Project Extension Report

Introduction

The general approach to extending the project has been turning it from an effective implementation of the base game functionality into a game which actually challenges and engages the players. The game is now enjoyable to play. We have endeavoured to increase variety, competitiveness, and usability within the game to ensure that the gameplay experience is as smooth and entertaining as possible.

Justification of Change

Some justifications of change may refer to the User and/or System Requirements. These are the requirements specified by Team HEC in their original project Requirements Document. For convenience, these requirements have been added as an Appendix the end of this report.

Architecture Justifications

The original project that we selected from Team HEC has a great source code foundation to build on as it utilises the Model-View-Controller architectural pattern for implementing user interfaces, we have attempted to extend the MVC principles, where relevant. This means that our game logic is effectively decoupled from the user interface.

GUI Justifications

In extending the functionality of the game, we have aimed not to clutter the screen too much with additional buttons. As more and more information is displayed to the user, it creates an unintuitive interface. This could cause confusion as as the players may not understand all the different - yet potentially similar -UI elements. Therefore each additional UI feature was carefully vetted for usefulness before we decided to add it. Each UI change or addition has been appropriately justified and recorded in this document.

The project we picked up consistent and well structured design. In the design of any new GUI elements, we have aimed to keep consistency with the original project's design methodologies.

Score

Justification of Change

In order to make the game more competitive we've added a score counter for each user. This gives the game a convenient end goal as users compete for the highest overall score. At the start of this assessment, the functionality for players to have a score was already implemented. This value was stored as a variable within the Player class, we considered the way it was previously implemented to be inconsistent with the rest of the game's resource system so a Score class was created.

User.GP.7.1	User **Must** be able to score points, such that the player with the most points will win the game.
User.GP.2.3	Goals **Must** support at least 10 different goals.
User.GP.2.4	Each Goal **Must** have an associated number of points a player can score for completing it
User.UI.3	**Must** clearly show both players' scores.

Architecture Justifications

Previously there was no way for players to be given score in the game. Players now receive score once goals have been completed. This change was made to bring the game in line with the brief for this assessment. The score was implemented as a class which inherits from the Resource class. This was chosen so that additional functionality can be added to the score system with relative ease, this follows the style in which other resources were implemented, which keeps the architecture style consistent.

The methods by which score is increased exist in the Player class. This is because any score attributes are owned by each instance of player and, also, so that it is clear which player's score is being increased. The addition of the score system makes the game much more competitive, as the players can measure how they are doing, when compared to the other player.

GUI Justifications

With regards to the GUI, the score is displayed at the centre of the top of the screen, during the game, to make sure it is easily visible to the players. This updates whenever a goal is completed and at the end of a turn, so that players always know their score.

Relevant Testing

See test plan pages 32-33 for black box testing of the score system.

Shop

Justification of Change

Over the course of this assessment, several features have been added that involve the player spending money in order to achieve some function. These features were added to the shop class as this helped to maintain the current structure of program.

User.GP.3.1	Players **Must** be able to obtain resources.
User.GP.13	There **Must** be an in game currency.
User.GP.14	Users **Should** be able to purchase resources.

Architecture Justifications

Over the course of this assessment, several functions have been added to the shop class. The functions added were buyTrain, repairStation and upgradeStation. These functions were all added to the shop class due to the fact that they consist of the player spending money and then some action being performed. This keeps all the functions that involve spending money in the shop to help maintain the current structure of the program.

Functions to support this have also been added to the player class and the station class. These classes deal with changing the resources of the player and the state of the station.

GUI Justifications

At the start of the assessment, the shop only had the functionality to buy and sell cards and fuel. Over the course of this assessment, functions have been added to the Shop class and the Player class to provide the backend logic for buying trains. The shop user interface has had the required buttons added to it to be able to buy trains to it, however these are not being displayed due to the underlying design of the shop GUI.

The shop GUI uses the same buttons for both buying and selling. This would not work well for buying and selling trains. The ideal solution for this would be to redesign the shop GUI into two classes, one for selling and one for buying. This was not done, due to the fact that we prioritised other features and extensions over the shop. Therefore, we did not have the time to make the changes we felt would have been necessary to ensure the shop would be extensible by another team.

Unfortunately, this decision has lead to the shop GUI becoming more decentralised than before, as the user interface elements for both buying and repairing stations

being added elsewhere. We decided to add both of them to the station info panel as we felt like this was the most intuitive place for the the player to find them as they are directly related to a particular station. The backend functionality still exists as a single class, which allows us to keep all the methods relating to the buying and selling of resources in one place, which helps with readability for future programmers.

Relevant Testing

See test plan pages 20-21 for unit testing of the shop.

Faults

Justification of Change

We have added faults to make the game more dynamic and engaging. This also fulfils the assessment requirements, as well as several of the specified requirements, detailed below.

Func.OD.4.2	Game **SHOULD** alert players when a random event occurs.
USER.GP.6.3	There **MUST** be at least two obstacles in the game.
USER.UI.10	**MUST** display hazards on screen.

Architecture Justifications

At the start of this assessment there was no implementation of obstacles within the game. We have since implemented two kinds of fault- stations, these are both junction faults that occur on stations around the map, some of these can be fixed whereas some are permanent. The faults are generated in a method in the `WorldMap` class which iterates over the list of stations, and decides whether or not they become broken.

When a station is broken, a boolean variable `isFaulty` in the station object is set to true. We implemented faults as a boolean in Station as it didn't really make sense to have a 'Fault' object when they can be represented more simply. A fault on a station means that any train which tries to move to it will be returned to the last station it went through. This is done in the `Route` class by checking whether the station is faulty each time a train moves, within the `Route` class' `update()` method.

The decision as to whether a particular station will become faulty is determined by a probability that is dependant on a station's level (`private int stationFaultLevel`). All stations start at level 0, which means they will have a 1% chance of becoming faulty on a particular turn. As stations are upgraded, this probability is reduced down to 0.2% in decrements of 0.2% each, for levels 1 up to 4 (which is the maximum level). A player can upgrade a station they own at any point during the game, providing it is not faulty.

GUI Justifications

A faulty station is displayed to the user on the GUI as having a cross on it. The button is updated by calling the `updateButton()` method in the `Game_Map_Station` class whenever it becomes faulty or is repaired. This is an extended of the previous

`changeOwner()` method from Team HEC's original implementation. The name has been changed to reflect the additional functionality.

Station information displays in the GUI have had a repair button added to them. A faulty station can be fixed by clicking on it, which will remove money from the player, and simply reset the boolean `isFaulty` attribute of a station to false.

Faulty stations appear identical on the map whether they are repairable or not- we did this because, for the most part, players will not be bothered about whether a station is repairable unless they particularly need to pass through it. When you try to repair a permanently damaged station, a warning message appears to alert the user that it can't be done.

At present, the station's level is not displayed to the user. We had planned to add this as a number in the centre of the station's circle, however, due to time constraints in this assessment, we have not been able to.

Relevant Testing

See test plan pages 27-31 for black box testing of the fault system.

Goals

Justification of Change

We extended the goals to make the game more varied and enjoyable to the players as the new goals help to prevent the game from becoming too repetitive. The added goals also satisfy one of the customer's requirements that the game must have both qualitative and quantitative goals.

User.GP.2.1	Players **Must** be provided with Goals
User.GP.2.3	Game **Must** support at least 10 different goals
User.GP.2.4	Each goal **Must** have an associated number of points a player can score for completing it.
User.GP.2.5	Goals **Must** be completable
User.GP.2.6	Users **Must** be able to accept or reject goals
User.GP.7.2	User's score **Must** be based on their achievement of goals
User.UI.1	The user's current goals **Must** clearly be shown

Architecture Justifications

A lot of the extensions made to the Goal system could be made fairly quickly as a lot of the pathfinding logic and completion checks were either in place or partially in place. Extensions to the Goal Factory class were made to allow the generation of quantifiable goals, and ensure that any goal generated was completable. Using the implementation of Dijkstra's algorithm to ensure any time limits were sensible helps to prevent player frustration at being unable to complete goals. Originally the goal superclass had a method that tested for goal completion whenever a train passed a station, this method has been edited so that instead of testing for a start station and an end station being passed, the method additionally tests for a via station being passed, the correct cargo was being transported and it ensures that the goal is still within it's allocated time limit if it has one. These extra tests are used where applicable depending on the goal type.

Four new types of goal were introduced, all of which extend the SpecialGoal class. Cargo Goals which required you to take a specific cargo (currently Diamonds) from one station to another (your train is slowed down by 10% due to the extra weight of the cargo). Route Goals require you to take a specific route (i.e Travel from station A to station B via station C). Timed Goals use Dijkstra's algorithm to calculate an appropriate time limit for the goal to be completed in. Combo Goals are a combination of Route and Timed goals (i.e Travel from station A to station B via station C in X turns). A new method goalFailed has also been added to the Goal superclass, which displays a message to the user upon failing a goal. All of the new

goal types reward the player with a larger sum of money and more points than the standard goals due to their extra requirements. These goals were introduced with the aim of increasing variety in gameplay to keep the game interesting for the players by encouraging them to try different strategies.

GUI Justifications

The GUI has been changed to display additional information about each of the special goals on the goal icons, such as the turn limit on a timed goal. This was a challenge as the way the icons had been written did not lend itself well to being extended easily. Given more time we would have changed the way the information is displayed in order to allow the simple addition of more goal information, such as positioning individual labels or creating a list of labels that could be appended to.

Relevant testing

See test plan pages 4-15 for unit tests of the goal system, and pages 16-20 for black box testing of the goal system.

Minor changes, overarching challenges and approaches

Minor Changes

Throughout the course of this project, we had to make many minor changes to this project. Many of these changes were minor bug fixes to ensure that the game functions correctly and is more usable and intuitive for a new user.

We added a system which gives players money at the start of each turn for each station they own, which ensures that players do not run out of money if they can't complete goals in time, which we considered to be frustrating for the player.

We removed a bug which crashed the game if you clicked the abort button whilst not having a route in the routing screen. We also removed a bug which continued to show the routing blips on the GUI after the routing screen had been closed.

We added the ability for the game to detect when it has reached the turn limit and then display who won, their score, and enable the player to close the game. Which prevents the players from playing the game forever.

(Func.SYS.11.1) Stations are now purchasable if you have a train on that station, then the station can be selected and the player is provided with an option to purchase the station, or upgrade it for a lower fault rate if the station is already owned.

(Func.SYS.11.2)(Func.SYS.11.7) Stations now provide resources of a specified kind and also gold every turn to the player that owns them.

(Func.SYS.13) System will quit if and only if the turn limit has been reached and the player has clicked to continue.

Overarching challenges

The project we chose was very large relative to other projects in the assessment, which meant that understanding the code in the initial stages of this assessment was a challenge. Due to the advanced nature of the project, and the limited timeframe we had to extend it, we had no time to restructure the code to our preferences, instead choosing to adopt the original authors' design style, which has meant coding has taken more time and effort, but is overall easier to comprehend.

Throughout the project, we have had problems with the lack of clear boundaries between classes. We have often found ourselves having to change multiple classes

to achieve a new piece of functionality, because it was unclear which classes were affected by the change. It has been a challenge adapting to a drastically different architecture style, and a different coding structure, as we have tried to maintain these standards for consistency to help with readability and expandability for future coders.

The GUI, while well made for the purposes of the last assessment, was written in such a way that extending upon it was difficult and time consuming, which has led to some initially planned GUI functionality not being added because of time constraints.

The project was extended using pair programming, as we found that this was a good way of eradicating bugs and errors, and ensuring it was always clear who was responsible for which sections of the project. We changed the pairs for this assessment to ensure that there was an even spread of coding skills across the pairs. We also continued with a scrum programming methodology.

The project was extended in line with Java standard coding practices, which was consistent with both the original authors' style, and also the style we adopted for the last assessment, which meant that we did not have to change practices.

The testing for the project was done in JUnit to keep consistency with the previous authors.

Relevant Testing

See test plan page 33-34 for black box testing of the game ending.

Appendix:

The following pages contain a full list of Team HEC's original requirements for reference.

A.1.1 Requirements and Specifications

A.1.1.1 Gameplay Requirement

User.GP.1	Game **MUST** be turn based.
User.GP.2.1	Players **MUST** be provided with goals.
User.GP.2.2	Goals **MUST** be based around sending trains from city to city.
User.GP.2.3	Game **MUST** support at least 10 different goals.
User.GP.2.4	Each goal **MUST** have an associated number of points a player can score for completing it.
User.GP.2.5	Goals **MUST** be completable.
User.GP.2.6	Users **MUST** be able to accept or reject goals.
User.GP.3.1	Players **MUST** be able to obtain resources.
User.GP.3.2	Players **MUST** be able to deploy resources.
User.GP.3.3	Game **MUST** have only 7 different types of deployable resource.
User.GP.3.4	Game **SHOULD** have a series of wild cards which cause random effects.
User.GP.4	Game **MUST** have random events which affect certain routes, cities or other gameplay features.
User.GP.5.1	Players **MUST** be able to send trains between different European cities (fictional or nonfictional).
User.GP.5.2	Players **MUST** be able to plan non-direct routes via other cities.
User.GP.5.3	Players **SHOULD** be able to abort routes. There **SHOULD** be a penalty for this action.
User.GP.5.4	Players **SHOULD** be able to halt and restart trains whilst on their routes.
User.GP.6.1	Game **MUST** include at least 5 different cities.
User.GP.6.2	There **SHOULD** be at least two junctions (i.e., train routes that intersect).
User.GP.6.3	There **MUST** be at least two obstacles in the game.
User.GP.7.1	User **MUST** be able to score points, such that the player with the most points will win the game.
User.GP.7.2	User's score **MUST** be based on their achievement of goals.
User.GP.8	The system **MUST** have a method in which the game ends and a winner is declared (or a draw is declared).
User.GP.9	The game **COULD** have multiple game modes.
User.GP.10	Game **MUST** support exactly two players on one computer.
User.GP.11.1	Game **COULD** support the purchasing and upgrading of stations to provide benefits to the owner. These benefits could be more train slots, charging a use fee when the other player

	passes through etc.
User.GP.11.2	Stations **COULD** belong to a Line providing benefits if a Player owns multiple stations on a Line.
User.GP.12.1	User **COULD** have the ability to upgrade trains to make them go faster, more efficient or support more carriages.
User.GP.12.2	Game **SHOULD** support more than one kind of train.
User.GP.13	There **MUST** be an in game currency.
User.GP.14	Users **SHOULD** be able to purchase resources.

A.1.1.2 UI Requirements

User.UI.1	The user's current goals **MUST** clearly be shown.
User.UI.2	The user **MUST** be able to track the progress of their trains.
User.UI.3	**MUST** clearly show both players' scores.
User.UI.4	**MUST** clearly differentiate between different player's trains.
User.UI.5	**COULD** display both players names.
User.UI.6	Users **MUST** have access to a start menu system to start games, load games and quit the program. This should be the first screen the users see.
User.UI.7	Users **MUST** have access to an in game menu system/ pause screen that allows user to save games and exit to the start screen.
User.UI.8	The start and pause screens **SHOULD** also feature controls for a preferences screen.
User.UI.9	**MUST** display trains on screen.
User.UI.10	**MUST** display hazards on screen.
User.UI.11	**MUST** display stations on screen.
User.UI.12	**MUST** display routes on screen.
User.UI.13	**MUST** display player resources on screen.

A.1.1.3 Input Data Requirements

Func.ID.1	Users **MUST** be able to use keyboard and mouse commands to control the game.
Func.ID.2	System **SHOULD** be able to take a save file and continue a previously played game.
Func.ID.3	User **MUST** be able to accept/ reject goals
Func.ID.4.1	System **MUST** accept routes provided by user and move trains accordingly.
Func.ID.4.2	System **MUST** abort train if prompted by the player
Func.ID.4.3	System **MUST** be able pause train if prompted by the player.
Func.ID.4.4	System **MUST** be able to resume a journey ar a player's request
Func.ID.5	Users **MUST** be able to tell the system to end their turn.
Func.ID.6	Users **MUST** be able to choose when to use their Wildcards

A.1.1.4 Output Data Requirements

Func.OD.1.1	The game **MUST** output the users' scores throughout the game.
Func.OD.1.2	The game **MUST** output the users' scores at the end of a game.
Func.OD.2	The game **MUST** update scores in real-time.
Func.OD.3	The game **MUST** display available items/resources in real-time throughout game.
Func.OD.4.1	Game **SHOULD** be able to represent generated events on the game map
Func.OD.4.2	Game **SHOULD** alert players when a random event occurs.
Func.OD.5	Game **SHOULD** be able to create a save file of a player's current game and output that file to the game's directory.
Func.OD.6	Game **COULD** output currently owned stations.
Func.OD.7	Game **MUST** have the ability to display users current goals.
Func.OD.8	Game **MUST** Clearly differentiate between players trains by using an appealing colour scheme.
Func.OD.9	Animation **COULD** be used represent train transitions.
Func.OD.10	Animation **COULD** be used to show events in the game.
Func.OD.11	Animation **COULD** be used in the UI for a more immersive game.
Func.OD.12	Animation **COULD** be used for gaining/losing/spending points and resources, to make the player aware of the change.
Func.OD.13	Animation **SHOULD** be used so players can track their train.
Func.OD.14	System **COULD** display players names.

Func.OD.15	System **SHOULD** have a start screen - displaying options to “start game”, “load game” and “quit game”
Func.OD.16	System **MUST** have an in game pause menu, displaying options for “save”, “quit” and “preferences”.
Func.OD.17	System **MUST** have a preferences screen for options such as sound
Func.OD.18	System **MUST** display stations on screen
Func.OD.19	System **MUST** display a player's current route

A.1.1.5 Functional System Requirements

Func.SYS.1	System **MUST** keep both players score.
Func.SYS.2.1	System **MUST** generate goals for players.
Func.SYS.2.2	System **COULD** Randomly generate goals for players.
Func.SYS.2.3	System **MUST** be able to monitor accepted goals
Func.SYS.2.4	System **MUST** be able to tell if a goal has been completed.
Func.SYS.2.5	System **SHOULD** have special goals which provide Wildcards as a reward.
Func.SYS.3.1	System **MUST** implement a turn based mode of operation.
Func.SYS.3.2	Players **MUST** only be able to move on their turn.
Func.SYS.3.3	System **MUST** be alerted at the end of a players turn
Func.SYS.3.4	System **MUST** monitor all actions taken during a players turn
Func.SYS.3.5	System **MUST** be able to declare an end to the game once the end-game condition has been met.
Func.SYS.4.1	System **MUST** keep track of players resources in real-time.
Func.SYS.4.2	System **MUST** support allocation of resources to trains, shops and stations.
Func.SYS.4.3	System **MUST** support allocation resources to a player.
Func.SYS.4.4	System **COULD** allocate resources to a player based on owned stations specialities.
Func.SYS.4.5	System **COULD** occasionally randomly allocate resources to players.
Func.SYS.4.6	System **MUST** support 4 fuel types as a deployable resource.
Func.SYS.4.7	System **MUST** be able to randomly assign a player a wildcard, if the player requires one. (From a goal etc)
Func.SYS.4.8	Wildcards **MUST** produce a strong effect on gameplay features. For instance, a wildcard could allow another player to have another resource, or teleportation.
Func.SYS.4.9	Player **SHOULD** be able to purchase resources.
Func.SYS.4.10	There **MUST** be a currency resource such as gold.

Func.SYS.4.11	Carriages **SHOULD** be a resource deployable to trains
Func.SYS.5.1	System **MUST** be able to produce events that have an impact on the current game
Func.SYS.5.2	System **COULD** randomly generate events
Func.SYS.6.1	System **MUST** allow trains to move from city-to-city.
Func.SYS.6.2	System **MUST** monitor the positions of all trains.
Func.SYS.6.3	System **MUST** be able to cancel the movement of a train if prompted by the player and will negatively influence a player
Func.SYS.6.4	System **MUST** be able to delay a train at user request.
Func.SYS.6.5	System **MUST** be able to resume a journey at a player's request
Func.SYS.7.1	System **MUST** generate random events
Func.SYS.7.2	System **MUST** apply generated events to map
Func.SYS.8.1	System **MUST** be able to add points to a players score .
Func.SYS.8.2	System **MUST** be able to assign points to a randomly generated goal.
Func.SYS.9	System **COULD** support multiple game modes. i.e. different win-conditions
Func.SYS.10	System **MUST** be able to support exactly two players on one computer.
Func.SYS.11.1	Stations **SHOULD** be purchasable
Func.SYS.11.2	Stations **COULD** provide benefits to the players that owned them.
Func.SYS.11.3	Stations **COULD** have a negative impact to a player that doesn't own it, such as station usage penalty
Func.SYS.11.4	Stations **MUST** be connected such that trains can move between them.
Func.SYS.11.5	Stations **COULD** be part of a coloured ‘Line’. Owning more than one station on a line could provide advantages to the player
Func.SYS.11.6	Stations **COULD** provide goals to players when trains arrive at them.
Func.SYS.11.7	Stations **SHOULD** provide resources for Players that own them.
Func.SYS.12.1	Trains **COULD** be upgradable in terms of speed, fuel usage and carriage limit.
Func.SYS.12.2	Trains **SHOULD** come in multiple varieties with individual benefits.
Func.SYS.13	System **MUST** to terminate itself safely.
Func.SYS.14	System **MUST** be able to save/load games
Func.SYS.15	Player **MUST** be able to quit current game and exit to main menu
Func.SYS.16	System **MUST** have the ability to change settings, such as sound.

A.1.1.6 Non-Functional system requirements

NFunc.SYS.1	The User Interface **MUST** be simple, intuitive and easy to understand.
NFunc.SYS.2	The system **SHOULD** be modular, extendable and updatable.
NFunc.SYS.3	The system **SHOULD** be stable.
NFunc.SYS.4	The system **SHOULD** be safe to use.
NFunc.SYS.5	The system **SHOULD** be secure and **SHOULD** not harbour any malware, spyware or other malicious programs.

A.1.2 Use Cases

A.1.2.1 Generate Goals

Use Case ID:	UC.2
Use Case Name:	Generate Goals
Actors:	Player 1, Player 2, system
Description:	This use case generates goals for a players.
Trigger:	Step 5 of the basic flow. The player tries to access the Goal Menu.
Preconditions:	System must be running a game and the player must be in phase 5 of the basic flow diagram. Player has a train in a Station.
Postconditions:	Goal Menu has 12 goals.
Normal Flow:	1. Player has just picked a goal from the goal menu. 2. System generates a new goal to replace the selected goal. 3. The System's goal menu now has 12 goals.
Frequency of Use:	Very common, whenever a player selects a new goal.
Assumptions:	Player does not have three goals already. Player has just picked a new goal.
Notes and Issues:	