# Game Components and GUI

Software Engineering Project (SEPR)

## Team “EEP”

*Richard Cosgrove, Yindi Dong, Alfio E. Fresta, Andy Grierson, Peter Lippitt, Stefan Kokov*
Department of Computer Science
University of York

For the current development stage, the TaxE game has been requested to include specific game Based on our available time remaining after the essentials were implemented components and has been requested to incorporate this through a GUI to allow the user to interact with these game components. All images used are open source and have been referenced. The following report discusses the design decisions made for the required game components and GUI, explaining the manner in which it has been implemented and justifications as to why.

**GUI**

The aim of our Game GUI was to create a consistent feel throughout the users time in the game and an easy to use interface, which would have to be judged by the clientele. The GUI needed to look of a highly, professional quality to encourage users to play the game and immerse themselves so much so that they would be playing for hours. These were suggested as a starting point by the clientele to come up with prototypes, hence the team conducted research for current turn-based games as market research to acquire ideas. Prototypes were made using RealtimeBoard [Appendix D] and shown back to the clientele. Multiple prototypes were put forward to the clientele who provided feedback containing strengths and weaknesses for each prototype. These were taken into account to create a final prototype that met all the essential requirements in the requirements specification. The final prototype was agreed by the clientele to implement. Updates will be given on any design decisions that may alter. A gameview hierarchy [Appendix E] was created by the development team to help guide developers, both present and future, as they are making the game, showing the interactions between the different game interfaces and components. It can be used to help understand the GUI architecture of the system.

To create the GUI of the game, our team had to make a design decision. Our team investigated into two methods: using libGDX or Java’s Swing. LibGDX used a Java game development framework for supporting while Java’s Swing utilises Java’s GUI widget toolkit. Research was conducted into both methods and a meeting occurred to discuss which method appealed to us. A design decision was made by the team to use Swing as it utilised Java’s existing GUI libraries, meaning that the coding was clean of errors. By using the existing Java libraries, our team did not need to construct their own background coding or game engine, which may have taken up a considerable amount of time due to the lack of programming expertise amongst all of the development team. Our time could not accommodate creating another engine and as some members of the team had previous GUI creating experience within other programming languages (such as VB.net), it was recommended to use a familiar environment and Swing enabled an easier transition to reduce the amount of training time.

Another reason was that a team member discovered a good featured addon for Java Eclipse called WindowBuilder. WindowBuilder uses the SWT Designer and Swing Designer to provide a WYSIWYG interface, creating a visual aid for the developer. This was far easier in creating the user interface we wanted as it constructed the code for you and did not require the developer to compile the code, acting in a similar fashion to other GUI creators, however, the code was computationally expensive and hard to understand. Our development team needed to refactor the code, which took up a lot of time and resources, but increased the readability and maintainability of our system for future developers.

All image files are of a JPEG format. A JPEG format was chosen because of the high quality in image without being allocated large amounts of bytes to store it in memory through its lossy compression. This will help the user process and store the game more efficiently without hindering image quality too largely, which would help to meet

the clientele's needs of processing and storing the game as fast as possible (non-functional requirements in requirements specification). Also, JPEG images were chosen from online sources that were copyright free as an alternative to drawing the pictures. The decision was made to spread our resources and time across the coding of the system to get the game mechanics working and not spend too long creating images. The game mechanics were more important to the clientele at this development stage, however, they did not want the GUI to lose its professional quality, hence appropriate JPEG images that were of a high quality were used. JPEG images are easily maintainable for future development teams and are implemented through very little coding.
At this stage, the GUI does not look like the prototypes shown previously to the client. Changes have been made due to a lack of time and different requirements being stated at this development stage. The team has done its best to comply with all the requirements stated by the client at this development stage, however, due to a lack of time and available personnel, not all requirements have been implemented or in the highest quality standard. As a design decision, enough space has been allocated onto the GUI for the development of game components at the later stages of development. This will provide future developers with creative freedom in where they may place additional features or modify existing ones.

By creating a house style for our GUI of grey and white throughout the game, the users will be able to immediately identify that the game belongs to the clientele, adding to the identity of the game. This may also help as a form of advertisement as users may remember the game and tell their friends when they see the house style in the real world, increasing the potential market. The grey and white contrasts sections of the map from each other to clearly define the separate components of the GUI and to put more of a focus on other parts. For example, the map is more colour against the surrounding GUI, hence it will help to focus the users onto the map where the game is taking place. The same effect can be seen with text throughout the game.

**Main Menu GUIs (need screenshot)**

*Choose Game Server*

As the game uses a network to play the game, the user needed some method in connecting to a game server list. By implementing a simple notification popup upon initialising [Appendix ], the system can explain what is required for the user to enter the game without needing to look up what to do in a user manual. It is designed to look like a technical issue because it would discourage users altering the server information unless they know what they are doing. For technical users, it would allow the user to change the address of the server they wish by entering a valid server into the text field without being expensive to process internally in the event that they or someone else may wish to set up a TaxE game server. This can be a selling point of the clientele as it provides users with freedom with regards to server creation.

A team decision was made to automatically fill the text field with the default TaxE game server to enable a user to enter the game as quickly as possible and without worrying about where they have to connect to incase they do not understand what is being requested. The user may not know an address of a TaxE game server too, hence providing a server that automatically places its address into the field will reduce the risk of users not being able to play the game.

Centralising the menu was automatically implemented as users have a large tendency to look towards the centre of the screen and any information portrayed to the user is generically done through the centre of the screen. This will stop users searching for the notification that may otherwise discourage them from playing the game as they may think the game has crashed, due to being the first menu, and may believe that the entire system is in a similar fashion, finding it hard to navigate or find any information in the game.

In addition, by providing a cancel button, users may immediately quit out the game in the event that they do not want to play or change their mind when initially running the game. This will stop the need to process the game any further or take up unnecessary memory allocation for users computer systems, reducing the risk of their system crashing. The same idea is implemented throughout most of the menu interfaces with the same reasoning.

*Game Menu*

After stating the TaxE game server to connect to, users are taken to the game menu [Appendix ]. The idea is to immediately engage the interest of potential users by implementing a background image related to the game. This provides a first impression of what the game will involve, i.e. trains, and creates a more interesting, professional touch to the game due to no whitespace being shown. The user would feel more encouraged to continue into the game.

The game menu has been implemented by creating separate screens as JPanels that lay on top of the other and made visible when required. The reason for this is because the developer knew how to implement the game this way, which saved time as it would be created quicker and the developer would not need training in a new method. Computationally, the game menus would also only need to be loaded once on the system rather than each time the menu is initialised, improving speed and memory efficiency, and helping to meet the non-functional requirements of the system. This method would be easily maintainable and adaptable in the future if additional features were to be added due to only needing to create a new JPanel and not a new interface menu.

The first screen they are taken to is the name screen. Within every multiplayer game, the players need a method in which to distinguish themselves from the others, hence a naming system is provided before entry into the game. The idea is to help bring out the creative freedom of individuals which may persuade users to join the game. It will also help users recognise their friends when creating a game lobby. The idea was shown to the clientele and was requested to be a requirement with a preferable priority as it was not required, however, our team implemented the solution as the amount of resources and time required was low due to our programming experience with basic GUIs. In the future development stages, the name system will be filtered to stop inappropriate names appearing as user names, otherwise it may deter users away from the system, but could not be implemented at this stage due to a lack of time, complex algorithms and it was not specifically requested. The user may not have an empty name, bringing up an appropriate error message in the centre of the screen to alert users of what went wrong, otherwise they would not understand what went wrong.

The second screen takes the user to the server screen showing the server list. The server list is shown as a table with its unique ID, game name (player's name), difficulty (easy, medium, hard) and creation date on the left hand side of the screen, taking up half of the screen. The user can double click a server to join it, go back to the previous menu or create a server where they will state the difficulty on the create game screen. Difficulty is not implemented as it is not required at this stage and not enough resources and time were available, but is there in the event that future development teams may wish to create it.
The development team made this design decision to show users only the vital information that will help them decide what game to join while providing a visually pleasing, interactive method in choosing the game server. Using a list can display the information in a consistent, professional layout while providing the ability to easily search for a game to join as this is done in many other games too. The scale of it also allows the user to track many different games at once and make the user focus on it. The development did, however, have to allocate extra resources and time due to the training and coding experience required.

If a user creates a game, then the final screen is a simple indication to the user that specifies that they need to wait patiently for another user to join. If a user is joining, they are taken directly into the game with the other user.

**Main Game GUI**
The main game GUI [Appendix F] is split into four key areas using custom JPanels in Swing: LeftPanel, RightPanel, TopPanel and BottomPanel. This is to aid the readability of the code to help current and future developers create or maintain features in the game by easily determining which area of the screen they wish to work on. It was also the method that our development team knew that would help place any features on the screen by using coordinates inside the JPanels, rather than on the main interface otherwise it can be glitchy and awkward to place Swing objects. Another idea here is to clearly split the key information that need to be displayed into these four JPanels to help the user while playing the game. It also helps to create a consistent and professional interface by helping to manage the layout through the positions of objects. This was also used to stop many features cluttering together that may overload the user and deter them from using the system. From a user perspective, having the interface

split into sections can help to identify and understand the game's conventions as well as providing a familiar interface upon each visit. Users will be able to quickly identify what information will be of use to them and saves the user having to click through many menus to find hidden key information about the game.

The resolution of the GUI is 1300x720. Originally, our team decided the game create the game's resolution at 1366x768 to replicate the average screen size of the target market (laptops and monitors), however, errors occurred when we did. By using our current resources, i.e. a team member's laptop, with a 1366x768 monitor, we tested the GUI, but the borders of the GUI were cut off when it was compiled. To solve the issue, we reduced the resolution to 1300x720 which would still keep a large game resolution to place all our objects onto, but small enough to fit neatly onto a 1366x768 GUI. Although we decided on a set window size most of the components in the game GUI are resizable, however, this feature is not fully tested and may require additional changes.

To help new users to the game, the GUI contains an instruction notifier. It will help guide new users through the first few plays of the game by asking for a particular action related to the routing, however, it may also help other users determine why certain actions are not being performed. Using black text and horizontal lines with a white background help to separate it from the rest of the GUI, making it a focus point for the user, and help to improve the text's readability so that users can read it. This is also supported by the fact that it is contained at the top of the screen and helps to continue the house style running through the game for the clientele's identity. The idea was supported by the client in the prototype stages and was recommended by the development team too as it would help the user and not be hard to implement.

As a note, the rightPanel also includes a tab for regions, the functionality of which is not included in this version as relates to the obstacles requirement in the next development stage. The regions tab was agreed upon by the team and the clientele which uses a map icon to symbolise it. Its purpose is to change the information in the rightPanel to the region information that the selected train is in . Regions can be used to implement geographically specific traps.

Each of the game components of the game GUI are explained below.

**GAME COMPONENTS**

**Map**

The map (point 12) [Appendix F] is a fundamental component of the GUI that provides the game with a playing board. The playing board will contain both players' trains, the graph containing the cities (nodes, point 17) [Appendix F] and railways (edges, point 13) [Appendix F] and the route in which the player's train is taking. The assessment does not state whether the map must be fictitious or a particular way of implementing as long as it complies with the following requirements: the map is part of the GUI; the map shows cities and train routes between these cities; the map is implemented during this stage of development. The reason for implementing a map for the clientele is to look more appealing to play. It provides the users with a purpose for completing the goals and is more enjoyable when you can visually see your playing pieces complete a goal to achieve victory. It also immerses the user more into the game to make them feel like they are involved in changing the game world, which would add to the potential replayability of the game.

The team made a design decision to implement a map of Europe. By using a map of Europe, providing a familiar environment to play the game in can add to its replaying potential. By relating the game to the real world, the game can add a realistic element to target a wider audience (those who look for realism), which adds a possible angle in the future to add realistic goals (e.g. relating to transporting resources in World War II) or obstacles (e.g. severe weather conditions at cities/junctions near Russia or Finland). The clientele also wanted a map for their game, hence the development team were meeting the minimum requirements set by the clientele and those that were discussed in the requirements specification.

By centralising the map in the interface, it would become the first object to be viewed on screen by most users. As the map is the main way in which a user interacts with the game, the centralised view makes it the main focus, which would draw users in. This is supported by having the map's colour scheme contrast with the surrounding

panels and helps distinguish the playable region of the game, reducing the risk of the player being confused and hence not playing the game.

In addition, due to our available resources and time for creating the game (including a lack of artistic flair amongst most individuals), a pre-made map was used, enabling the team to focus entirely on the coding. As the client had set a deadline, the first priority was meeting his requirements and having the game mechanics implemented would at least make a playable game. If time was available, the map could be have been created to add a personality and touch unique to the game, which would have been an extra selling point for the clientele to potentially gain more users.

**Age**

Age within the game is not a required game component, however, it was first stated in the requirements specification when shown to the clientele as a potential feature. It would be included to provide the game with an exciting feature and act as a good selling point of the game to gain more users, in which they highly recommended to implement. Age has four discrete values: Steam Age (first), Combustion Engine Age (second), Electrical (third) and Futuristic (fourth). A player progresses to the next age by completing a set amount of goals to enable players to gain a sense of progression and accomplishment in the game, adding to its replay and fun potential. A new age provides access to new resources which is currently only speed modifiers at this development stage.

The age uses a JLabel within a JPanel to output the age value as a string to the users of the system. This is to provide more accurate placement on the GUI to make the system look more professional and to allow variables to be output to the screen through the GUI. The current age the player is on is shown in the top left hand corner of the game GUI (point 8) [Appendix F] to be out of the way of the main game playing board while providing a quick visual reference for the users who wish to know their current age. If further interfaces are made, then it can be consistently shown throughout all menus. For example, if the player enters a store to purchase resources, they can look at the age without backtracking or closing any menus to enable them to recognise the resources they can purchase based on age. This would help increase game experience by helping to support the strategic element added from age.

The idea of the age is to give future developers of the system a creative angle to add new resources, such as new trains or traps, to allow the game to have a longer lifecycle than other games on the market, i.e. a way in which to get users to repeatedly play the game. The code is highly readable and easy to understand to enable future developers to understand how we implemented the age component so that they can manipulate it how they wish to add extra features based on the age or modify existing ones. By increasing the readability of the code, we increase the maintainability of the system for future developers. For example, if a bug is detected, then a future developer could quickly pick up the coding of the age and fix the problem.

From the development team's perspective, the time required to implement the age game component was not substantial. The team's experience in programming was good enough to have one developer work on implementing the age into the GUI while another would implement the coding for the module. This would allow us to distribute our resources well by enabling the developers to program in parallel as often as possible to help maintain our target of reaching the deadline on time. It would not reduce the chance of risk occurring, however, the severity would be lessened by having other developers to help if requested and by having enough spare time to sort the problem out.

**Resources, including how they are generated or devised. Can talk about Ages increasing more resources available. Store etc.**

The reason for implementing the resources was to add an extra dimension to the game for more interesting gameplay, while also being a required game component. The requirements state that the game must support at least 10 different resources and a player can gain 2 resources per turn, have at most 7 on them and must have them removed once spent. There are two main resources (gold and metal) and up to five other resources at one time, displayed in the inventory at the bottom of the screen.

The game incorporates the 10 unique resources by using a combination of types of resources, such as those that affect the trains speed, different train upgrades available at the stated ages and obstacles (though not at this stage). The ideas for these were provided by the team, which were given feedback on, and the clientele, which were automatically considered providing our team had the coding experience, if it was required at this assessment stage, any available time and if it was high on priority. For example, the client suggested using springs as an obstacle, however, our team does not need to implement obstacles at this stage and therefore does not need to be included. Another example is the use of different permanent train upgrades which were originally suggested to have multiple values, but due to time constraints and already reaching the minimum resource count it will not be implemented at this stage. The structure of the code for the train upgrades have been done in a highly readable way in which it can be maintained well and can be added to with relative ease.

To comply with the 2 resources per turn requirement, the gold and metal resources (point 4 and 5) [Appendix F] were suggested to the clientele, in which they agreed it would be a good addition. This is a spendable resource and would be displayed in the top right corner of the GUI as JLabels with a gold icon (Point ) and a metal icon (Point ) with their respective values. The icons stand out boldly and clearly in comparison to the background to indicate it is not part of the map, but still part of the game.The icons would allow the player to visually identify the resource value they want to know, which is quicker and less effort to them than reading “gold” or “metal”. In addition, the values would always be present when entering the store to improve the gameplay experience as the user would not need to exit the store to recognise what they can buy. Unfortunately, we have not been able to implement the store system, making gold and metal redundant, however, these have been left in the event that another team may wish to create it. From a developer point of view, it was far quicker and of a higher quality to acquire from an open source than to create our own, especially when time was of the essence and our team wanted a working solution for the client over graphics because the client requested these as top priority.

Other resources are classed as usable and are placed in the inventory at the bottom of the screen (point 19) [Appendix F], which could be a resource that helps the player or an obstacle to hinder the other player. Obstacles are not available because they are not required at this development stage. This is done by using a store to purchase resources which are placed into an available inventory slot. An Inventory JLabel as a square box in the BottomLabel (i.e. inventory) will be filled with the appropriate resource purchased (denoted by a type of symbol), which will become clickable to use in the game and would depend on the resource. The resource is a square box to help define the clickable area for users, and allow developers to pinpoint the exact location they want it to improve the layout quality, making the GUI look more professional. The two resource types currently implemented (eight including them at the different ages) are speed boosts (rocket icon, point 21) [Appendix F] and slow downs (snail icon, point 21) [Appendix F] with an additional empty resource icon (point 22) [Appendix F]. These were acquired to help users categorise the type of resource they are visually, which should help speed up and improve game experience for users. It was able to be implemented as similar coding is used in other parts of the GUI and our developer had enough experience to tackle the task on their own, but based on our time and resources, we had to acquire the images from an open source. This loses identity to the game but can be amended later by future development teams if required. For example, a train upgrade would have to be clicked onto a train while an obstacle would have to be clicked on the track. The resource, once used, will be removed from the inventory positioned and replaced with an empty resource slot. There are a maximum of five resource slots in the inventory to comply with the requirement of having a maximum of 7 resources at a time. The decision to implement an inventory system was suggested by the team to the clientele and they recommended to include it as it is far easier for the user to manage and take note of what resources they have available to use by using visual aids. It would also make it easier for the developers to implement into a separate class that would help improve the maintainability and readability of the code.

By using a click and click to use inventory system, the game would become more interactive, keeping users more engaged in the game and would add an extra unique selling point for the game. This would be hard to implement based on the teams current experience overall, but would be a very useful GUI component to the resource system and recommended by the clientele. Therefore, enough time will need to be given to train and implement the

system. The same system explained above was used to implement trains onto the GUI, however, this was too complicated and not required to complete at this stage, therefore the team focused on tasks elsewhere.
At the end of each turn, resources will be checked to see if they have been spent and removed if so to comply with the removal of resources requirement. This will be shown through the inventory system where, once a resource has been used, will remove the resource symbol from its slot and replace it with an empty symbol slot (point 22) [Appendix F]. From a client's perspective, the removal of resources meant a more interesting game environment in which to play, keeping users engaged for longer periods and a purpose behind using the resources. while stopping both players from cheating. The inventory system was also a good GUI addition to enable users to keep track of what spare resource slots they do have so that they can acquire more if they want to and it was recommended by the client to implement. From a developer's perspective, the resources were removed to stop both players using the same resource repeatedly over the turns, stopping any potential cheating, and to comply with a must have requirement. Based on our current resources and team ability, the removal system was not hard to implement and seemed the best method (efficiency and our individual programming skill) to implement.

**Cities/Stations**

In the TaxE game, cities (point 17) [Appendix F] are used as destination points (nodes on a graph) to create journeys (paths) by following train routes (arcs) using the Java Vertex class. The cities are used to create goals, which in this current assessment is absolute goals such as "get to Paris from Rome" while quantifiable goals will be added in the next assessment. Cities are denoted on the map as a red square that have a name tag next to them and should be geographically correct. There must be at least five cities and these must be connected to the train network. From the client's perspective, having cities would give a purpose to the game as a user going to cities to complete goals is one of the fundamental game mechanics that they requested. By using red squares with a name tag next to them, a user would easily be able to distinguish the difference between what is a city and what is not, reducing the chance of any confusion amongst users. Users may otherwise be discouraged from wanting to play again, reducing its potential ability to be replayed and therefore fun factor. From a developer perspective, the method in which this was implemented was agreed by the team. It was far easier and required less skill to implement due to the fact that our team had experience in creating graphs before and Java had created a built-in class to use for a graph situation. The red square was simple and quick to create as it did not require any artistic ability that our team lacked. This meant more resources could be deployed elsewhere while also reducing the amount of time spent implementing this feature, which could be redistributed to other areas. The readability and maintainability of the code is also increased by using a conventional style to create graphs set out by the Java Vertex class and generic coding standards. This should allow future developers to understand the methods that link to the graph in a much simpler manner for something which can be complex.

**Absolute Goals**

The reason for implementing the goals is because they are a fundamental part of the game's mechanics, defining what the player must achieve to acquire any sort of score and determine which player wins. The requirements and assessment state that the game must be able to be given a train goal each turn, have at maximum three goals and support at least 10 different goals (absolute or quantifiable). The assessment also states that, at this stage, only absolute goals are required.

To comply with the game requirement of being given a train goal per turn, the game provides a goal to the player each turn. It also generates a goal to a random city that starts in the ending city of the previously generated goal (e.g. first goal ends in Sofia, next goal will start at Sofia) to comply with one of the requirements of the game and to enable player's to chain together goals if they wish. By explicitly showing the goals at the right hand side of the screen (right JPanel) under the "details" box, a consistent location is provided on every screen to provide a familiar GUI to provide a quick reference point for the user to understand what they need to do to win the game. If the user does not know what they are doing, then they may lose interest in the game or find it harder to understand how to acquire score. Containing the goals inside a black outlined box aids in helping the user identify where key information would be found, i.e. goals in this case, while bold text helps to clarify the difference between a goal and its objective details. These were favoured by the clientele as all the goal information would be provided on screen at the same time without the need to switch interfaces or go through additional menus, improving its ease of use. A

bullseye image was also chosen for the goal (objectives) tab to symbolise “hitting the target” or reaching your goal, which would look more appealing to users to click. At this stage, it does not do anything, however, it will be used to alter the information contained in the right JPanel between the goals and region information.

For the developers, the implementation of such a design was considered to be the strongest amongst the team members out of all our prototypes and was fairly easy to implement by outputting the player goal list into the text field JLabel. The list can be adapted with ease when the user completes or acquires a new goal by simply removing or adding a goal to the list. Due to our current coding ability and the time constraints, it seemed the best solution to get the fundamental component implemented as quickly as possible, which could then be adapted at a later date if any time was available and would depend on error. It could be implemented quickly through the available Java libraries for Swing that we knew of and was a task that could be allocated to one developer without the need of assistance, allowing our team to distribute resources elsewhere so that everyone could work in parallel and on separate tasks.

**Scoring**

The scoring feature is a required feature under the goal game component, but it must not be implemented at this development stage. A score will be calculated at the end of the turn the goal is completed on based on the achievement of the goal where longer or more complicated goals acquire more score than simpler or shorter goals. The reason for implementing scoring is to provide a competitive method between the two players and a way in which to persuade players to play again, a good reason for the clientele. It will also satisfy the requirements made for this stage of the assessment process. The scoring is shown in the top left hand corner (point 10) [Appendix F] with two separate scores: Player One (master) and Player Two (slave). This was placed here to allow consistency between all menus, i.e. make it always visible, and to enable the two players to have a quick reference point to take note of who is winning to plan their next moves. The colour scheme was chosen to improve the readability against the maps colours and to put a focus on the score as it will determine which of the two players wins and may help users strategise as to what to do next. The clientele backed the decision when we showed them the prototypes and suggested to use it to help indicate which player’s turn it is to stop any potential confusion occurring that could slow the game down and make the game less appealing to play again. This was, however, implemented differently and is explained in the turns section.

From a developer’s perspective, the maximum scoring would be implemented using Dijkstra’s algorithm to calculate the shortest path between two cities based on all possible paths. If the user decided to take a longer path, the score would not be increased based on the assumption that a longer path would reach the goal in a longer amount of time. The algorithm has already been implemented as it was easy to code as the team has created similar ones before for past projects. Future development teams would be able to utilise the algorithm in the next development stages for the scoring system.

**Store**

The store is an additional feature to manage the resources within the game. Two resources, gold and metal, are earnt per turn that can be used to purchase more resources to be placed in the player’s inventory. Unfortunately, due to time constraints, the store has not been implemented by the development team, but has a position ready on the GUI to be added when possible. Resources and time needed to be distributed to the fundamental requirements for this assessment instead as they had not been functioning due to bugs and were more important to be completed. This may have been due to a lack of coding experience, therefore taking longer than it was originally forecasted to implement the other features. Instead, the resources are now randomly generated and placed into a player’s inventory. The reason behind the store not being finished is that the store contained far too many bugs in the code to function correctly without any bugs and insufficient planning was considered to take into account the amount of testing and fixes required. The store is also not necessary to meet the requirements of the clientele, however, it would have been a good addition to the game as a selling point. The code has been left within the program as another team may wish to try and implement the store. In the future, it is advisable to try to forecast in more detail the costs, resources and time available to the team. From a clientele’s viewpoint, this is a loss as a selling point of their game. Not enough time was given to the development team to implement the store and it’s GUI

as the game had to be built from scratch, however, they can agree that their other requirements have been met at this stage. The next stage of development can implement this solution.

**Junctions**

Junctions (point 14) [Appendix F] are a form of city that has a location (node) that is connected to the train network (node connected to weighted edges) and can be travelled to as part of a journey (path). A junction, however, can not be set as a destination point in our game. To comply with the requirements, there must be at least two junctions and there must be an obstacle related to the junction, which does not have to be implemented at this stage. Junctions are shown in the same style as cities, except they do not have a name tag associated with them. This is to help the user recognise that it is a reachable destination, but will not be linked with any goal. From the client's point of view, having junctions would add another dimension to the gameplay as extra routes could be established for the player to choose, adding to the idea of multiple replays without the user losing interest in the game. It would also provide the potential to add other dynamics to the game such as resources that affect junctions. From the development team's point of view, having junctions would not be time consuming to implement as the underlying graph structure would already be in place from implementing cities. The difference is that the junction does not need to be labelled on the map, therefore the task should be relatively resource light. Only the developer who worked on the graph has to be allocated to the task and more resources and time can be deployed elsewhere. The development team would also need to implement this to comply with the requirement of having at least two junction, however, implementing the code behind it in a maintainable and readable fashion would allow more junctions to be added to the game should it be requested by the clientele.

**Routings**

The routings (point 16) [Appendix F] are a fundamental component of the game components requested by the clientele and are displayed visually on a level above the map to enable users to still see the map for a sense of gameplay. They are stored in the system as part of a weighted graph, storing each edge used and its weight as this was the best way to represent a journey. Its weights can be used to calculate a score when required in the next assessment, adding to the program's maintainability and readability. The GUI specifies a routing by clicking on the player's train (becomes highlighted) and then clicking on an adjacent city (node) to the train. The rail track (arc) between the two cities turn white instead of staying black to help users determine the journey that their train will take. These white rail tracks display a white number (with red outline) to specify to the user the order of the rail tracks in the journey. The idea here is to add clarity for the player to understand which train they have selected and the route in which it will take, otherwise the player might be at loss as to where their train is going. This would reduce the fun factor of the game and hence reduce the potential of a user replaying the game, which is not what the clientele would want. The colour style was chosen to contrast against the map's colour scheme and helps to determine the playing board for the user, which was supported by the clientele through the prototype. The journey can have a destination removed by clicking on the most recently added city, which will revert the rail track to its original black colour. The development team prioritised this task and allocated as many resources and as much time as possible to the completion of the routing system to ensure the requirement was met by the deadline due to requiring some additional training and time to test it.

**Train**

The trains (point 15) [Appendix F] are a necessary component to the game. They are sprites used by the player to travel from one city (a node) to another via a railway link (an edge). The requirements specification, as agreed by the clientele, states that a train will be used to visually show the progression of each player on a goal. As each turn passes, the train moves along the rail network until it reaches its final destination. The train will progressively move along its specified route to show the progression of the player to their goal visually by changing the x and y coordinates appropriately. Trains are placed onto the map at a random initial city to provide a unique experience each time the game is initialised, adding to the potential of a user replaying the system. The idea was also requested by the clientele when the team suggested it.

Showing the progression of goals on the GUI was a design decision that seemed to be the best solution as users would have a better game experience when they can visually see how well they are performing, rather than reading

about it. The resources and time required to implement the trains would not be extensive as the game requires a , with support available through online sources and books if required.

Additionally, the team decided to implement a colour coding system for the trains. This was shown to the clientele, in which they agreed it would be a useful addition. By following a classic colour coding convention of green for friendly trains and red for opponent trains, a user will be able to quickly identify which are their trains and which are their opponent's. This will help the user when using resources and powerups to aid them in using the resource on the correct train. For example, when the player has a speed boost and they want to use it on their own train and not their opponents. The colour coding system is unique to the player, i.e. friendly trains are green on Player One's screen but Player Two's screen will show them as red, and occurs in the turn label too, to identify whose turn it is. From a developer point of view, the implementation of such a feature was not hard based on the old and new experience of the team, hence the team only needed to allocate an individual to the task and allowed us to distribute our resources elsewhere. As for the client, they may now see this as another unique selling point to their game.

**Turns**

As the game is required to be turn-based, being able to alternate which player is playing is a vital requirement. To progress towards the completion of a goal in the game, players take in turns and alternate as to who is using the game board. The solution implemented is networked, using a game server that contains a master (Player One) and slave (Player Two). The server checks whether it has received information from a player as to their desired move. If it has not, then the player is still in their current turn. Once it has received the player information, the server executes their given action and passes control of the game to the other player, where the cycle repeats. This was the suggested method to take as the server effectively clarified whose turn it is by the data in which it received, which could then be output in some form to the GUI. In the requirements specification, the development team discussed implementing a network solution.

The network solution was chosen because it would enable two players to connect with each other over a network, enabling them to not be on the same machine. It was also programmable as one developer had experience in creating a game server before that utilised Java's libraries, hence no additional training would be required and only one team member would need to be allocated to the task allowing the resources to be distributed evenly. It is a complex system to initialise, however, no adjustments need to be made to it. The idea was suggested to the client and since then has been pushed for to enable the game to have a wider audience range and increase the replay potential by allowing users to compete against players all over the world and not just those locally.

A player's turn is indicated by a colour code system. The colour code system is shown in use in the top-left hand corner with the End Turn button (point 2) [Appendix F] and right hand side with the Player Turn Indicator text (point 1) [Appendix F]. The colour code system deployed is similar to the train component, enabling the text to change colour to indicate whose turn it might be (green for yourself and red for your opponent). The text also helps to the clarity of addressing this issue and is of a font size large enough to stop the user having to zoom in or out throughout the game. This is because it can be a nuisance to sort the zoom, if altered, to fit everything in one screen on a visual display unit. From the development team's perspective, the Player Turn Indicator text and the End Turn button were not hard to implement based on our coding ability and provides a computationally inexpensive way for providing an indicator and method in which the turn progression code can be executed.