1 Initial Approach

1.1 Game Selection

After viewing the groups' presentations, we each put forward our first and second choices and justifications. We chose DUS's game as it was a variant on the FVS code with which we were by then reasonably familiar with. Furthermore they had implemented all the required features of assessment 3 but had not made any major overhauls to the code, thus making it easier for us to modify.

1.2 Play-testing

Whilst the key goal of this assessment was to focus on the requirements change, the game was still played multiple times by members of the team in order to check for bugs and any improvements that we felt would aid the gameplay.

2 Implementation Planning

It was required in this assessment to implement a replay feature and track modification. This did not require further evaluation of DUS's requirements as these were new requirements. However naturally time was spent examining the code in order to decide the best methods in which to implement these new features in line with the existing architecture.

2.1 Replay

In order to implement the Replay system, we first had to choose how to structure our implementation. The game was first broken down into its basic player actions to identify what must be "recorded" in order to replay these actions for the user. Below are all of the possible events that may happen in a game:

- Player receives a goal
- Player receives a resource
- Player deploys a train
- Player uses a power-up
- Player routes their train
- Player ends their turn
- Player drops a resource
- Player drops a goal
- Connection gets randomly blocked

Every other aspect of the game, such as goal completion, is computed dynamically. This meant if we stored the above actions we could replay them on a fresh instance of a game and get the same result. With this information we formulated ideas for 2 possible implementations.

Click Storing

All of the games controls are taken from the mouse, therefore every click on a UI element could be stored and then, when required, these stored clicks could then be replayed. This would replay a game exactly as it was played.

Event Storing

Conversely instead of storing the click, the event that occurred because of the click could be stored e.g. storing "Train X was deployed at station Y" instead of "Train button was clicked" followed by "Station button was clicked".

Since we would have to record certain non-click actions with either system, such as giving a player a goal, we felt it made sense to store every event that occurs.

2.2 Track Modification

In order to implement the addition and modification of track, we first had to identify exactly what we wanted to allow the players to do, and how frequently they could do it. We decided on the following points:

- Players can add or remove track. This gives the game greater strategic depth, and allows players to aid their own progress or hinder their opponent's.
- Adding or removing will be used like a power-up / resource. This allows us to keep our UI simple and keep the game easy to understand.
- Players have a maximum distance to add track. This stops huge, unrealistic tracks across the map which look ugly and would detract from gameplay.
- Players receive the ability to modify track upon completing a goal. This was the most satisfactory way of awarding this ability, as all other methods we could think of were too random for such a useful ability.
- Players choose whether to add or remove track, but not both. As players were receiving the ability on completing a goal, we did not want to stack the odds in favour of the player who completes more goals; leading to an even bigger advantage.

Next we had to decide on how we were actually going to implement this feature in the game. As we had decided to let it work in the same style as power-ups, it makes sense to implement in the same manner as the other existing power-ups.

The power-ups currently implemented in the game (which include Engineer and Skip) are represented in the game by a class in *gameLogic.resource*. This holds the information required by the power-up; such as starting and ending stations. However, the other types of power-up are granted randomly like trains, using the *resourceManager* class. Since we wanted players to receive them as a reward, we simply add the resource when the player completes a goal, which is already checked for in *GoalManager*.

2.3 Additional Modifications / Observations

The following observations were noted when initially play-testing the game as things that could be implemented / improved upon depending upon ease of implementation and time constraints. Our main priority was replay and track modification. Those that were implemented are covered in more detail within section 3 of this document. Those that were ultimately not implemented include a brief explanation of why this was the case below.

- DUS's game lacked a backstory or "theme" so it was decided to rebrand the game as "Snakes on a TaxE" (Snakes on a Train across Europe) – an animal themed game in which players must transport animals across Europe. This was chosen as we felt it was more important to spend time ensuring the track modification and replay features were correctly implemented than attempting to rebrand the entire game from scratch.
- DUS's game had no in game instructions: we felt the addition of these would make it easier for users to pick up the game more quickly
- There was no way for users to select how long they wish to play the game for
- Only the current players' score is viewable at any one time. A user may wish to see both to compare.

- The game lacked any sound.
- The UI was fairly bland: all buttons / controls are simply different shades of grey
- When a goal offered bonus points the specific train was not always available within the players' resources.
  NOT IMPLEMENTED – we chose for this to remain giving players the chance to play tactically to get their bonus points e.g. by dropping unwanted train types and waiting for one which would provide them with a bonus.
- Viewing the route of train informs the user how many turns the route will take – this does not take into account obstacles.
  NOT IMPLEMENTED – this was deemed an obvious feature as obstacles can be unexpected thus the effect they have on timing need not be accounted for in advance

3 Formalised New Requirements

Upon receiving the client's additional new requirements for the software and deciding how we wished to implement them, the requirements were formalised as follows. No traceability matrix has been included within this report but all requirements below are deemed to have been met.

3.1 Turn Recording

Events within the game must be recorded so that the user may "replay" the events and view them at a later time in the game. This includes players receiving a goal or resource; deploying a train or power-up; dropping a goal or resource; ending their turn; routing a train; and any connection blocks that randomly occur.

3.2 Turn Replay

Users will be able replay recorded actions and view the movements on screen. The replay mode must be able to be paused / exited and users will be able to replay actions either in real time or faster than real time.

3.3 Track Addition / Removal

Players will receive a power-up allowing them to add or remove a piece of track upon completing a goal. There will be a limit on the maximum distance track may be added and track currently occupied by a train may not be removed.

4 Modifications

4.1 Replay Feature

Description
The specification required the ability to replay the game as it being played by the players. This feature is the implementation of that requirement. As events occur during a game they are recorded by the replay manager and can be played back in real time or faster than real time by skipping the player thinking time. The replay can also be paused or restarted, or even exited back to the main game. Entering the replay again will start from where it left off.

Implementation – Back-end
ReplayManager - The Replay Manager is in charge of anything related to replays. It keeps track of every event that occurs in a game and when replaying keeps track of what actions have already been

replayed, how long it has been since the replay started etc. It also keeps track of the speed at which the replay is being played at. For a full breakdown of the methods involved with the Replay Manager please refer to the javadocs.

Action (and its subclasses) - Action is an abstract class that is the foundation of every event that occurs during a game. Its core abstract method is '*play()*' which is called when that action is replayed. Every possible event that can occur in the game has its own subclass of Action, which includes everything from giving goals to players to placing and routing trains.

Context - The context class was changed slightly with this feature implementation. There is now a second game instance called *replayingGame*. Players cannot play on this instance, it is used as the fresh game to start replaying events on to. It is also where you enter and exit the replay from, which is here because the Context class has access to the main game instance which we need when switching between the main game and the replay game.

The method *Context.getGameLogic()* previously returned the logic for the main game but it now returns the logic for the current game on display, either the main game, or the replay game if the game is in replay mode. This change allows the majority of the system to be abstracted from the replay system. The system just works with the logic that is given to them, it doesn't know if the game is in replay mode or not.

GameScreen - The *render()* method was changed here as we want the system to replay actions at the correct time when replaying and record actions when not replaying. In addition to this the player initialisation code was moved to *GameScreen* as it needed access to the replay manager to record the first goals and resources given to players.

Implementation – Front-end
TopBarController - With the implementation of the replay system, the user needed some way of controlling the replay. When in normal playing mode, the top bar has a button that allows the user to enter and start the replay. Then when in replay mode the top bar gives the user more minute control over the replay. The user is able to:

- Exit the replay and go back to the main game
- Restart the replay from the beginning
- Pause/Play the replay
- Skip Thinking Time - This option plays actions immediately on after the other and skips the time users spent thinking about what action to make in the main game.

Challenges
The specification states that the replay system needs to have a way of replaying the game at quicker than real time. However since all train movement is calculated when you declare the route, if you changed the playback speed, say to double the speed, the trains that already have their routes declared would not move any faster. We solved this issue by implementing the 'skip thinking time' option, which plays actions one after the other without waiting for the correct timestamp. This means trains all move at real time speed, but the replay is still quicker than real time since the time the players spent thinking about what they want to do is skipped.

In the code that controls the map, there is a method that checks if a connection is blocked. If it cannot find the connection is just returns true. Since we now have 2 maps, one from each game instance, there were issues with connections being checked between maps and this return true caused the system to think that every connection was blocked when in replay mode. 3 days of development time was lost looking for this bug that was hidden deep inside the map code. This bug was fixed by changing the method that checks for blocked connections to check by comparing the names of the stations rather than comparing the instances of the stations.

In order to have the game wide access that the replay manager needs, we put our instance of *ReplayManager* in Game, where our game logic is stored. This means that each instance of Game has an instance of *ReplayManager*. Since we have an instance of Game to project our replay onto, the replay game technically has a Replay Manager. It is, however, never used and there are measures in place to prevent any actions performed during a replay being stored for future replays. Ideally the *ReplayManager* instance would've been placed in the Context class, but since we need access to certain parts of Game this couldn't happen without rewriting some core architecture code.

Overview
Practically, this means that there are now 2 instances of the game: one in which the players actually play and one that is used for replaying events, stored in our Context class where there was previously only one. The problem with this was that whenever the game would want to do something, it would have to check what instance of the game is currently in use. It would have been challenging and unpleasant to sort through the code and perform a check at every event.

Instead of this we changed the *Context.getGameLogic()* method to do the checking for us. If we are not in replay mode, return the *gamelogic* for the main game, else return the replay game logic. This meant we actually had to make very few changes to our code. Users could switch between normal and replay mode all they want and when the *GameScreen* renders and asks the context class for the current game logic, it just returns the correct logic for them. This abstracts most of the system from the replays. The majority of the system doesn't know if the game is being replayed, it just works with the information it is given. The goal manager doesn't know what game it is currently working with, it just knows that a train just arrived a station and it needs to check if a goal has been completed or not.

When an event occurs in the game that needs to be recorded, an instance of the relevant Action class is created, which contains the details of what happened, including a timestamp of when it happened. Then that action is stored in the *ReplayManager's* list of actions.

When the user starts a replay, it checks the timestamp of the next action every frame, and if that amount of time has passed since the replay was started, the next action is replayed. The user has the option to pause the replay which stops the replay manager from performing any actions. Users can also enable a setting called "Skip Thinking Time". When enabled, this feature skips all of the time it took the players to take their turns and just replays every action one immediately after the other.

We decided against implementing a variable playback feature, where users could view a replay at 2 times the speed for example, because of a limitation with the underlying game architecture. When routes for trains are declared, all of the trains' movements are calculated at that moment. This means if the route was declared when the replay was at normal speed, even if the user sped up to 2 times speed, the trains would not change their speeds because they had been pre calculated. Solving this issue would have involved rewriting a lot of the underlying train movement architecture and we felt that our time would be better spent polishing other aspects of the game.

Testing Reference(s)
PIK System Tests 7, 8, 14
PIK Unit Test 1

4.2 Addition and Removal of Connections

Description

The specification required the ability to dynamically add and remove track. This feature is the implementation of that requirement. If a player completes a goal, they receive a bonus reward called 'modifier'. This allows for the addition of one track segment (of limited length) or the removal of a piece of pre-existing track. This means that the map changes dynamically throughout the game. We believe that giving this resource to the players as a goal is the best method, as it is clearly extremely useful and awarding it randomly would make the game even more luck-based.

Implementation – Back-end

gameLogic.resource.Modifier

This new class is very similar to the equivalent ones for other power-ups. It contains the two stations which the track is to be created or removed between, and this allows them to be accessed easily by other functions.

gameLogic.player.GoalManager

The *GoalManager* handles the goals for each player and checks for their completion. We modified it so that when a player completes a goal they are awarded a new modifier resource using *player.addResource()* within the *trainArrived* function.

gameLogic.player.Map

We implemented a new function, *doesConnectionExist,* which accepts two stations as inputs and returns True if they are connected and False if they are not. This is used when testing whether or not it is possible for a player to remove a piece of track in *DialogButtonClicked*. We also added two methods *addConnection* and *removeConnection. addConnection* accepts either two stations or two station names as strings and returns a new connection between them. *removeConnection* takes a connection and removes it from the map.

Implementation – Front-end

Button.java

This file contains an enumeration with all the text which is placed on buttons throughout the game. We updated it with our new dialogs (ADD_PLACE, REMOVE_PLACE, MODIFIER_DROP).

DialogButtonClicked.java

This file handles the actions taken when a power-up is used. This is done with a large switch statement, handling each possible dialog button, inside the clicked method. We added the new functionalities to this which we defined in Button.java, which were done in a very similar way to the existing power-ups.

clickListener.ModifierClicked

This new class is very similar to the ones already defined for the other power-ups. It is a click listener, which detects when the modifier is selected from the resources in the UI. It also handles displaying a message in the top bar using the enter method, telling the player that they have activated a modifier.

resourceController

This class handles the drawing of the player's resources in the UI. In the *drawPlayerResources* method, there is a chain of if statements handling the different resource types. We updated this with 'modifier' and the appropriate variables.

Challenges
This was easy to implement as it was really similar to the existing Obstacle resource, and since connections are rendered every frame anyway, all we have to do to add or remove one is to update the array containing them.

Overview
The actual implementation of the modifier was created in the *DialogButtonClicked* file. This was because we wanted to keep it consistent with the other power-ups already in the game. It is implemented in a very similar fashion to the other power-ups which require a start and end point, however, it checks for maximum distance between the two stations. We coded the addition and removal of track as two separate parts so that each one could be easily modified and to keep it consistent with the existing code.

We ultimately decided against implementing a visual demonstration of the maximum distance, because the allowed range is large enough that in most cases players will not realise it exists. It only comes into play when a player tries to make large and visually unappealing tracks, in which case a simple message is enough to deter players from trying it again.

The GUI is implemented in the same fashion as the rest of the game. It has two main parts; the button to activate the modifier, and the pop-up dialog messages. Both are handled in the same way as existing power-ups. More details can be found in the implementation section below.

Overall, we tried to implement the addition and removal of track as a fun but strategic resource which players may receive as a reward. We wanted it to be a special type of power-up, and so kept the implementation as consistent as possible in line with the existing power-ups already implemented in the game.

Testing Reference(s)
PIK System Tests 3, 4, 10
PIK Unit Test 2

4.3 Start Screen Alterations: Instructions Screen, Turn Selection, Theme addition

Description
Previously there were no instructions available in-game and no option for the user to control how long they wished the game to last. This required adding buttons to perform these actions to the start screen and was able to be implemented quickly by using the code from our previous assessment. This also aided in re-theming the game by using our original animal-themed start screen (altered to fit the slightly smaller window dimensions in DUS's game) and then redesigning the actual content of the instructions page in order to make it appropriate for this game.

Implementation - Back-end n/a

Implementation – Front-end
Buttons were added into *MainMenuScreen* for "How to Play" and options of 30, 40 or 50 turns to select. *gameLogic.Game* had a function to edit the number of turns added to it.

The class *InstructionsScreen* was used for the instructions screen, which is just a background image. It also contains a "back" button which is located in the top left hand corner to take players back to the main menu screen.

Challenges
There were no major challenges associated with this modification.

Testing Reference(s)
PIK System Tests 15, 16

4.4 GUI alterations: Colour Coded Buttons, Score Display

Description

Only the current player's score was previously viewable at any one time. Now both scores are viewable. The GUI was additionally very bland with some of the white text difficult to read on the light grey background.

Implementation - Back-end n/a

Implementation – Front-end
The button colours in *GoalController* and *ResourceController* were updated so that goals have a teal background by default (making the white text easier to read) and resource buttons are colour coded for easy identification:
Train – red, Obstacle – purple, Modifier – blue, Skip – green, Engineer – yellow
Previously only the current player's score was displayed on their turn. We decided to change this so that both scores are visible at all stages of the game. This was done by slightly modifying *playerHeader* within *goalController* which was used to display the score. Now the method retrieves player 1 using *getAllPlayers().get(0)* and then returns player1.getScore(). The slight change to make the float into an integer was done by the previous group so went unchanged. This method was copied to resemble player 2, at which point *drawHeaderText()* needed to be updated to display both players' scores. Two simple lines which draw the values returned by the two methods with some formatting means the scores are now displayed at the top right of the map.

Challenges
Originally the GUI was going to be redesigned so that the map was completely viewable and not covered by the goal and resource buttons. This however proved challenging due to issues such as the need to alter the screen window dimensions and how mouse-overs are used within the game. Ultimately it was decided a redesign was unnecessary and that brightening up the map with some more colours would suffice.

Testing Reference(s)
PIK System Test 1

4.5 Sound

Description
To add a variety and uniqueness to our game it was decided to add a small elevator-style music loop that players have the option to turn on or off through a mute button which would be displayed in game. The status of the sound is signified by the colour of the mute button; green for playing and red for muted.

Implementation - Back End
Sound Manager - A class to be in charge of the loading and playing of the music loop. It is fitted with methods to play, mute and unmute the sound.

Implementation - Front End
GameScreen was edited so that a button was created in game to mute/unmute the sound at any point during play. The button is created in the *addMuteButton()* method and is just a simple text button that switches between the two colours every time it is clicked, and calls the *muteBGMusic()* and *unmuteBGMusic()* methods from *SoundManager* alternatively.

Challenges
There were no major challenges associated with this modification.

Testing Reference(s)
PIK System Test 17

5 Software Engineering Approaches

5.1 Agile Development

Once again, we chose to use an agile approach for this stage of development as it has been used for all stages of the SEPR project and we felt that maintaining this approach would provide continuity and be the easiest approach to take.

5.2 Team Communication

The team met face-to-face every week during term time to plan for the week ahead and discuss progress in implementing features and making design choices, as well as any problems that had arisen. Significantly more communication was also done online via Facebook and Skype in order to plan meetings and ensure that everyone was kept updated.

5.3 Coding Style

A major shortfall in previous assessments for our team has been a lack of commenting / not commenting code as comprehensively as perhaps necessary. More time has therefore been spent ensuring code is readable and well commented and checking over other team members' code this assessment. All of the new changes have been given javadoc comments and also comments within methods explaining more thoroughly what certain pieces of code are doing.

We also aimed to keep the replay system as abstract as possible from the rest of the system as, if this was done successfully, we would be able to just give the system either main game logic or the replay game logic and it would handle it in the same manner. The change to *Context.getGameLogic()* (described in 3.2) enabled this. Most of the system has remained unchanged from the last cycle, it is just handed different logic from the different games. For example, the code that checks if a goal has been completed has changed, but if we are in the main game we give it information about trains in the main game, and if we are in the replay, we give it information about trains in the replay game. This should keep the code not only readable, but very scalable too.

5.4 Software Configuration Management & Change Management

After choosing the project we made sure to understand the code; the way it was structured and how functionality was delivered. We did this by going over the code and reading any comments, as well as adding our own if anything was unclear. We also looked to improve the code. We tried to find any bugs, code duplication, sloppy naming and other code smells to remedy.

It was decided that no corrective changes were needed as DUS had done extensive testing and we failed to find any errors within their game. Similarly with deletive and perfective changes: we were overall happy with their features and did not want them removing (see exact desired alterations in section 2.3) and certainly didn't want to have to modify the underlying architecture. We discussed how any additive changes could be made, such as the GUI changes and adding an instructions screen. This was in addition to the extra functionality needed after the requirements change in order to satisfy the customer.

DUS's original requirements have not been discussed within this document as they were deemed to have been met with no requirement for our team to work further on satisfying them: the primary focus of this assessment was satisfying the new requirements.

With multiple people working on changing the same software, it was important to use version control to help with coordination. We continued to use Git and made sure that all team members pushed to and pulled from the central repository regularly so we were all up to date with changes made.

It was equally imperative that we made sure these code changes fit into the architecture of the code and documented well so it would be understandable in the future. Naturally this also included ensuring that all of the associated test-documents, reports, new requirements, user manual and website were updated in time for the release of this "final" version of our software to the external client.