting with the display, e.g. to find the final arrival time of the entire schedule shown in **Figure 15**, the user can observe that vehicle block 4 ends last, then select the shift covering the final work piece by clicking the green rectangle to unveil its properties which will include the finish time, thus giving the user their answer.

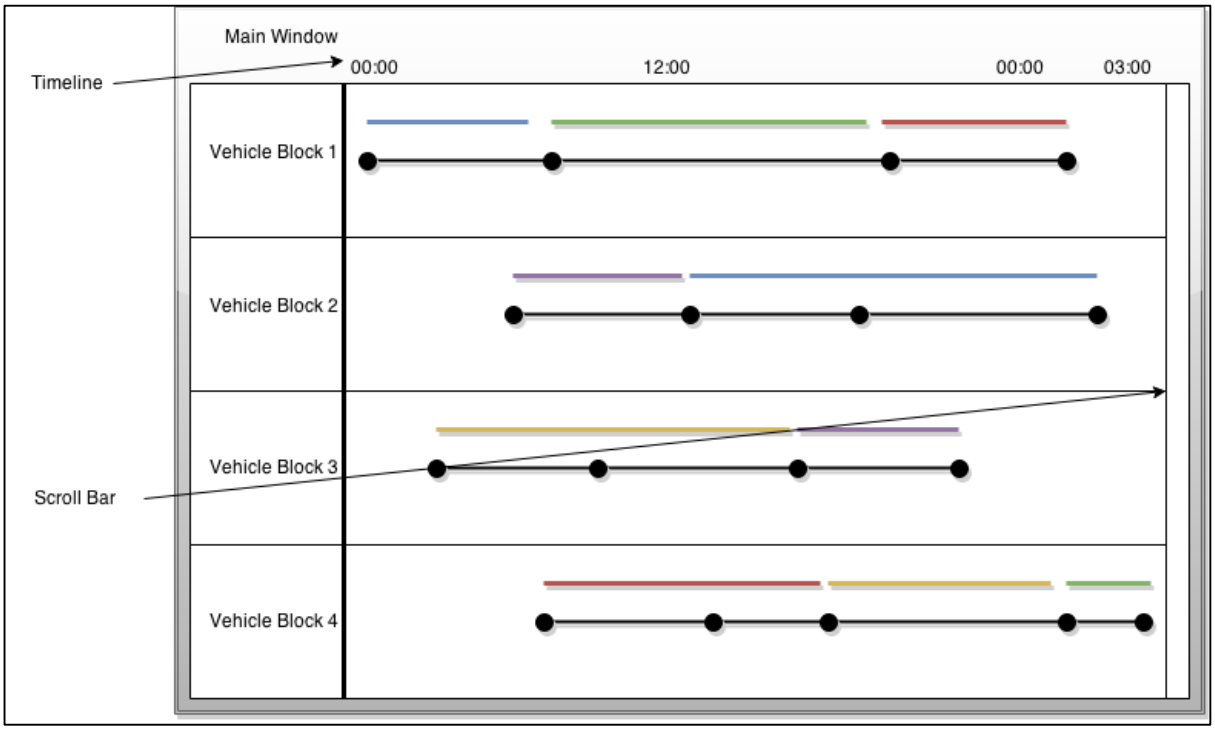


Figure 15 Design of main window

3.3.5 Specified Shift Display Window

When considering how the application would scale as the data set size increase, it lead to thinking how the user could efficiently inspect a specific shift in the schedule. The problem being that if a shift has a WS covering a WP on vehicle block 1 and another covering a WP on vehicle block 60, it will require the user to scroll a long way to be able to inspect them both against each other. Therefore the ability to visually group a shift will be implemented that will allow the user to collect and display the specific vehicle blocks that a selected shift covers. This will be done by opening a new window when a shifts WS is selected. In the new window the relevant visual vehicle blocks from the main window will be copied over and displayed in the new window. There will also be fields where the property values of the shift will appear, e.g. cost.

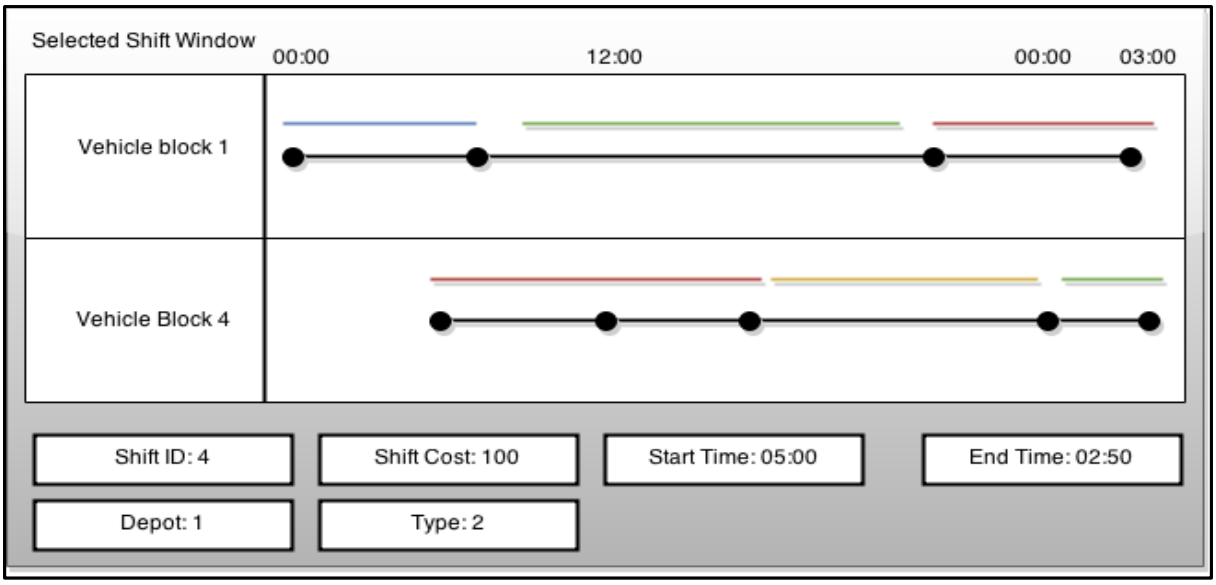


Figure 16 Design of interface for the user specified shift window.

Figure 16 displays the outcome of what will happen when the user selects the red shift in **Figure 15**. The shift covers vehicle blocks 1 and 4, which can be seen grouped together above. Property values of the shift are displayed in cells below.

3.4 Software Development Tools

3.4.1 Programming Language

Prior to the design process, C++ was decided to be the most suitable programming language for the implementation of the project based on the minimum requirements (discussed in section **2.5**). However, due to being unsuccessful in attempts to find certain functionalities within the Qt framework, C++ was discarded as a potential programming language.

The successful experimentation using WPF and C# that is explained in section **3.3.3.2** led to them becoming the most likely candidates to be chosen for the project. However, as is outlined in section **3.3.2**, to create a visual schedule, vehicle blocks will need to be iteratively created and stacked on top of each other. Vehicle blocks will be built up of multiple graphical components, therefore there needs to be a way of creating a reusable, graphical component that can be custom designed to contain other components. WPF's UserControl class provides exactly that (Microsoft, c2015, *UserControl Class*).

Therefore due to the discovered capabilities of WPF and its compatibility with C#, which is a programming language that has similar syntax to C++. C# with the WPF framework was the chosen programming language for the implementation of the project.

3.4.2 IDE

Integrated Development Environments offer facilities to developers with the purpose of aiding the software engineering process. Different IDEs provide different functionality, making it a developers choice to decide on the most appropriate for their project. Most supply compilers for their supported programming languages. Some offer visual designers that allow developers to view changes to their design as they modify the code and functionality that enables the developer to build an interface via dragging and dropping components. Due to the lack of prior experience with using WPF, a visual designer was highly sought after.

Microsoft's Visual Studio is an Integrated Development Tool that provides an environment for developing WPF applications using a visual designer. It also offers support for the C# programming language via a compiler and debugger (Microsoft, c2015, *Visual Studio*). Due to not being able to find another IDE that provides a visual designer for WPF, along with being generally impressed with the layout of Visual Studio, it was chosen for the project.

3.4.3 Version Control

Version control is a security measure that enables developers to save increments of their work to a remote machine, giving them access to each save when faced with the following situations:

- The developers work that is saved to their own machine becomes damaged or lost
- The developer 'breaks' the software they have saved to their own machine, so they need to revert back to a previous version where the software was stable

Git version control is readily integrated into the Visual Studio environment, enabling access to the committed work via a developers online Visual Studio account. Due to the ease of use of the integrated version control, Git was chosen.

4. Implementation

This section describes the implementation of the design split into iterations, detailing:

- How the functionalities defined in the design were created within the software application
- Where and why the work deviated away from the initial designs
- When modifications were made to features that had already been created
- Testing and evaluation performed at the end of each iteration
- What extension work was performed

4.1 Iteration 1

4.1.1 Creating and Populating the Project Folder

The first stage of the implementation was to create a project folder that was suitable for development of a WPF application. Visual Studio offers the option to create a readymade WPF project folder that contains all the necessary property and library files. The folder also contains an empty *MainWindow.xaml* file and its corresponding interaction logic file, *MainWindow.xaml.cs*. *MainWindow* is set as the first object to be initialised when the application is executed. The project folder was then populated with empty class files that were named based on the structure designed in section **3.2**.

4.1.2 Initial Data Handling

As stated in **2.4.1**, the GreedyHeuristics program can be used as a starter template for handling the data contained within the Trains and Shifts files. Therefore the development performed in this iteration mainly consisted of manually porting specific parts of the Python program over to C#. Modifications were made where thought necessary, some of which have already been defined in the design section, to make the handled data more suitable to the design of this solution. A detailed explanation is also given due to the importance of having a good understanding of how the data is handled. The set of data used to test the data handling process was a small sample set called DS1.

DS1	
Relief Opportunities	25
Work Pieces	22
Vehicle Blocks	3
Candidate Shifts	26

Figure 17 Table displaying number of train crew schedule components within DS1

4.1.2.1 Reading in the data files

4.1.2.1.1 *Trains* Class

The *Trains* class contains a *static* method called *ReadFromFile()* that takes a single string parameter argument and returns a *Trains* object. The string is called *fileName* and is used to initialised a new *StreamReader* object. The *StreamReader* object creates a stream from the file specified by the string value and reads all the values into a list of integers. This is achieved by the following process:

1. Splitting the entire file wherever there is a single piece of whitespace
2. Remove all the empty entries created from the previous step
3. Store all of the left over values to a list of strings
4. Select every value within the list of strings and convert them into 32-bit signed integers
5. Store the integer values to a list of integers

The integer values, within the list, are stored into local properties by following the format of the Trains file (outlined in section **2.3.1**). The *ReadFromFile()* method returns a call to the constructor of the *Trains* class file, passing the local variables as parameters. The constructor assigns the parameters to the corresponding property of the *Trains* class. Hence initialising an object that contains the relevant raw data values from the specified .trains file.

4.1.2.1.2 *Shifts* Class

The *Shifts* class is similar in structure to the *Trains* class described above. It has a static method called *ReadFromFile()* that takes the .shifts file name as a parameter, however it also takes an integer value, *numberOfShifts*, that has come from the .trains file. Shift files follow the format in which each line describes a separate shift, therefore reading in a .shifts file is achieved by the following process:

1. *StreamReader* object is initialised using the file name

2. For loop iterates over the range of the *numberOfShifts* value
3. *StreamReaders ReadLine()* method is used to read one line of the file per every iteration
4. The line of data is split wherever there is a single piece of whitespace
5. All empty entries created in previous step are removed
6. Values left are stored into a list of strings
7. Each string value is converted to an integer and is stored into a list of integers
8. List of integers is added to a list of list of integers

The *ReadFromFile()* method then returns a call to the constructor of the *Shifts* class file, passing the list of list of integers as the only parameter. The constructor then copies the parameter to the only property of the *Shifts* class, *allShifts*. Hence initialising an object that contains the raw data for every shift available for the schedule.

4.1.2.1.3 *RawData* Class

The *RawData* class has two uninitialized properties, a *Trains* object and a *Shifts* object. Its constructor takes two strings as input, *trainsFilePath* and *shiftsFilePath*, and initialises both the Trains and Shifts object properties by passing the strings as parameters respectively. It also passes the value that represents the number of shifts from the .trains file to the *Shifts* constructor. The *RawData* object is initialised within the *MainWindow*. At this point, the file names for the data sets are hardcoded into the initialisation of the RawData object for ease of use whilst developing.

4.1.2.2 Initialising *ReliefOpp, WorkPiece* and *Shift* Objects

The next step in the data handling process was to organise the stored trains and shifts data into objects that represent the components of a train crew schedule. This involves creating an object for all of the ROs, WPs and shifts that are detailed in the data sets and grouping them into three lists; *AllReliefOpps, AllWorkPieces* and *AllShifts*. The lists made the objects easy to transport and access throughout the application.

According to the design, this entire process was to be contained in the constructor of the *PrepareData* class, which would take a *RawData* object as its single parameter and use the stored raw data to create and assign values to each object. However, it became apparent while implementing the design that it would be far more practical to split the process into three different class files. One file designated to creating and collecting all *ReliefOpp* objects called *ReliefOpportunities*, another to creating and collecting all *WorkPiece* objects called *WorkPieces* and the final to creating and collecting all *Shift* objects called *CandidateShifts*. These new class types would be initialised within *PrepareData* where the relevant properties of the *RawData* object would be passed as parameters. The objects were stored as

properties of the *PrepareData* class which kept them collected and easy to access throughout the rest of the application.

The rationale for splitting the process into three was that it was clear that the code was becoming hard to follow, which would make it difficult to find and edit if needed later on in the project. Splitting the process made it easier to find the desired code that might need to be changed.

4.1.2.2.1 *ReliefOpportunities* Class

The purpose of this class was to initialise and populate *ReliefOpp* objects using values from the stored data within *RawData* and then then collect them all into the class's list of *ReliefOpp* objects property called *AllReliefOpps*. The process that achieved this was as follows:

For loop iterates over the number of ROs in the data set

1. *ReliefOpp* objects are initialised within each iteration of the loop. Five parameter arguments are passed through containing the data for the following properties of a RO; ID number, arrival time, departure time, train unit ID and the location ID. These values are retrieved from lists using the count value of the for loop
2. An inner for loop iterates over the number of vehicle blocks
3. An if statement checks if the *ReliefOpp* objects ID value is less than or equal to the last RO on each vehicle block. If it is, then the *ReliefOpp* object is assigned the count value of the inner for loop and the for loop breaks. This value is the vehicle block that the RO is contained within
4. The *RO* is then appended to *AllReliefOpps* and the loop moves onto the next iteration

4.1.2.2.2 *WorkPieces* Class

The *WorkPieces* constructor takes multiple parameter arguments from the *RawData* object as well as the list of all *ReliefOpp* objects, *AllReliefOpps*. A new *WorkPiece* object is initialised for every pair of RO*s* next to each other within a vehicle block. This is achieved by:

1. A for loop that iterates over every vehicle block
2. An inner for loop that iterates over every RO within the vehicle block except for the last. This is due to the last RO of every vehicle block not having a corresponding WP because it is the end of a vehicle block
3. *WorkPiece* objects are initialised within the inner loop. This follows the rule that the number of WPs in a vehicle block is always equal to the number of ROs in a vehicle block minus 1 (outlined in section **2.2.1**)

The *WorkPiece* class files' constructor takes three parameter arguments; the ID of the *WorkPiece* and the start and end *ReliefOpp* objects of the *WorkPiece*. The start and finish times of a WP are defined by the departure time of the start RO and the arrival time of the end RO respectively. The *duration* property is calculated by subtracting the start time from the finish time. The start and end *ReliefOpp* objects are also stored as properties for each *WorkPiece* object.

When each *WorkPiece* is initialised within the inner for loop, the start *ReliefOpp* object has its *AddWorkPiece()* method called. It takes the *WorkPiece* object as a parameter and stores it into the *workpieceStarted* property of the *ReliefOpp* object.

WorkPieces has one property, *AllWorkPieces*, which is populated with every *WorkPiece* object created within the constructor.

4.1.2.2.3 *CandidateShifts* Class

The purpose of this class is to create and populate *Shift* objects from the data file values that describe the shifts in the data set and then collects them all into *AllShifts*. This class type takes the 2 dimensional list of integers that was initialised in the *Shifts* object and contains each line of raw data from the .shifts file. This is achieved by:

1. A for loop that iterates over the number of shifts in the data set, hence iterating over the 2 dimensional list of integers
2. A list of integers is initialised and has a list from the 2D list assigned to it that contains all the data values for a specific shift
3. A new *Shift* object is initialised with certain values from the list of integers being passed as parameters to the constructor. These values are; the shifts ID, the number of WSs contained within the shift, the cost of the shift and the shifts depot
4. An inner for loop iterates over the number of WSs in the shift
5. The identification values of the start and end RO of each WS are retrieved from the list of integers and used to access the corresponding *ReliefOpp* objects from *AllReliefOpps*, which are then added to a list of *ReliefOpp* objects respectively. This list is then added to a list of list of *ReliefOpp* objects which is a property of a *Shift* object. The 2 dimensional list, in simpler terms, details when each WS of a shift starts and finishes
6. The identification values used in the previous step are used again to create an inner-inner loop that iterates over the ROs that the WS covers, except for the last
7. Each *Shift* has a *workPieceInstancesCovered* property that is a list of *WorkPiece* objects. The *WorkPiece* object contained in each *ReliefOpp* object that is iterated over within the inner-inner for loop is then added to the *workPieceInstancesCovered* list. The *Shift* is then added to the *WorkPiece*

objects *choiceShifts* property, a list of *Shift* objects that represents the shifts that provide cover for the WP

8. The *Shift* object is then added to *AllShifts*

4.1.2.3 *Schedule* Class

The Schedule class contains the train crew schedule in the form of a list of *Shift* objects. The *Schedule* object is initialised in the main window by calling a method from the *PrepareData* class named *CreateScheduleMaxCovGain()*. Within this method is an algorithm used to generate a list of *Shift* objects so that every WP is covered by at least one shift, whilst also attempting to minimise the amount of overcovering.

As is stated in **2.4.1**, one of the algorithms within the GreedyHeuristics program should be used within the project solution simply as a tool for generating an initial list of *Shift* objects that can be displayed in the visual train crew schedule. Due to the limited timeframe of the project, no time was spent trying to understand the steps of the algorithm as it was being ported into the solution as it was not of relevance to the projects aim. However, it was noted that for every *Shift* object that was assigned to a *WorkPiece* object, the *WorkPiece* objects *coverCount* property was increased by 1.

Once the algorithm has completed its cycle, the method returns the *Schedule* object with its populated list of *Shift* objects property.

4.1.3 Iteration 1 Evaluation

Tests were performed throughout the implementation of handling the train crew scheduling data to ensure that it was being read in and stored as expected. The method for testing was to print properties of the objects to the debug console, and then compare the results to the data files. An example of the code used to achieve this can be seen in **Figure 18**.

```
public void PrintString()
{
    RO_vehicle.ForEach(x => System.Diagnostics.Debug.WriteLine(x.ToString()));
}
```

Figure 18 Snippet of code used to print a property of the Trains class: RO_vehicle.

4.2 Iteration 2

4.2.1 Implementing the schedule design

As was outlined in the initial design (discussed in section **3.4.1**), WPFs *UserControl* class was used to create a graphical *VehicleBlock* class type. The *UserControl* class has a *content* property that defines the *ContentControl* of the *UserControl*. This effectively means the container that is used to define the layout for the *UserControl*. It was important that the container that the visual vehicle blocks components were drawn within had a constant width

so that time could be measured consistently. Therefore a *Grid* class was used to structure the layout of the *VehicleBlock UserControl* to allow the developer to create rows and columns and define their height and width respectively. The grid had three columns and two rows; the first column displayed the vehicle block id value and the second contained the visual vehicle block, within which were the *ListBox* classes that were to draw the *ReliefOpp, WorkPiece* and *Shift* objects. The third columns sole purpose is to add padding to that the visual schedule is never overlapped by the scroll bar.

4.2.1.1 Initialising and stacking Vehicle Blocks

To visually form the entire schedule, the *VehicleBlock* objects had to be directly stacked on top of each other so that their horizontal positions were the same. WPF has a *Container Control* called *StackPanel*, which can be used to vertically place controls on top of, or below each other (Microsoft, c2015, *StackPanel Class*). A *StackPanel*, called *scheduleStack*, is contained within *MainWindow* and is populated with each *VehicleBlock* object. The children of a *StackPanel* are displayed automatically when the application is executed. The *StackPanel* was contained within a *ScrollViewer* object that enables scrolling through *StackPanel* containers content.

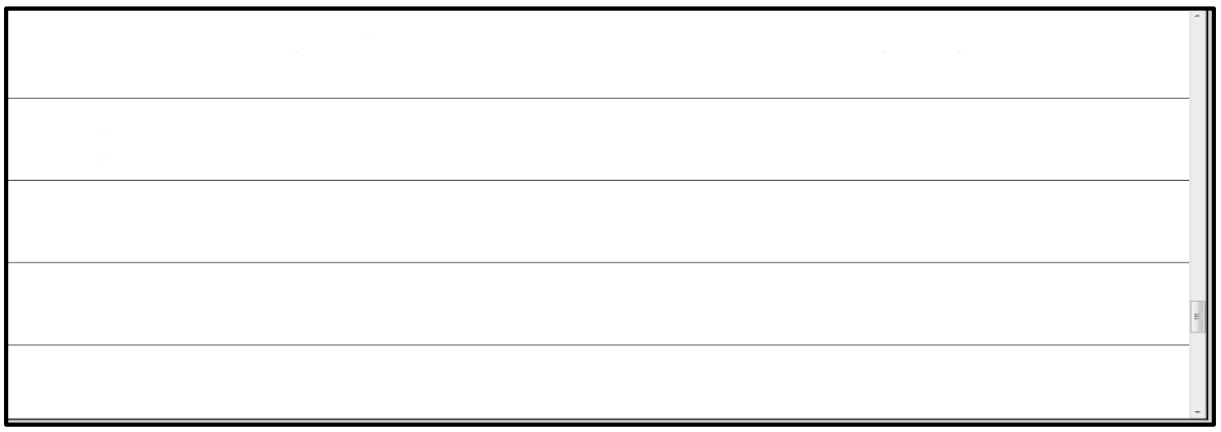


Figure 19 Executed application displaying stacked *VehicleBlock* objects and scroll bar

4.2.1.2 Converting Time Property Values

In order for *ReliefOpp*, *WorkPiece* and *Shift* objects to be positioned and sized correctly within each *VehicleBlock*, their property values that represent the time at which they occur needed to be scaled to fit the width of the vehicle blocks container. This was achieved using WPFs *IValueConverter*, which offers developers a method of manipulating a binding value (Microsoft, c2015, *IValueConverter Interface*). Two converter class files, *TimeValueConverter* and *DurationConverter*, were created and both given two static

properties called *EarliestTime* and *LatestTime*. The *Trains* class file was modified so it stored the earliest time in the schedule rounded down to the nearest hour (in minutes) and the latest time rounded up to the nearest hour (in minutes). Both these values were then assigned to the static properties *EarliestTime* and *LatestTime* respectively. Using these values, the two files then performed the following conversions:

- ***TimeValueConverter –*** Takes the bound time value and the width value of the vehicle blocks container as input parameters and calculates the horizontal position value using the following formula:

  ((*EarliestTime* – bound time value) / (*EarliestTime* – *LatestTime*)) * width of container = Horizontal value

- ***DurationConverter –*** Takes the bound duration value of each object and the width value of the vehicle blocks container as input parameters and calculates the width value for the objects visual rectangle using the following formula:

  (bound duration value * (width of container / (*LatestTime* – *EarliestTime*))) = Width value

The converter files were added to each relevant data binding via its *Converter* property so that each bound value is passed to the converter which then calculates and returns the converted value.

4.2.1.3 Displaying ROs and WPs

The cell defined by the second column and row of each *VehicleBlock* object's *Grid* container initially contained two *ListBox* classes that overlapped each other. One was bound to a list of *ReliefOpp* objects and the other to a list of *WorkPiece* objects. Although the ellipses and rectangles that were drawn to visually represent a RO and WP were positioned and sized correctly according to the bound data values that come from the corresponding *ReliefOpp* and *WorkPiece* objects, it was impossible to select the items of the *ListBox* that was overlapped by the other. The initial solution to this problem that was considered was to add a button to the main window that would allow the user to switch between being able to select WPs or ROs, however this was disregarded as it would overcomplicate the interface.

After referring back to the property values for the *ReliefOpp* class it was decided that, by making use of the *workpieceStarted* property, the *ListBox* that drew the WPs would be removed completely. Then the *ItemTemplate* property value of the *ListBox* that draws ROs, called *reliefOppListBox*, was customised to draw an ellipse for every *ReliefOpp* object and a rectangle for every *ReliefOpp* object for which the *workpieceStarted* property was not null. This was done by bounding the *WorkPiece* object's (that is contained within the *workpieceStarted* property) duration value to the width property of the rectangle and having it passed through the duration converter file.

```
<ListBox.ItemTemplate>
    <DataTemplate>
        <!--Grid Container to structure the controls within the ItemTemplate-->
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition Height="12"/>
                <RowDefinition Height="Auto"/>
            </Grid.RowDefinitions>
            <Canvas Grid.Row="0" Height="12">
                <Ellipse Fill="Black" Width="10" Height="17" Canvas.Left="-5"/>
                <!--Rectangle that visually displays startedWorkPiece using duration as binding-->
                <Rectangle Canvas.Bottom="1" Name="workPieceLine" Fill="Black" Height="7" Width="{Binding workPieceStartedDuration, Converter={StaticResource durationConverter},
            ConverterParameter={StaticResource width}}"/>
            </Canvas>
        </Grid>
    </DataTemplate>
</ListBox.ItemTemplate>
```

Figure 20 Code that defines the ItemTemplate property for reliefOppListBox

The process of passing the correct collection of *ReliefOpp* objects from the list of all *ReliefOpp* objects to the *ItemsSource* of each *VehicleBlock's reliefOppListBox* was performed within the same for loop that the *VehicleBlock* objects were initialised within. Collecting the *ReliefOpp* objects appropriately was achieved as follows:

1. An integer variable called *skipValue* was initialised to 0. This value was used to store the amount of *ReliefOpp* objects that had been passed into the previous *VehicleBlock* objects *reliefOppListBox*'s *ItemsSource* property
2. A for loop iterates over the number of vehicle blocks using a counter, *i,* which represented the vehicle block that was being defined from the data
    a. *skipValue* was used to skip along the indexes of the list of all *ReliefOpp* objects to the point at which they had been passed from into the last *ItemsSource* (*skipValue* = 0 for first iteration)
    b. The integer value returned from

        (*lastReliefOpportunityOnVeh[i]* – *skipValue*)

        was used to take that many *ReliefOpp* objects, starting from the index that was reached in step a and pass it through to the *ItemsSource*

    c. *skipValue* was then updated with the value of *lastReliefOpportunityOnVeh[i]* and the loop started again

** *lastReliefOpportunityOnVeh* – List of integers that represent the ID value of the last RO of each vehicle block within the data files.

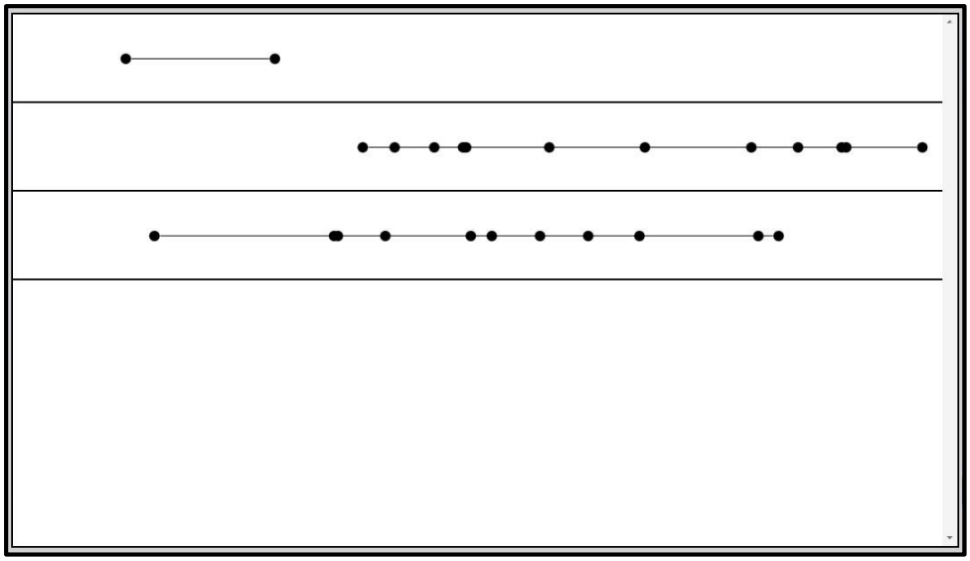


Figure 21 Screen cap of application displaying ROs and WPs components of each vehicle block from the DS1 data set.

4.2.1.4 Displaying Shifts

The cell defined by the first row and second column of the *VehicleBlock* class's *Grid* container is where the visual shifts are to be displayed. The initial approach was to:

1. Create a single *ListBox*
2. Bound its *ItemsSource* to a collection of *Shift* objects that would be a subset of the *Schedule* objects list of *Shift* objects
3. Use the properties of each *Shift* object to define the width and X axis position of the visual representation along with the converter files

Whilst implementing this approach it became apparent that it would not work due to a single *Shift* object covering potentially multiple vehicle blocks. This is because a shift is split into WSs. The solution to this was to implement a new class type called *WorkSpell* that would make it possible to 'split' *Shift* objects into visually presentable objects. For the purposes of saving memory, *WorkSpell* objects are only created for *Shift* objects that are present in the *Schedule* objects list of shifts, not for every *Shift* object stored in the running applications data. Therefore the initialisation of all *WorkSpell* objects takes place in a method of the *Schedule* class called *CreateWorkSpellList()* which then collects all the objects into a list of *WorkSpell* objects that is a property of the *Schedule* class. The initial property values of the *WS* are as follows:

- ***id* –** Defines the identification value of the *WorkSpell* object
- ***startReliefOppId* –** Integer value representing the identification value of the RO that the WS starts at in the schedule
- ***vehicleBlock* –** Integer value representing the vehicle block that the WS covers

- ***startTime –*** Integer value representing the start time of the WS in minutes
- ***endTime –*** Integer value representing the end time of the WS in minutes
- ***workSpellCost –*** Double value representing the cost of the WS in terms of minutes. This also represents the duration of the WS
- ***parentShift –*** **A** *Shift* object that represents the shift that the WS is a part of in the data set

The *CreateWorkSpellList()* method is called once the crew scheduling algorithm has finished populating the *Schedule* objects list of *Shift* objects property. The process used to initialise and populate *WorkSpell* objects is as follows:

1. An outer for loop iterates over the number of *Shift* objects contained in the *Schedule* objects list of *Shift* objects property, which is retrieved using the *.Count()* method
2. Count value *i* from the outer loop is used to access each *Shift* from the *Schedule* objects list of *Shift* objects to retrieve the *numberOfWorkSpellsInShift* integer property value
3. An inner for loop then iterates over the *numberOfWorkSpellsInShift* value
4. *WorkSpell* objects are initialised within the inner loop. The values passed as parameters to the constructor are: ***id, startReliefOppId, vehicleBlock, endTime, workSpellCost and parentShift***
5. Each *WorkSpell* object is then added to a list

To be able to pass collections of *WorkSpell* objects into a *ListBox* using a similar method to the one used for *ReliefOpp* objects, the list of all *WorkSpell* objects was ordered based on the value of their ***startReliefOppId*** property. This means that the list of integers, *lastReliefOpportunityOnVeh* (defined in **4.2.1.3**), can be used to collect the *WorkSpell* objects into the vehicle blocks they cover. The problem with this process was that no account was taken of the possibility of overlapping WSs in a crew schedule. When the program was run, the *WorkSpell* objects were correctly collected into the *ListBox's* of each *VehicleBlock* object and the horizontal positioning and width of the visual WSs was also correct. However, some of the visual WSs could not be seen or selected due to them being overlapped by other WSs that also cover the same vehicle block for at least some of the same time. What was required was to somehow modify the vertical position of were a WS is placed within a *VehicleBlock* object so that it would not visually overlap with any other WS.

A three dimensional (3D) list of *WorkSpell* objects was initialised. The outer list represented the vehicle blocks in the schedule, the list of lists represented the stack that overlapping WSs are split between by putting them into lists of *WorkSpell* objects. **Figure 22** below was created to help visualise the 3D list. The process of splitting all the *WorkSpell* objects into

the 3D list was achieved using the following process which starts from when the *WorkSpell* objects have been order by their ***startReliefOppId*** propert value:

1. All the *WS* objects were collected into a list of lists from the sorted list. The method used to achieve this was similar to the method that organised the *ReliefOpp* objects into their respective vehicle blocks by using a *skipValue* (discussed in section **4.2.1.3**). The outer list represented the vehicle block that each list of *WorkSpell* objects was collected into
2. A for loop iterates over the lists of *WorkSpell* objects, called *workSpells,* that define which vehicle block each WS is contained within
3. A 2 dimensional list of *WorkSpell* objects called *stackedRows* is initialised as empty. *stackedRows* can also be visualised from observing **Figure 22**. Its outer list will represents the rows stacked on top of each other and its inner list will contain the *WorkSpell* objects contained within each row
4. An inner for loop iterates over every *WorkSpell* object within *workSpells*
5. The *stackedRows* 2D list cycles through each *WorkSpell* object contained in each of its rows and compares them to the current *WorkSpell* object to see if their time value properties overlap (Code that compares time values can be seen in **Figure 23**). If the current *WorkSpell* does not overlap with any other *WorkSpell* object in a row of *stackedRows*, then it is added to that row. If the current *WorkSpell* object overlaps with another *WorkSpell* object in all of the existing rows in *stackedRows*, then a new row is created and added to *stackedRows* and the current *WorkSpell* is added to the new empty row
6. Once all the *WorkSpell* objects within *workSpells* are exhausted, the outermost for loop moves onto the new vehicle block and *stackedRows* 2D list is added to the 3D list which is a property of the *Schedule* class type

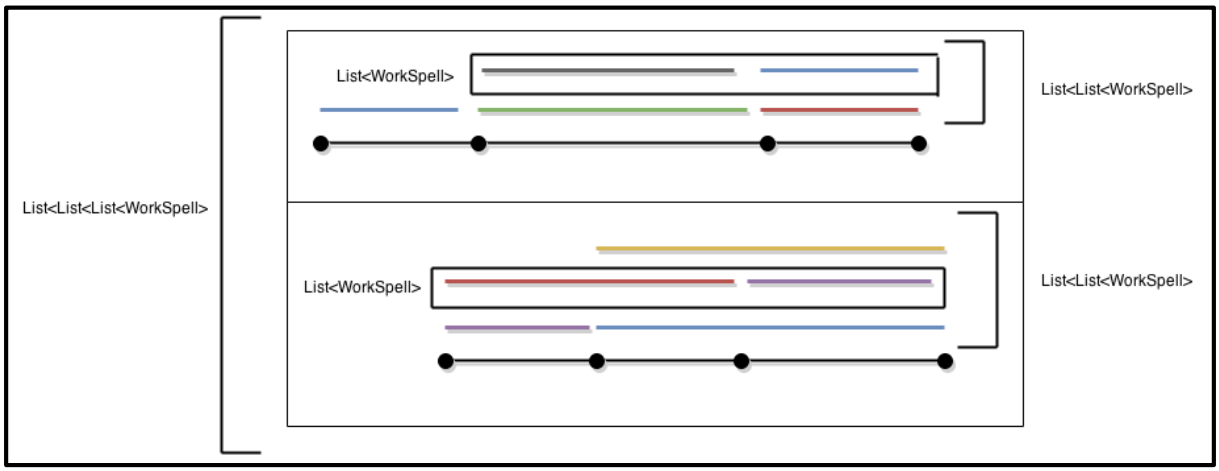


Figure 22 Diagram created to help visualise the three dimensional list of WSs

```
// compares this workspell to another to see if they overlap. True is returned if the Workspells overlap
public bool OverlapsWith(WorkSpell other)
{
    return this.startTime < other.endTime && other.startTime < this.endTime;
}
```

Figure 23 Snippet of code that compares two *WS* objects to see if they overlap. True value is returned if they do

A new *UserControl* class called *WorkSpellRow* was created with its *content* property being a *ListBox* that's *ItemPanel* is a *Canvas*. The purpose of this *WorkSpellRow* is so that a *ListBox* can be iteratively created for every row of *WorkSpell* objects, defined by the 3D list, and then stacked on top of each other to create vertical separation in the visual display. This is done by adding each *WorkSpellRow* object to a *StackPanel* that is contained within the *Grid* of the *VehicleBlock* class at cell row 1, column 2. The height of the *VehicleBlock UserControl* is set to be automatically created, based on the height of the components it contains. This allows for, in theory, an infinite number of *WorkSpellRow* objects to be added before the WSs stop being visually displayed.

To be able to distinguish which WS belongs to which shift in the visual representation, the *Shift* class type is given a *Color* property value that is assigned using a random number generator. Each *WorkSpell* object inherits a colour value from its parent *Shift*, however it has to be converted from a *Color* to an instance of a *SolidColorBrush* class so that it can be bound to a *Rectangle* object's *Fill* property.

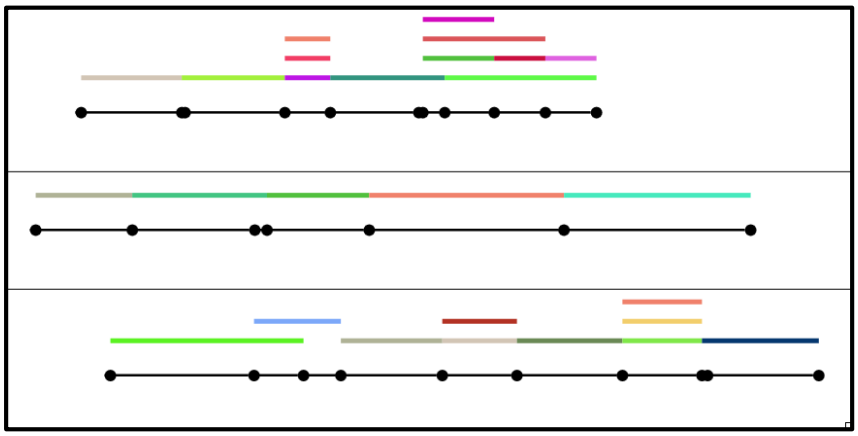


Figure 24 Screen cap of application being executed. Overlapping WSs are stacked above each other.

4.2.2 Iteration 2 Tests

The tests performed throughout iteration 2 involved using a larger data set, DS2 that was supplied by Raymond Kwan, to make sure the application could sensibly handle realistically sized train crew scheduling data sets.

Comparing Data Sets		
	DS1	**DS2**
Relief Opportunities	25	307
Work Pieces	22	260
Vehicle Blocks	3	47
Candidate Shifts	26	73517

Figure 25 Table comparing size of DS1 and DS2 data sets

One problem created from this iteration was that due to the colours being assigned randomly, it is possible for two shifts to have the same colour value which could cause confusion for the user when trying to distinguish shifts from within the visual display.

4.3 Iteration 3

4.3.1 Displaying data for selected *ReliefOpp* and *WorkPiece* objects

The process of displaying the data for a user specified *ReliefOpp* or *WorkPiece* object (discussed in section **3.3.3**) was split into two stages:

1. Retrieving and storing the *ReliefOpp* and *WorkPiece* objects that correspond to the visual component selected by the user
2. Creating a new window that will display the data values of the retrieved *ReliefOpp* object and its corresponding *WorkPiece* object if one exists

4.3.1.1 Retrieving the selected *ReliefOpp* and *WorkPiece* object

The *ListBox* class has a property called *SelectedItem* that retrieves the users currently selected item. An event called *SelectionChanged* is triggered when a new item of the *ListBox* is selected by the user (Microsoft, c2015, *Selector.SelectionChanged Event*).

A method within the *VehicleBlock UserControl* class, called *OpenReliefOppDialog()*, handles the *SelectionChanged* event for the *reliefOppListBox*, within which a new *ReliefOpp* object is initialised and assigned the value of the *SelectedItem* property. A *WorkPiece* object is initialised and assigned its value from the retrieved *ReliefOpp* object's *workpieceStarted* property if it does not equal *null*. Hence retrieving and storing the *ReliefOpp* and *WorkPiece* objects that correspond to the visual component selected by the user.

4.3.1.2 Window Displaying Retrieved *RO* Data

The purpose of the window class, called *ReliefOppDisplay*, is to provide a quick way of viewing the specific data values of RO and WP components within the train crew schedule.

The window is small and displays the data values in a simple, compact grid format so to not take attention away from the main visual schedule. The window is initialised within the *OpenReliefOppDialog()* method, discussed in the previous section, and is passed the *ReliefOpp* and *WorkPiece* objects that have already been initialised and assigned values to. The constructor then assigns the data values of the *ReliefOpp* object and *WorkPiece* object(if one exists) to the appropriate label's *content* property so that it can be displayed in the window. If the *WorkPiece* has the value *null* then the corresponding labels display "N/A". Hence creating a new window that displays the data values of the retrieved *ReliefOpp* object and its corresponding *WorkPiece* object if one exists.

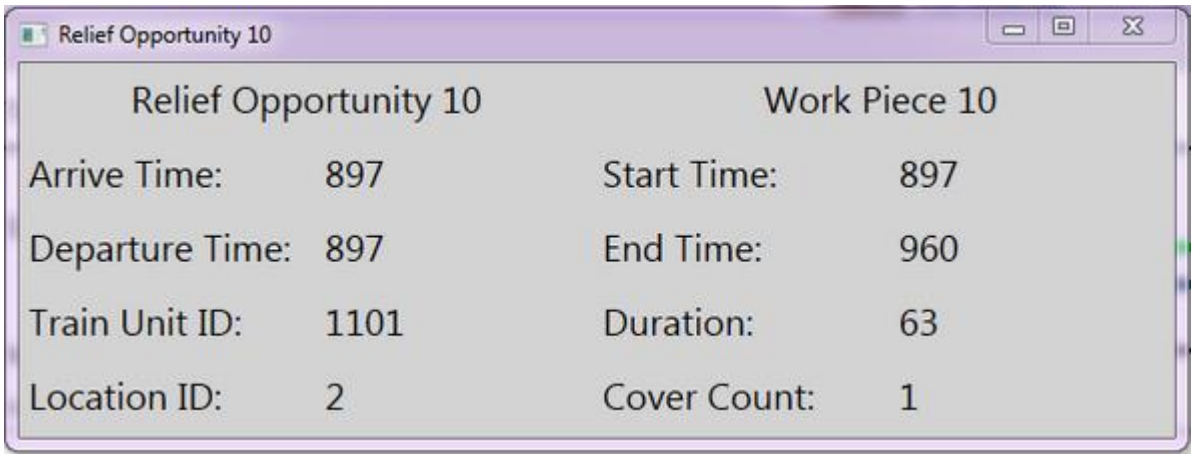


Figure 26 Screen cap of *ReliefOppDisplay* window displaying the data for a user specified RO and WP

4.3.2 Iteration 3 Progress Meeting Feedback

During the 7th week of the projects timeline a progress meeting was held with supervisor Dr Raymond Kwan and assessor Professor Kristina Vuskovic with the purpose of giving feedback on the work performed so far. The feedback given centred around the difficulty of understanding the visual representation of the train crew schedule, i.e. its appearance was to abstract.

4.3.2.1 Resultant Modifications

Modifications were made to the visual schedule in an attempt to make it easier to understand, these were as follows:

- A visual key was added to the main window that showed what each shape represents in the train crew schedule
- Labels were added to the visual vehicle block to give components more meaning:
    - A label below each RO containing its location value

- Two labels on the far left side of the *VehicleBlock UserControl* that gives guidance to the user as to where the shifts are being displayed and that the value under each RO is its location id

The visual timeline was still due to be added, which would add further context to the visual schedule.

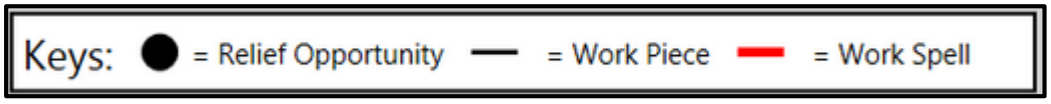


Figure 27 Visual key added to the main window

4.4 Iteration 4

4.4.1 Visual Timeline

To visually display the timeline of the schedule a *TimeLine UserControl* is created. Its *ContentControl* is a *Grid* that defines three columns, each the exact same width as the columns of the *VehicleBlock UserControl's*. This is so that when the *TimeLine* object is added to the *scheduleStack* it will perfectly aligned with each *VehicleBlock* object added after it. Within the second column of *TimeLine's Grid* is a *Canvas* container called *timeLineCanvas*. When the application is executed and the visual display is being created, the *timeLineCanvas* is populated with labels positioned at every hour mark within the visual schedule. The content of each label is the hour of the day that it represents. The process of creating and populating the labels is as follows:

1. The earliest and latest hours (these values are in terms of minutes) of the train crew schedule are taken as parameter arguments by the *TimeLine's* constructor and stored as variables *earliestTime* and *latestTime* respectively
2. The total number of hours between the earliest and latest hour is calculated and stored using the following formula:

   (*latestTime* – *earliestTime*) / 60 = Number of hours

3. A for loop iterates over the number of hours using counter value *i*
4. A *label* is initialised within each iteration called *timeLabel*
5. The time value that *timeLabel* will represent is calculated using the following formula:

   earliestTime + (60 * i) = *time*

6. A *TimeSpan* object is used to convert *time* from minutes to hours and minutes in 'hh:mm' format which is then assigned to the *content* property of *timeLabel*
7. The horizontal position value of *timeLabel*, named *bindingTime*, is calculated using the following formula:

((*earliestTime* – *time*) / (*earliestTime* – *latestTime*) * *width*) = *bindingTime*

***width* value is the constant value used for the width of the entire visual schedule

8. *timeLabel* is added to *timeLineCanvas* and positioned using *bindingTime*'s value

TimeLine is added to the main windows *StackPanel* (*scheduleStack*) before the *VehicleBlock* objects are so that it appears at the top of the visual schedule.

4.4.2 Specified Shift Display Window

The process of implementing the design that is outlined in section **3.3.5** is split into three stages:

1. Retrieving the *Shift* object that corresponds to the WS selected by the user
2. Collecting the *VehicleBlock* objects that the retrieved shift covers
3. Displaying the collection of *VehicleBlock* objects to the user

4.4.2.1 Retrieving the *Shift* object

1. The process within the *WorkSpellRow UserControl* that is used to store the selected *WorkSpell* object is identical to the method outlined in section **4.3.1.1**, by making use of a *ListBox's SelectedItem* property and *SelectionChanged* event
2. A *Shift* object, called *SelectedShift*, is initialised and assigned the retrieved *WorkSpell* objects *parentShift* property value
3. A *delegate* type is declared within the *MainWindow* class that enables its *DisplayShiftMethod()* method to be invoked from within the *WorkSpellRow UserControl*
4. The method is invoked and the *SelectedShift* property is passed through as a parameter to the main window where it is stored as a property of the *MainWindow* object called *SelectedShift*. Hence the *Shift* object that corresponds to the WS selected by the user is retrieved and stored

4.4.2.2 Collecting *VehicleBlock* objects

When each instance of a *VehicleBlock UserControl* is initialised in the main window (discussed in **4.2.1.1**) it is stored into a list called *allVehicleBlocks*. Using *SelectedShift's vehicleBlocksCoveredByShift* property value (a list of integers representing the vehicle blocks that the shift covers in the data) *VehicleBlock* objects are retrieved from *allVehicleBlocks* and collected into a list called *shiftVehicleBlocks*. Hence collecting the *VehicleBlock* objects that the retrieved *Shift* covers.

4.4.2.3 Displaying Collection of *VehicleBlock* objects

The initial design plan to display the vehicle blocks that had been collected into *shiftVehicleBlocks* was to; open a new window that contains a *Stackpanel* and add each

VehicleBlock item from *shiftVehicleBlocks* to it. However, this did not work due to it being impossible to visually display an instance of a *UserControl* twice at the same time. Each *VehicleBlock* item within *shiftVehicleBlocks* was already being displayed in the main window. Two possible solutions were considered:

- Create copies of each *VehicleBlock* object within *shiftVehicleBlocks* and add them to the new windows *StackPanel*. This is effectively the same solution created in the design process except the problem of not being able to display a single *UserControl* instance twice at the same time is side stepped. However, this method was discarded after it dramatically slowed the application due to the processing power required to copy every component of each *VehicleBlock* object
- Instead of a new window being created to display the vehicle blocks the *MainWindow's scheduleStack* property is cleared of all its children components and is filled with the *VehicleBlock* items within *shiftVehicleBlocks*. Changing a *StackPanel*'s contents automatically causes the user interface to update the display with the new contents. This solution was preferred for two reasons; no new memory had to be allocated to and it kept the visualisation of the schedule to within the same window

Modifications had to be made to the initial design of this functionality so that the chosen solution could be implemented into the application sensibly. The changes were:

- The *shiftVehicleBlocks* list had become redundant due to the *VehicleBlock* objects no longer being passed to a new window. It was removed, and the *VehicleBlock* objects that the *SelectedShift* object covers are collected directly into the *scheduleStack*
- The labels that were to display property values of the selected shift in the initial design (**Figure 16**) were moved into the main window
- When a visual WS is selected, the *Shift* object passed from the *WorkSpellRow UserControl* is stored into the main windows *SelectedShift*. The labels for the specific values of a shift are populated using the *SelectedShift* object. Therefore the process of visually collecting the *VehicleBlock* objects does not occur when a WS is selected by the user
- Instead, the main window has two buttons, *ViewShiftButton* and *ViewFullScheduleButton*. When the *SelectedShift* property equals *null*, both buttons are disabled. The buttons have the following functionally:
    - ***ViewShiftButton -*** When *SelectedShift* is assigned a *Shift* object, i.e. the user selects a WS, the *ViewShift* button is enabled and when clicked the *scheduleStack* is emptied of all its children and the *VehicleBlock* objects

that the *SelectedShift* covers are collected and added to it. Hence visually displaying the vehicle blocks covered by the user specified shift

- ***ViewFullScheduleButton –*** Is enabled when *ViewShiftButton* is clicked on. This button offers a way for the user to revert back to viewing the entire schedule. This is done by clearing the *scheduleStack* and refilling it with all the *VehicleBlock* items contained within *allVehicleBlocks*

4.4.3 Enabling the user to modify the schedule

Giving the user the option to manually modify the crew schedule enables them to maximise the cost efficiency of each train crew schedule where the scheduling algorithm used fails to do so. It also helps in the case where a schedule has already been finalised but at last minute a crew becomes unavailable, a train planner would be able to remove the crew from the schedule, see what pieces of work are not covered by a shift and add cover to each of them so that the train crew schedule is valid once again.

4.4.3.1 Deleting a Shift

A button was added to the main window called *DeleteShiftButton*. The button is disabled unless there is a value being stored into *SelectedShift*, i.e. the user has a shift selected. The buttons click event handler does the following process:

1. Calls a method on the *Schedule* object called *removeShift()* and passes the *SelectedShift* through as a parameter argument. The method then removes the *SelectedShift* from its *shiftList* property (The list that contains all the shifts that are in the train crew schedule)
2. Then a method is called on the *SelectedShift* object that reduces the cover count of each *WorkPiece* object that it is cross referenced with
3. Then the *CreateWorkSpellList()* method on the *Schedule* object is called. This method initialises the 3 dimensional *WorkSpell* container (discussed in section **4.2.1.4**) hence removing all *WorkSpell* objects from storage. It then creates new *WorkSpell* objects based on the updated *shiftList* and sorts them into the 3D container
4. Finally called is the method within the main window that initialises and populates the *VehicleBlock UserControl's* and then stacks them on top of each other to form the visual schedule. This effectively re-draws the schedule with the new *shiftList* that no longer contains the deleted *Shift* object

This enables the user to delete any shift in the schedule manually.

4.4.3.2 Adding a Shift

The initial functionality that allowed the user to add a shift from all the candidate shifts in the data that were not already in the schedule was as follows:

1. Open a new window, called *candidateShiftDialog*, from the main window when a button called, *ViewCandidateShiftsButton*, is clicked
2. List every *Shift* object stored within the application except for the *Shift* objects that are in the crew schedule in a *ListBox* container that uses a grid format to display each shifts ID, number of WSs, cost, type and depot properties
3. The user selects a shift and presses an “Add Shift To Schedule” button
4. The *Shift* object is passed to the *MainWindow* where it is added to the *Schedule* objects *shiftList*
5. The process of re-drawing the visual schedule is the same as described in section **4.4.3.1**, by re-creating all *WorkSpell* objects and all *VehicleBlock UserControls*

This method worked when using the sample data set DS1, however when the larger data set, DS2, was used the application would crash whilst trying to load the list of 70,000+ *Shift* objects into the *ListBox*. Therefore a new method had to be devised that would reduce the amount of *Shift* objects being passed into the list at once.

Each *WorkPiece* object had a property called *choiceShifts* which is a populated list of *Shift* objects that supply cover to the WP. The concept was to allow the user to add a shift to a specific WP. This would cut down the number of candidate shifts needed to be displayed to the user at once. It is also sensible because if the user is not looking to add a specific shift, but instead add cover to a specific WP then this solution would let them do that exactly that. The concept was implemented using the following process:

1. A button was added to the RO and WP property values window (Section **4.3.1**, **Figure 26)** called *addShiftButton* that is only enabled if the *WorkPiece* property of the window did not equal *null*
2. The event handler for when *addShiftButton* is clicked initialises a list of *Shift* objects called *shiftsCoveringSelectedWorkPiece* and populates it with the items from the stored *WorkPiece* objects *choiceShifts* property except for the *Shift* objects present in the *Schedule* objects *shiftList*
3. A *candidateShiftDialog* window is initialised and the *shiftsCoveringSelectedWorkPiece* list is passed as a parameter argument to the constructor of the window
4. The *shiftsCoveringSelectedWorkPiece* is passed through to the *candidateShiftDialog’s ListBox ItemsSource*, which visually displays each *Shift* in a vertical list. The list orders the shifts in terms of their ID value to aid the user in finding a specific shift
5. When the user selects a shift item and presses the *AddShiftButton* the corresponding *Shift* object is passed through to the main window using a *delegate* type method

6. The *Shift* object is added to the *Schedule* objects *Shiftlist* and the visual schedule is redrawn to display the new shift

While this implementation can still cause long wait times for the *ListBox* to be populated with *Shift* objects, it does not crash when using the larger data set, hence offering a way for users to manually add shifts to the schedule.

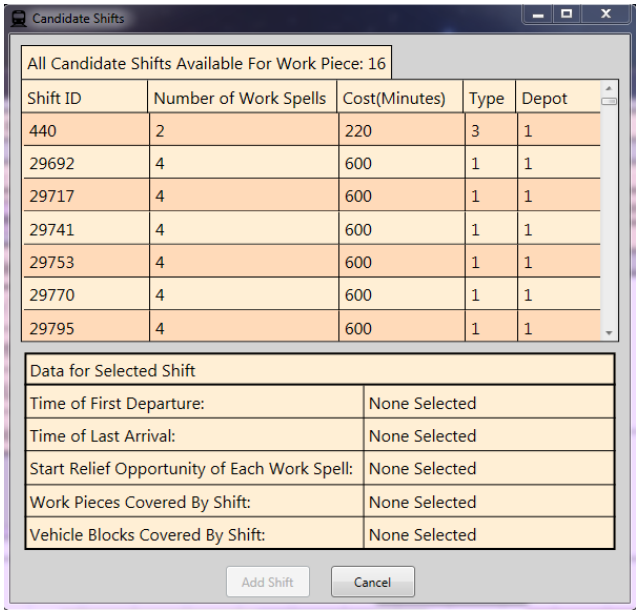


Figure 28 Screen cap of *candidateShiftDialog* window displaying a list of shifts that can be added to cover WP 16

4.4.3.3 Deleting the entire Crew Schedule

Originally not part of the design however this was added as functionality so that if the train planner wanted to build a train crew schedule from scratch, this would offer a short cut to deleting all the shifts in the current crew schedule. Deleting the crew schedule is achieved as follows:

1. The *Schedule* objects *shiftList* property is cleared of all of its items
2. The visual schedule is re-drawn with no *Shift* objects to draw

This process is instigated when the user presses a button called *DeleteCrewScheduleButton* within the main window.

4.4.3.4 Functionality to generate a new crew schedule

With the functionality of deleting the entire crew schedule being added it naturally makes sense to add functionality that allows the user to create a new train crew schedule. For example the user deletes the crew schedule so that they can analyse just the ROs and WPs of a data set and then wants to create a new train crew schedule once they have a better understanding of those components.

Two labels were added to the top of the main window, their content being; the total number of shifts in the current train crew schedule and the total cost of all shifts in the current train crew schedule. With these labels and the ability to repeatedly generate new train crew schedules it is possible to analyse the scheduling algorithm being used by averaging the total cost of the crew schedule that the algorithm produces. The train planner is also able to repeatedly generate train crew schedules to try to create the cheapest possible.

The process of generating new train crew schedules is as follows:

1. The process is instigated when the user clicks the *CreateCrewScheduleButton* button from within the main window
2. The *Schedule* objects *shiftList* property is cleared of all its items
3. The algorithm that generates a train crew schedule is called and populates *shiftList* with *Shift* objects
4. The *Schedule* objects *CreateWorkSpellList()* method is called that populates the 3 dimensional list with *WorkSpell* objects from the *Shift* objects within *shiftList*
5. The visual schedule is re-drawn using the newly populated 3 dimeonsional list of *WorkSpell* objects. Hence generating a new train crew schedule to be displayed

4.4.4 Iteration 4 User Evaluation

In order to ensure the quality of the solution a user evaluation was performed towards the end of iteration 4 whilst there was still time to implement improvements to the application. The purpose was to test the ease of understanding and navigating through the user interface. A University of Leeds MEng Computer Science student was chosen based on the criteria that from their taught studies they would have the knowledge to critically evaluated the software application. The user had no prior knowledge of train crew scheduling. The full user evaluation form with the written answers included can be found in **Appendix D**.

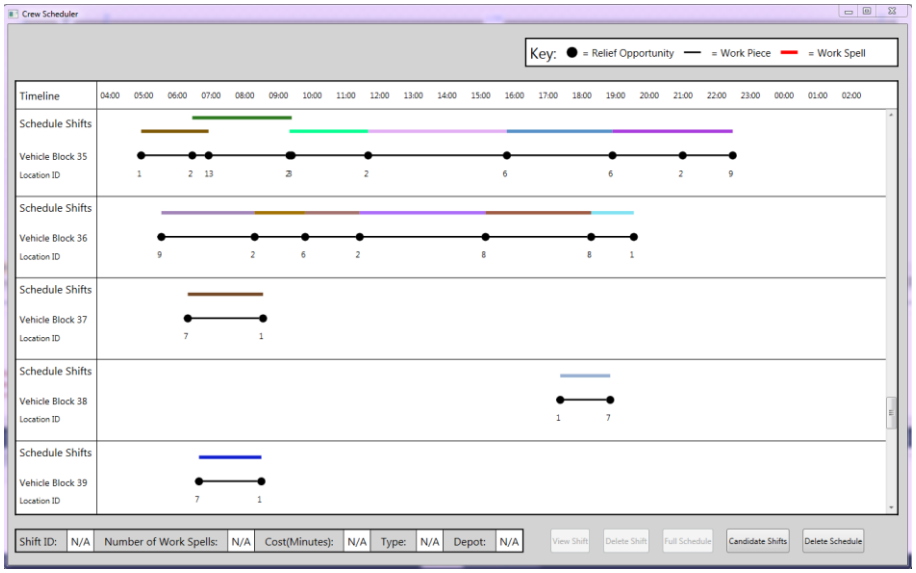


Figure 29 Screen crap of the applications main window whilst user evaluation took place

4.4.4.1 Structure

The user evaluation was split into three stages:

1. 5 minutes was spent by the user familiarising themselves with the application
2. The user performed 7 tasks within the user interface and recorded their results. These tasks were designed so that the user would have to use all functionalities of the user interface. Each task was performed and the results given were all correct
3. The user then gave feedback by answering 4 questions

4.4.4.2 Feedback

The feedback given by the user when asked each question was as follows:

1. Is the application easy to understand?

“ Not really because I don’t know what any of the train schedule-specific words mean but when asked to find a specific thing it was usually not too hard ”

2. Did you find it easy to navigate through the data?

“ No because the buttons to select work pieces and shifts were too hard to click (too small). The buttons should be bigger vertically and provide better feedback when clicked. More visual feedback should be shown when selecting work shifts because at present the transition is not noticeable enough. It was also tedious trying to find a specific work piece. Some of the work pieces could not be clicked on due to their being too small. ”

3. Is the application visually pleasing?

“ The application does not resize properly and the application does not look native to the platform. However, the colours for the work pieces look nice. ”

4. How do you think the application could be improved?

“

- Make it easier to find a specific shift
- Make the buttons easier to click by making them bigger
- Highlight all work spells of the same shift when one is selected
- Provide better visual feedback when selecting a work shift
- Make the window resize better
- An undo button
- Zoom function
- When “view shift” is selected, show which shift is enabled
- Double clicking on a shift should view that shift
- Some sort of toggle button to swap between viewing shift and schedule

”

4.4.4.3 Conclusion

Based on the feedback of the user evaluation and the time left available to spend working on the software application, a list of modifications was compiled that were to be made within the time left:

- **Increase the height of the rectangle components that WPs and WSs –** It was clear from the users feedback that a large annoyance was how hard it was to select WP and WS components. Increasing the width makes the hit box larger and therefore easier to click
- **Add a label that displays the colour of the shift that is selected by the user –** Due to the large amount of shifts in the schedule when using DS2, it is difficult to keep track of which shift is selected. Being able to see what the colour of the selected shift is will aid in finding it within the visual schedule
- **Change the cursors appearance when hovering over a selectable component –** A answer from the feedback, “Provide better visual feedback when selecting a work shift”, implied that it was hard to tell when a WS component had been selected or not. Therefore changing the cursors appearance to a hand whilst it hovers over a WS component will provide the user with knowledge that if they click at that position a component will be selected
- **Change the background colour of each *VehicleBlock* object that contains WS components that belong to the selected shift –** Changing the background of vehicle blocks when a shift is selected achieves both:
    - Providing the user with visual feedback that a WS component has been selected

    - And, to quote from the feedback, “Highlight all work spells of the same shift when one is selected”. Which will make navigating to each WS component that belongs to the selected shift an easier process
- **Give the user the option to view the shifts contained in the current train crew schedule as a list containing their property values. Also allowing them to select a shift in the visual schedule by selecting it from the list of property values –** Whilst observing the user evaluation it became clear that when the user wanted to find a shift based on a particular property value e.g. its ID or depot, it was a tedious process of selecting each WS component until finding the correct one. This list would enable the user to find a shift within the schedule from its ID value along with other property values too
- **Instead of having two separate buttons that allows the user to swap between viewing all vehicle blocks and vehicle blocks just for a selected shift, a single *ToggleButton* is used (described in section 4.5.1) –** It became clear during the user evaluation that having two buttons to perform the process of switching between viewing a collection of vehicle blocks and the entire schedule was a pointless waste of screen space due to the buttons never being enabled at the same time as each other

4.5 Iteration 5

4.5.1 Implementing the *ToggleButton*

A WPF *ToggleButton* has two states, *Checked* and *UnChecked*. It swaps between these states when it is clicked by the user and the default position is *Unchecked*. This enables the button to have two different event handlers for when it is clicked on; one for when it is checked and the other for when it is unchecked (Microsoft, c2015, *ToggleButton Class*). Instead of having two buttons for swapping between viewing the entire train crew schedule and the collection of vehicle blocks for a selected shift the main window has a *ToggleButton*, called *ViewShiftViewScheduleToggle*, that is enabled when the *SelectedShift* property does not equal *null*. The *ToggleButton* makes it possible for the user to jump between viewing the full list of vehicle blocks and a subset by doing the following:

1. When the user **checks** *ViewShiftViewScheduleToggle* the *scheduleStack* is emptied of all its children and the *VehicleBlock* objects that the *SelectedShift* covers are collected and added to it. Hence visually displaying the vehicle blocks covered by the user specified shift
2. When the user **unchecks** *ViewShiftViewScheduleToggle* the *scheduleStack* is emptied of all its children and every *VehicleBlock* object from *allVehicleBlocks* is added to it. Hence visually displaying the entire schedule

This implementation creates more space in the main window for other components.

4.5.2 Loading data files using file browser

Up until this point of the implementation the file names of the data sets had been hardcoded into the solution to avoid having to manually search for each file whilst developing. However, for the final delivery this was not useful as it does not allow the user to load multiple data sets without closing the application and editing the code. The process of enabling the user to select files from local storage is as follows:

1. The process is instigated when the user clicks the *LoadFilesButton* from within the main window
2. The main windows *getDataFileNames()* method is called
3. A file browser is presented to the user asking them to select a .trains file
4. The returned file name is stored to a list of strings called *fileNames*
5. Another file browser is presented to the user asking them to select the corresponding .shifts file
6. The returned file name is appended to *fileNames*
7. The main windows *run()* method is called and *fileNames* is passed as a parameter argument. Within *run()* is where the *RawData* object is initialised which starts the entire process of reading in the data using the file names, collecting it and displaying it

4.5.3 Saving the Train Crew Schedule to a file

Enabling the user to save a train crew schedule to a file opens up the possibility that the application can produce an output that could be used within another part of the planning of rail services operation e.g. Rostering (Discussed in section 2.2). The process of saving the current crew schedule is as follows:

1. The process begins when the user clicks a button called *SaveScheduleButton* that is contained in the main window
2. The buttons click event handler calls a method on the *Schedule* object called *SaveScheduleToFile()* and passes the *Shifts* object (section **4.1.2.1.2**) that has a 2 dimensional list property which contains all the raw data from the .shifts file called *allShifts*
3. Within the *SaveScheduleToFile()* method a list of integers called *shiftIDs* is populated with the ID values of every *Shift* object contained within the *Schedule* object's *shiftList* property
4. A *SaveFileDialog* is initialised and opened which presents a file browser to the user (Microsoft, c2015, *SaveFileDialog Class*)

5. Once the user has created a file name and pressed “Save” a *StreamWriter* is initialised using the created file name
6. The ID values contained within *shiftIDs* are used as index values to select lists of integers from *allShifts*
7. The *StreamWriter* writes the retrieved lists of integers to a new file line by line in an attempt to replicate the format of the .shifts files where each line represents a shift
8. The new file created is given the extension .schedule

Attempts were made to make it possible so that the user could load a saved train crew schedule file back into the application to review a previously created schedule. However, it became apparent that this would be a lengthy process and with the limited time left it could not realistically be implemented. It would have taken a while because it would have required a whole new *readFiles* method that read in the specific format of the .schedule files.

4.5.4 Small Additions

During the time between the end of implementing the last major functionality of the application and the delivery of the final solution, some small additions were made to the application. These are as follows:

1. Visual RO components had their widths reduced so that they covered less horizontal space. This was done to better emphasise that a RO occurs at a specific time in a train crew schedule
2. A new label was added to the main window that lists the start and end times of every WS that belongs to the currently selected shift. This is to help the user visual where the WS components are within the train crew schedule
3. Vertical lines were added to the visual schedule to better display the hour marks
4. The contents of the label that previously displayed the vehicle blocks ID value was changed to the train units ID for that vehicle block. This was done to emphasise that a vehicle block represents the journey of a particular train unit

4.5.5 Iteration 5 Tests

The tests performed to ensure that the application correctly loaded and saved files were as follows:

- **Loading files –** Having already loaded both the small and large data sets into the application this was a case of observing the visual schedule to see if it was identical to as it was prior to the new loading functionality

- **Saving files –** The save file functionality would be performed and the created file would be observed to see if the contents matched the current train crew schedule within the application

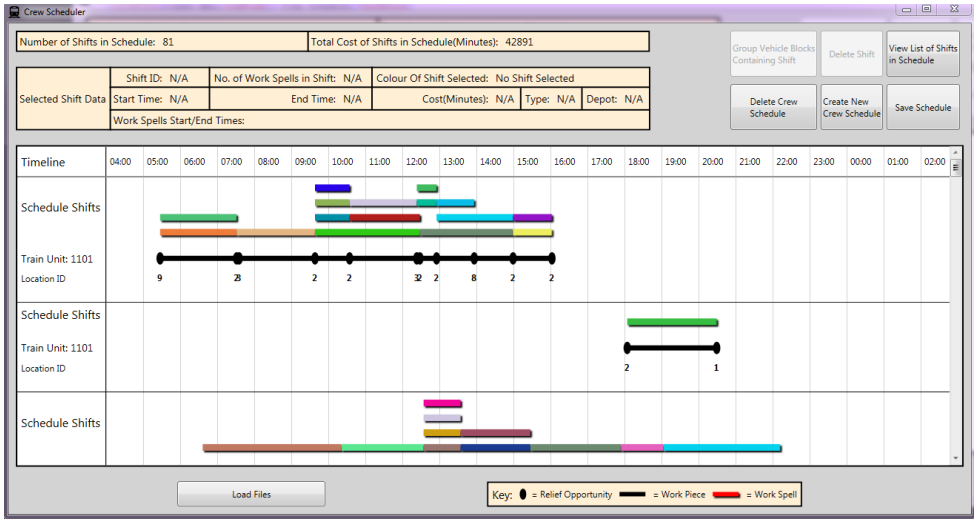


Figure 30 Screen cap showing the finalised main window of the application

5. Evaluation

5.1 Comparison to Atif Iqbal's Existing Solution

During the taught year of 2013/2014 Atif Iqbal undertook a project called "Scheduling Train Crews via a Graphical User Interface project". The aim of his project was to develop a software application that generates a train crew schedule and displays it as a visual schedule within a user interface (Iqbal, 2014, p1). Both solutions are designed to read in the same data file formats. The final solution of his project will be compared to the solution produced from this project. The comparison is performed to evaluate how each of the solutions handle presenting the large amounts of data involved in train crew scheduling and what functionalities each solution offers to the user. Iqbal's project was performed in the same timeframe as this project therefore comparing the two solutions will help gain an understanding of the quality of this solution.

5.1.1 Summary of functionalities of Iqbal's solution

First a brief summary of the main functionalities of Iqbal's application that can be compared to the functionalities of this projects solution:

1. Iqbal's solution gives the user the option to view a generated train crew schedule in terms of either Train Units (Vehicle Blocks) or WP components
    a. **Train Units –** The schedule is split into visual vehicle blocks that have identical widths and are directly stacked on top of each other. Each vehicle block is a bar chart with the X axis displaying the identification value of each RO and the Y axis displaying shift ID values. Each RO item displayed along the X axis is equidistant from each other. The rows created from the Y values represent a specific shift that is covering some part of the vehicle block. The bar chart contains horizontal rectangles that represent WSs. Each has a specific colour to distinguish it from others that are not part of the same shift within the same vehicle block (Iqbal, 2014, p39)
    b. **Work Pieces –** The schedule is split into visual WPs that are stacked on top of each other. Each visual WP's width is defined by the duration of the WP it represents in the data. The start and end times of each WP are displayed at the start and end of the X axis' respectively. The WP charts all start from the same horizontal position and, due to their varying widths, do not all end at the same horizontal position. The purpose of this visual display is to give an indication of the length of time of each WP within the data (Iqbal, 2014, p33)

2. Within the window that displays the schedule in terms of train units the user can add shifts to the crew schedule (Iqbal, 2014, p26 – p27). To add a shift the user performs the following steps:
    a. Selects a WP from a drop down list that displays them by their ID value
    b. Selects a shift that offers cover to the selected WP from a list of shift ID values that is generated and displayed once the WP ID is selected. The shifts that are already contained within the train crew schedule are not listed (Iqbal, 2014, p26 – p27)
3. Within the window that displays the schedule in terms of train units the user can delete shifts from the train crew schedule (Iqbal, 2014, p31). To delete a shift the user performs the following steps:
    a. Selects a WP from a drop down list that displays each WP by its ID value
    b. Selects a shift from a list of shift ID values that represent the shifts that are within the train crew schedule
4. The user can save the train crew schedule to a file in terms of shifts and their property values (Iqbal, 2014, p33-34)
5. The user can save a printed copy of the visual train crew schedule (Iqbal, 2014, p34 – p37)

5.1.2 Critique of Iqbal's solution compared to the solution of this project

A critique of each functionality summarised in section **5.1** compared to the similar functionalities performed in this projects solution:

1. **Visual display of schedules -**
    a. Iqbal's train unit representation of the schedule gives no indication of time due to the distance between each RO being the same throughout each chart. For the user to find the start and end times of a WS they would have to work out the ID values of the WPs that are at the start and end of the shift by counting pairs of ROs, then navigate to the visual display made up of WP charts and find the corresponding charts based on the ID values. To find the start and end times of a WS in this projects solution the user has to select the WS and the times are then presented in a label on the same page. Therefore in terms of explaining what happens in terms of time within the train crew schedule this projects solution outperforms Iqbal's
    b. The colours used to distinguish WSs in Iqbal's solution are assigned based on what order the WS is drawn onto the vehicle blocks chart which means a specific colour does not represent a specific shift which makes visualising a shift across the entire schedule confusing. In this solution for this project it is a

Shift object that is assigned a colour value therefore allowing the user to visualise a shift across the entire schedule via the colour of its WSs.

c. Iqbal's application displays WPs as separate charts stacked on top of each other. For large data sets it would become increasingly difficult to visualise the data due to the build-up of the number of stacked charts as the number of WPs increased. This project's solution avoids this problem by condensing crew schedule components into vehicle blocks and instead of displaying all of their property values within the visual display, they can be accessed by interacting with the components. This means that as the size of the data set increases Iqbal's visual schedule size increases at a much higher rate than the visual schedule of this project. To give an idea of the difference between the behaviour of the two applications as the data set sizes increase, the number of visual blocks added to the schedule per WP when switching between using DS1 and DS2 has been calculated (**Figure 25)**:

   In Iqbal's solution, the number of visual blocks added per WP is 1, due to each block representing a unique WP.

   To calculate the number of blocks added per WP within this projects solution we must first calculate the difference between the number of vehicle blocks of each data set due to the number of blocks drawn into the visual representation being equal to the number of vehicle blocks in the data set:

   47 – 3 = 44

   Then calculate the average number of WPs per vehicle block from data sets 1 and 2:

   ((22 / 2) + (260 / 47)) / 2 = 8.265

   Then divide the number of WPs by the difference in vehicle blocks between data sets to get the number of blocks added per WP:

   44 / 8.265 = 0.188

   Due to the visual blocks of both solutions being around the same size in terms of screen space it is clear that this solution scales better when displaying data sets of increasing sizes

d. It is impossible to obtain the following properties of each train crew schedule component in Iqbal's solution:

   i. RO – Location and Train Unit ID
   ii. Shift – Type and Depot

These are all important properties for each component e.g. without the Train Unit ID value of a RO a train planner cannot determine which vehicle a crew will work on. All of these properties are available in this projects solution by selecting the corresponding component in the visual train crew schedule. Every property value displayed in Iqbal's project is accessible within this projects application. Therefore in terms of data made available to the user this project outperforms Iqbal's

e. Iqbal's visual train unit schedule displays RO components by their ID value which makes the process of referring to a RO by its ID value in the schedule an easy process of observation. Finding a RO by its ID value in the solution for this project requires the user to select visual RO components within the schedule until they navigate to the correct one. Therefore in terms of navigating to a specific RO based on its ID value Iqbal's solution outperforms this solution

f. To avoid visual WSs overlapping Iqbal's solution creates a new row for every different shift within a vehicle block. The method created to avoid visual WSs overlapping in this projects solution is to only add new rows for a WS component to be contained in if there is no row available that already exists where the WS will not be overlapped in. This saves space by reducing the amount of rows having to be created whereas Iqbal's solution makes no effort to save space. Therefore as the data set size increases Iqbal's schedule size will increase at a higher rate than this solutions

2. **Adding a shift to the schedule –** When adding a shift to a train crew schedule in Iqbal's solution the user is given no information about the shift to be added except for the ID value of the shift and the ID value of one of the WPs that the shift will cover. When adding a shift to a train crew schedule using this projects solution all the properties of a shift from within the data files are presented to the user. Therefore a train planner would be able to make a more informed decision on what the properties are of a shift they are considering to add to a train crew schedule when using this projects solutions. What neither of the solutions do is allow the user to view a single list that contains all of the candidates shifts that are within the data files
3. **Deleting a shift from the schedule –** Deleting a shift in Iqbal's solution comes with the same problems as adding a shift, no data is given about the shift that is to be deleted other than its ID value and the ID value of one of the WPs that it covers. Before deleting a shift from a train crew schedule in this projects solution the user can manually select the shift from within the visual schedule to view all of its property values before deciding to delete it from the schedule. This once again means that a train planner can make a more informed decision whilst using this projects solution

4. **Saving the Crew Schedule –** Both applications allow the user to save a train crew schedule to a file. However Iqbal's application allows the user to save a printed copy of the visual schedule. This allows the user to refer back to a visual schedule. This projects solution provides no such functionality

5.1.3 Conclusion

Iqbal's solution provides limited data to the user in comparison to this projects solution. As can be seen from the calculations in section **5.1.2** this solution scales better when the size of data sets increase. Visualising the timeline of the schedule within Iqbal's solution is a confusing process of jumping back and forward between the two displays to be able to understand when each component occurs. Within this projects solution all the components that a train crew schedule consists of are displayed within a global timeline which makes it possible to compare start and end times of components just by observing their positioning. Both solutions provide the functionality of adding and deleting shifts from the train crew schedule and both enable the user to save the crew schedule to a file. However, only Iqbal's solution provides a way of printing the visual schedule which offers the user a method of storing and referring back to a visual display once the application has been closed.

Therefore it is believed that, whilst Iqbal's solution provides the user with the ability to print the visual schedule, it is otherwise outperformed by this projects solution due to; the difference in data provided to the user, the manner in which data is presented to the user and how they both scale when the data set size increases.

5.2 User Evaluation

An end of project user evaluation was performed by a researcher in Railway Planning. The purpose of this evaluation was to gain feedback from someone who has background knowledge in the relevant areas of the project. The assumption made is that a person who has background knowledge of train crew scheduling will be able to better formulate an objective opinion, than someone without background knowledge, on the applications attempt to solve the problem. The user evaluations structure was split into three steps:

1. A brief description of the projects aim, the timeline in which the project was performed and the general purpose of the software application was given to the user
2. The user was given time to explore the application and ask questions
3. The user performed a functionality evaluation that was designed to make them use all of the applications functions. The expected result was given in one column and the user documented the actual result in the column next to it
4. The user then gave feedback by answering previously prepared questions

5.2.1 Functionality Evaluation

Each functionality of the software application performed as expected during the user evaluation. The original template containing the users answers to this section as well as the feedback section can be found in **Appendix C**.

5.2.2 Feedback

Overall the feedback given was very positive. The answers to each question are as follows:

> Question 1: Do you think the application offers a good solution to the problem?
>
> Answer 1: “Yes, it does provide a good interface that allows modification of crew schedules in a feasible way. The user does not need to check for feasibility, but the application does. Moreover, the implemented algorithm enables to create new crew schedules and to compare them since properties are displayed.”
>
> Question 2: Is the application easy to understand?
>
> Answer 2: “Very easy”
>
> Question 3: Did you find it easy to navigate through the data?
>
> Answer 3: “Yes. They are graphically depicted, which also facilitates their comprehension.”
>
> Question 4: Is the application visually pleasing?
>
> Answer 4: “Yes. Different colours indicate different shifts, and different options allow to highlight desired properties”
>
> Question 5: How do you think the application could be improved?
>
> Answer 5: “It could be improved if the saved data can be loaded graphically. Different colours for different shifts would make them easier to differentiate. However, it is solved since, whenever one clicks a shift, just those work spells belonging to this shift are highlighted.”

5.2.3 Conclusion

The feedback received was very positive. Regarding the points made by the user in answer to the question “How do you think the application could be improved?”:

- **“It could be improved if the saved data can be loaded graphically” –** Attempts were made to achieve this as is discussed in section **4.5.3** however there was limited time left to complete the implementation
- **“Different colours for different shifts would make them easier to differentiate. However, it is solved since, whenever one clicks a shift, just those work pieces belonging to this shift are highlighted.” –** Whilst,

as the user mentions, there is already a part solution to the problem that it is possible two visual shifts can have the same colour. Another possible solution that could be implemented if time allowed it, would be to create a list of all the colour values available in WPF, then when a crew schedule is generated, hand the colours out from the list to each shift as it is added to the schedule. This would guarantee that every shift would have a unique colour

5.3 Self-Evaluation

To perform an objective self-evaluation on the project, the final solution is compared to the minimum requirements and possible extensions that were created at the outset of the project. Then a discussion on what would be done differently if the project was to be undertaken again.

5.3.1 Minimum Requirements

1. ***Read and prepare the data that describes a daily train crew schedule into a sensible class structure***

The class structure and data storage that was developed at the start of the implementation stage eased the process of adding functionalities throughout the rest of the implementation due to property values being easy to locate and access. The addition of the WS class type made displaying the shifts within the visual schedule possible. Therefore this minimum requirement was met.

2. ***A software system which uses the prepared data to create a visual display of a train crew schedule that is contained within a graphical user interface***

The software solution creates a visual train crew schedule that is compiled of the components discovered and defined during the background reading stage of the project. The quality of the visual schedule is validated by the feedback from the end of project user evaluation.

3. ***Allow navigation through the train crew schedules data via user interaction with the visual display***

Through the use of the *ListBox* class of WPF the visual display can be interacted with by the user. The user can; select components to view their specific property values, collect sections of the schedule so to better visualise certain components and highlight specific sections of the schedule.

4. ***Enable the user to modify the schedule e.g. by adding and deleting shifts***

The user is able to add and delete shifts from a schedule one by one. They can also able to delete the entire crew schedule and can create an entire new crew schedule at the press of a button.

5.3.2 Possible Extensions

- ***The application lets the user save the crew schedule to a file***

The user is able to save the current crew schedule to a file, however there is no current use for that file. It cannot be re-loaded back into the application nor is there another application that it could possibly be used for. So whilst the solution does perform the function, it has no use.

- ***Implement 3D graphics into the visual schedule display***

This possible extension was omitted due to the limited time available and because it wouldn't offer any aid in terms of helping the user navigate through the train crew schedule data, which was the main priority. However, shadows were added to the visual WS components in an attempt to make the display more visually pleasing.

- ***Improve an algorithm used to generate crew schedules***

In hindsight, due to the complexity of the algorithms used to generate crew schedules, this was an unrealistic 'possible extension'. However, what was noticed from the algorithms used is that none of them take the cost of shifts into account when generating a crew schedule. If somehow it was implemented that shifts with lower costs were somehow preferred to others then the algorithm would be more effective at minimising costs when generating crew schedules.

5.3.3 What would be done differently

Below is a list of what would be done differently and what steps would be taken to avoid the problems faced throughout the project:

1. Resizing of the desktop application was not taken into consideration at the start of the implementation. When eventually attempts were made to make the application resize sensibly it proved a very difficult process due to the different size values used throughout the entire source code. To avoid this, resizing tests would be performed throughout the entire implementation
2. More frequent user evaluations would be performed throughout the implementation stage to make sure the application was simple and easy to use. When the first and only user evaluation was performed during the implementation stage it was clear from the feedback that the application was not simple to use for a new user

5.4 Future Work

If more time was given to spend developing the software application the following list of improvements would be made:

1. **List of all Candidate Shifts –** Functionality that allows the user to view a list that contains every candidate shift from within the data files. This was attempted during the implementation stage (Section **4.4.3.2**) however the methods that were attempted caused the application to crash due to the large number of candidate shifts in the data set. A possible way of achieving this would be to create a list that loads 50 shift items at a time and the user can select a “load more shifts” button that would display 50 more
2. **A Zoom Function –** As was pointed out by the user during the mid-implementation user evaluation, the visual display can be very condensed when the time difference between schedule components is very small. This can lead to visual components overlapping each other which makes them hard to visualise and select. Some sort of zoom function would allow the user to zoom in on the components, splitting them apart. This could also allow for extra data to be displayed within the visual schedule by making property values visible once the user has zoomed in a certain amount
3. **Select the train crew scheduling Algorithm –** It could be possible that a train planner wants to compare train crew scheduling algorithms against each other. Functionality would be added that allows the user to select what algorithm they want to use before the crew schedule is generated
4. **File Handling Errors –** The application currently crashes if the user selects files that are not of the format of .trains and .shifts files. This would be fixed using file handling errors that would present a message to the user informing them of their mistake

5.5 Conclusion

To conclude, the results of each evaluation stage were overall positive. The software application provides an overall better solution to the problem than the pre-existing solution that was compared to. The feedback provided by the user who had background knowledge in Railway Planning was very positive and all of the minimum requirements were met.

Had time been managed more efficiently at the beginning of the project, then it is likely that there would have been time to add some of the features listed in section **5.4**.

Overall I am pleased with the solution that I have produced.

List of References

Albers, M. (c2009). Models, Methods and Applications. *Freight Railway Crew Scheduling*. 1 , .

CENTERS for MEDICARE & MEDICAID SERVICES. (2008). SELECTING A DEVELOPMENT APPROACH. . 1 (Introduction), p1.

Iqbal, A. (2014). Scheduling Train Crews via a Graphical User Interface. . 1 , .

Jaksmata. (2008). *Combo box.* Available: http://en.wikipedia.org/wiki/Combo_box. Last accessed 25th May 2015.

Kwan, R. (2009). Case studies of successful train crew scheduling optimization. . 1 (2), p47.

Laplagne, I. (2008). Train Driver Scheduling with Windows of RO's. . 1,.

Maheshwari, S and Jain, D. (2012). A Comparative Analysis of Different types of Models in Software Development Life Cycle. *International Journal of Advanced Research in Computer Science and Software Engineering*. 2(5) (III), p286.

Microsoft. (c2015). *Control Class.* Available: https://msdn.microsoft.com/en-us/library/system.windows.controls.control%28v=vs.110%29.aspx. Last accessed 28th May 2015.

Microsoft. (c2015). *Data Binding Overview.* Available: https://msdn.microsoft.com/en-us/library/ms752347%28v=vs.110%29.aspx. Last accessed 29th May 2015.

Microsoft. (c2015). *Introduction to WPF.* Available: https://msdn.microsoft.com/en-us/library/aa970268%28v=vs.110%29.aspx. Last accessed 25th May 2015.

Microsoft. (c2015). *ItemsControl.ItemsPanel Property.* Available: https://msdn.microsoft.com/en-us/library/system.windows.controls.itemscontrol.itemspanel(v=vs.110).aspx. Last accessed 25th May 2015.

Microsoft. (c2015). *ItemsControl.ItemTemplate Property.* Available: https://msdn.microsoft.com/en-us/library/system.windows.controls.itemscontrol.itemtemplate(v=vs.110).aspx. Last accessed 25th May 2015.

Microsoft. (c2015). *IValueConverter Interface.* Available: https://msdn.microsoft.com/en-us/library/system.windows.data.ivalueconverter%28v=vs.110%29.aspx. Last accessed 30th May 2015.

Microsoft. (c2015). *ListBox Class.* Available: https://msdn.microsoft.com/en-us/library/system.windows.controls.listbox%28v=vs.110%29.aspx. Last accessed 25th May 2015.

Microsoft. (c2015). *Rectangle Class.* Available: https://msdn.microsoft.com/en-us/library/system.windows.shapes.rectangle(v=vs.110).aspx. Last accessed 25th May 2015.

Microsoft. (c2015). *SaveFileDialog Class.* Available: https://msdn.microsoft.com/en-us/library/system.windows.forms.savefiledialog%28v=vs.110%29.aspx. Last accessed 31st May 2015.

Microsoft. (c2015). *Selector.SelectionChanged Event.* Available: https://msdn.microsoft.com/en-us/library/system.windows.controls.primitives.selector.selectionchanged(v=vs.110).aspx. Last accessed 29th May 2015.

Microsoft. (c2015). *StackPanel Class.* Available: https://msdn.microsoft.com/en-gb/library/system.windows.controls.stackpanel(v=vs.90).aspx. Last accessed 29th May 2015.

Microsoft. (c2015). *System.Windows.Shapes Namespace.* Available: https://msdn.microsoft.com/en-us/library/system.windows.shapes(v=vs.110).aspx. Last accessed 25th May 2015.

Microsoft. (c2015). *ToggleButton Class.* Available: https://msdn.microsoft.com/en-us/library/system.windows.controls.primitives.togglebutton%28v=vs.110%29.aspx. Last accessed 29th May 2015.

Microsoft. (c2015). *UserControl Class.* Available: https://msdn.microsoft.com/en-us/library/system.windows.controls.usercontrol%28v=vs.110%29.aspx. Last accessed 28th May 2015.

Microsoft. (c2015). *Visual Studio.* Available: https://www.visualstudio.com/. Last accessed 29th May 2015.

Microsoft. (c2015). *WPF Container Controls Overview.* Available: https://msdn.microsoft.com/en-gb/library/bb514628%28v=vs.90%29.aspx. Last accessed 25th May 2015.

Opcom. (c2001). Rail Crew Scheduling, Rostering and Management. . 1 (Overview), p1.

Qt. (c2015). *Using Containers in Qt Designer.* Available: http://doc.qt.io/qt-4.8/designer-using-containers.html. Last accessed 25th May 2015.

Appendix A
External Materials

GreedyHeuristics Python program provided by supervisor Dr Raymond Kwan.

The Max Coverage Gained and First Encountered algorithms contained within the solution were not devised during the project. It was stated by the project supervisor that the algorithms could be used within the solution as a means to generate crew schedules.

Data sets DS1 and DS2, provided by supervisor Dr Raymond Kwan.

Kwan, R (2015) File formats for the train crew scheduling datasets, Technical notes, School of Computing, Univ of Leeds.

Appendix B
Ethical Issues Addressed

The people who participated were given the following information prior to the user evaluations they performed:

- The purpose of the project
- What they were expected to do during the evaluation
- What the results of the user evaluation will be used for
- What information of theirs will be given in the report

They were then requested to sign a consent form.

This was done to provide proof of the users consent and to inform them of what details of theirs will be shared.

Appendix C Completed End of Project User Evaluation Form

Functionality Evaluation

Step #	Procedure	Expect Results	Actual Results
Test Purpose: Load Data Files From DS2 Folder			
1	Press “Load Files” Button	Message box saying “Select the Trains File you want to be read in.” appears	OK
2	Press “Ok” button	File browser opens	OK
3	Find and Open “DS2.trains” file from within DS2 folder on memory stick	Message box saying “Select the corresponding Shifts file.” appears	OK
4	Press “Ok” button	File browser opens	OK
5	Find and Open “DS2.shifts” file from within DS2 folder on memory stick	A visual schedule is drawn into the main window	OK
Test Purpose: Delete Crew Schedule			
1	Press “Delete Crew Schedule” Button	All visual work spells are removed from the schedule	OK
Test Purpose: Create new Crew Schedule			

1	Press “Create New Crew Schedule” Button	A new set of visual work spells are added to the schedule	OK
Test Purpose: View details of a specific shift			
1	Click on a visual work spell	Vehicle blocks that are covered by the selected shift are drawn a different colour. Labels at the top of the window are populated with data of the selected shift	OK
Test Purpose: Visually group vehicle blocks that are covered by the selected shift			
1	Press “Group Vehicle Blocks Containing Shift” button	Vehicle blocks covered by the selected shift are visually collected	OK
Test Purpose: Swap back to viewing the full schedule			
1	Press “Back to Full Schedule” button	The full schedule is redrawn into the window	OK
Test Purpose: View a list of the shifts contained within the crew schedule			
1	Press “View List of Shifts in Schedule” button	New window opens containing list of all shifts within the crew schedule	OK

Test Purpose: View a shift within the visual schedule by selecting it from list of shifts contained within the crew schedule			
1	Select a shift from the list of shifts	Shift selected becomes highlighted and labels below are populated with data relating to the shift	OK
2	Press “View Selected Shift in Schedule” button	Current window closes. Main window is presented. Labels at top of main window are populated with data relating to the selected shift. The vehicle blocks that the selected shift covers are drawn a different colour	OK
Test Purpose: Delete selected shift			
1	Press “Delete Shift” button	The selected shift is removed from the visual schedule	OK
Test Purpose: View property values of a Work Piece			
1	Click on a visual Work Piece	New window opens displaying property values for the selected Work Piece and the start Relief Opportunity	OK
Test Purpose: View list of candidate shifts that offer cover to the selected Work Piece			

1	Press “Add Shift to Work Piece” button	New window opens containing list of candidate shifts that offer cover to the selected Work Piece	OK
Test Purpose: Add shift to schedule that covers the selected Work Piece			
1	Select a shift from the list presented	Selected shift is highlighted and further properties are displayed in grid at bottom of window	OK
2	Press “Add Shift” Button	All windows close except the main window. The selected shift is added to the visual schedule as work spell components	OK
Test Purpose: Save Crew Schedule to a file			
1	Press “Save Schedule” button	File browser opens	OK
2	Name the file DS2 and save it to the desktop	A new file is saved to the desktop that contains lines of integer values that represent the shifts within the current schedule	OK
Test Purpose: Load new set of data file			
1	Press “Load Files” button	Message box saying	OK

		"Select the Trains File you want to be read in." appears	
2	Press "Ok" button	File browser opens	OK
3	Find and Open "DS1.trains" file from DS1 folder on memory stick	Message box saying "Select the corresponding Shifts file." appears	OK
4	Press "Ok" button	File browser opens	OK
5	Find and Open "DS1.shifts" file from DS1 folder on memory stick	A new visual schedule is drawn that represents the data from the new files	OK

Feedback

Question	Answer
Do you think the application offers a good solution to the problem?	Yes, it does provide a good interface that allows modification of crew schedules in a feasible way. The user does not need to check for feasibility, but the application does. Moreover, the implemented algorithm enables to create new crew schedules and to compare them since properties are displayed.
Is the application easy to understand?	Very easy
Did you find it easy to navigate through the data?	Yes. They are graphically depicted, which also facilitates their comprehension.

Is the application visually pleasing?	Yes. Different colours indicate different shifts, and different options allow to highlight desired properties.
How do you think the application could be improved?	It could be improved if the saved data can be loaded graphically. Different colours for different shifts would make them easier to differentiate. However, it is solved since, whenever one clicks a shift, just those work spells belonging to this shift are highlighted.

Appendix D Completed User Evaluation Form That Was Performed During the Implementation

Instructions

1. Spend 5 minutes to gain an understanding of the user interface
2. Perform the tasks presented in the Tasks section
3. Answer the questions in the FeedBack section

Tasks

1. What is the total cost of the crew schedule: 40424
2. Find what time Train Unit 1202 ends its work at and record the value here: 1654
3. Find the identification numbers for the shifts covering Train Unit 1391 and record the values here: 11733 11690 425
4. Delete all shifts covering Train Unit 1391 and then add shifts to cover both its pieces of work so that the overall cost of shifts covering Train Unit 1391 is less than 600, record their ID values here: 22 425
5. Find and record the cover count of work piece 241: 1
6. Add Shift 49167 to work piece 11 and record all the Train Units it covers: 1203 1101 1107
7. Calculate the entire cost of shifts covering train unit 1105: 1969

Feedback

1. Is the application easy to understand?

Not really because I don't know what any of the train schedule-specific words mean but when asked to find a specific thing it was usually not too hard

2. Did you find it easy to navigate through the data?

No because the buttons to select work pieces and shifts were too hard to click (too small). The buttons should be bigger vertically and provide better feedback when clicked. More visual feedback should be shown when selecting work shifts because at present the transition is not noticeable enough. It was also tedious trying to find a specific work piece. Some of the work pieces could not be clicked on due to their being too small.

3. Is the application visually pleasing?

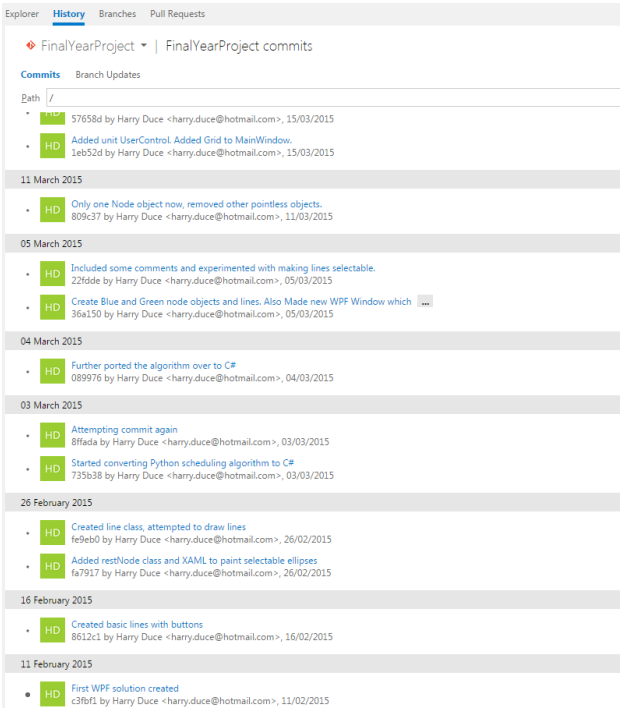
The application does not resize properly and the application does not look native to the platform. However the colours for the work pieces look nice.

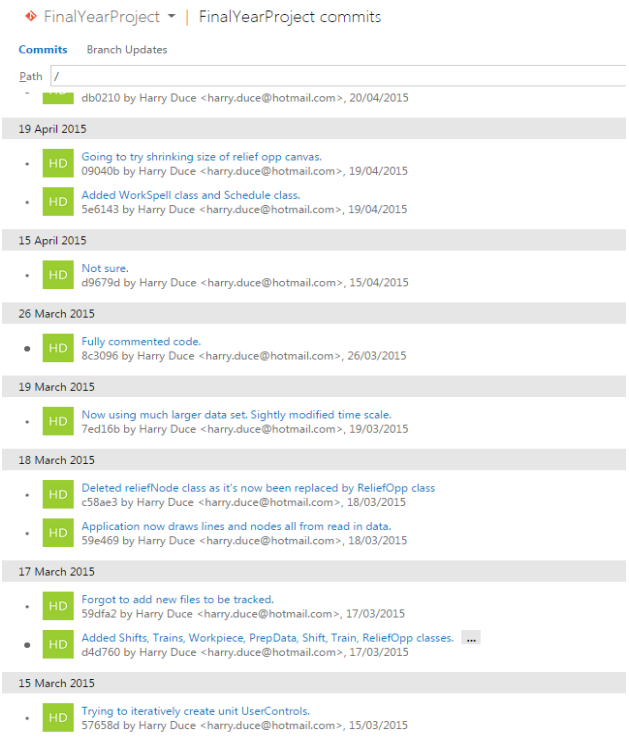
4. How do you think the application could be improved?

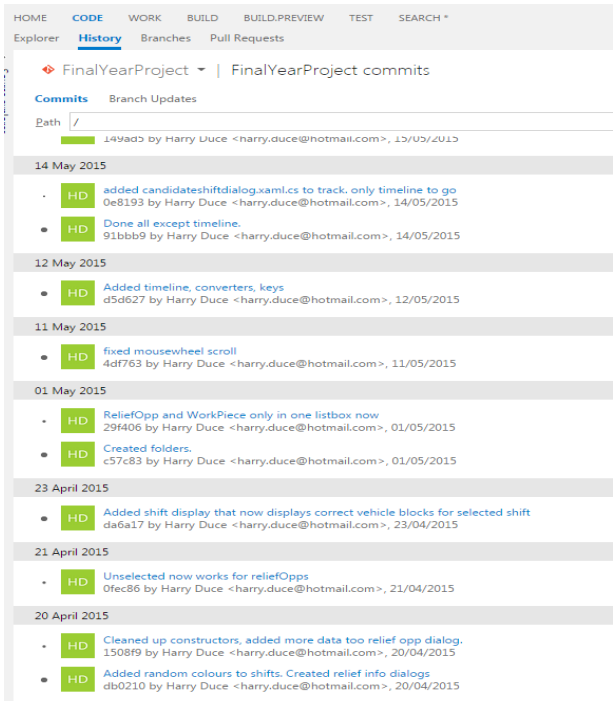
- Make it easier to find a specific shift
- Make the buttons easier to click by making them bigger
- Highlight all work spells of the same shift when one is selected
- Provide better visual feedback when selecting a work shift
- Make the window resize better
- An undo button
- Zoom function
- When “view shift” is selected, show which shift is enabled
- Double clicking on a shift should view that shift
- Some sort of toggle button to swap between viewing shift and schedule

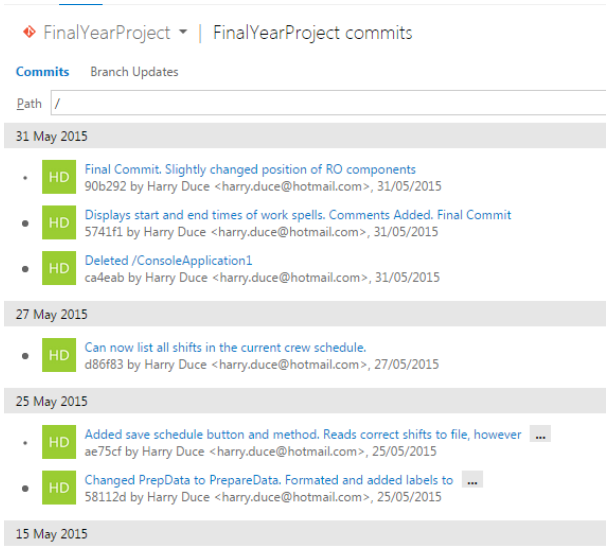
Appendix E Proof of Version Control System

Due to me not supplying the repository for my software application, I have provided proof that the version control system that was chosen (Discussed in section **3.4.3**) was used throughout the project in the form of screen caps of the commits made:






HOME
CODE
WORK
BUILD
BUILD.PREVIEW
TEST
SEARCH *
Explorer
History
Branches
Pull Requests
FinalYearProject | FinalYearProject commits
Commits
Branch Updates
Path /
14 May 2015
HD added candidateshiftdialog.xaml.cs to track, only timeline to go
0e8193 by Harry Duce <harry.duce@hotmail.com>, 14/05/2015
HD Done all except timeline.
91bbb9 by Harry Duce <harry.duce@hotmail.com>, 14/05/2015
12 May 2015
HD Added timeline, converters, keys
d5d627 by Harry Duce <harry.duce@hotmail.com>, 12/05/2015
11 May 2015
HD fixed mousewheel scroll
4df763 by Harry Duce <harry.duce@hotmail.com>, 11/05/2015
01 May 2015
HD ReliefOpp and WorkPiece only in one listbox now
29f406 by Harry Duce <harry.duce@hotmail.com>, 01/05/2015
HD Created folders.
c57c83 by Harry Duce <harry.duce@hotmail.com>, 01/05/2015
23 April 2015
HD Added shift display that now displays correct vehicle blocks for selected shift
da6a17 by Harry Duce <harry.duce@hotmail.com>, 23/04/2015
21 April 2015
HD Unselected now works for reliefOpps
0fec86 by Harry Duce <harry.duce@hotmail.com>, 21/04/2015
20 April 2015
HD Cleaned up constructors, added more data too relief opp dialog.
1508f9 by Harry Duce <harry.duce@hotmail.com>, 20/04/2015
HD Added random colours to shifts. Created relief info dialogs
db0210 by Harry Duce <harry.duce@hotmail.com>, 20/04/2015


FinalYearProject | FinalYearProject commits
Commits
Branch Updates
Path /
31 May 2015
HD Final Commit. Slightly changed position of RO components
90b292 by Harry Duce <harry.duce@hotmail.com>, 31/05/2015
HD Displays start and end times of work spells. Comments Added. Final Commit
5741f1 by Harry Duce <harry.duce@hotmail.com>, 31/05/2015
HD Deleted /ConsoleApplication1
ca4eab by Harry Duce <harry.duce@hotmail.com>, 31/05/2015
27 May 2015
HD Can now list all shifts in the current crew schedule.
d86f83 by Harry Duce <harry.duce@hotmail.com>, 27/05/2015
25 May 2015
HD Added save schedule button and method. Reads correct shifts to file, however ...
ae75cf by Harry Duce <harry.duce@hotmail.com>, 25/05/2015
HD Changed PrepData to PrepareData. Formated and added labels to ...
58112d by Harry Duce <harry.duce@hotmail.com>, 25/05/2015
15 May 2015