Table of Contents

1. Introduction

1.1. Document

This document covers the requirements specification, architecture and design for a turn based train game named TaxE. It is split into four sections, Requirements and Specification, Abstract Proposal for the Software Architecture, Project Planning and Risk Assessment.

1.2. Single Statement of Need

TaxE is a two-player turn-based train routing simulation game. The challenge for players is to make shrewd choices on goal selections and train upgrades in order to acquire high scores and win. Individual game difficulties, choices of goals and flexibility of game time are aimed to provide a diverse game experience for both beginners and experienced players.

1.3. Brief Overview of Stakeholders

There are a number of stakeholders in this software project. The first of these is the team that is planning and executing the project, including those managing, developing and testing. Initially, the stakeholder with the most input is the customer, as they deliver the brief from which all user requirements are taken from. These requirements have been discussed in further meetings with the customer, who is working on behalf of the final stakeholder, the user. For this game the intended user is a Computer Science student.

2. Requirements

2.1. Use Cases

Before creating an initial set of requirements, we came up with some example use cases for a potential turn based train games. The main two Use Cases we based our requirements on are listed in Appendix 1.1.

After a meeting with Richard Paige, our acting customer for this project, we further refined the requirements, from which we produced a visual prototype (Appendix 1.2) which was used to finalise the requirements with Richard.

2.2. Resources

During our initial meeting with Richard, we discussed the idea of resources in the game. We agreed that the primary resource is money and the secondary resources are things that users can buy in order to help them win the game. These can be trains, fuel or obstacles.

Obstacles are ways to disrupt an opponent's game. These can be things such as blowing up a section of track to stop an opponent's train, blowing up a carriage of an opponent's train, a points failure at a track junction, speed limits on sections of track or an opponent's train breaking down.

2.3. User Requirements

1. From the Main Menu

1.1 The user should be able to start a new game.
Rationale: The customer has said that the user should be able to create a new game whenever they want, but also be given the choice not to, allowing other actions to take place instead of automatically loading a new game.

1.2 The user should be able to return to a game at a later date.
Rationale: The customer has stated that the user should be able to come back to a game if they do not have time to finish it.

1.3 The user should be able to view other player's scores.
Rationale: The customer has suggested that to make the game more competitive and fun, the user should be able to view other player's scores to compare scores, see the high scores and provide a bit more incentive to get a high score.

1.4 The user should be able to exit the game.
Rationale: The customer has said that the user should be able to exit the game from any screen to allow stoppage of the game so the user can continue with another task on their computer.

1.5 The users should be able to enter their player names when setting up game.
Rationale: To make the game more personal, the customer has said that the users should be able to enter individual player names when starting a new game.

1.6 The users should select individual game difficulties when setting up the game.
Rationale: The customer said this so the game will allow each player to play the game at their own preferred difficulty level, allowing more experienced players to still have a challenge whilst allowing new players to still play at a level they are comfortable with.

2. Gameplay

2.1 The main game screen should be a map of Europe with the different train stations and the routes between them.
Rationale: The customer said that the main game screen should give the user an overview of the game showing them the map of Europe with the different train stations and the routes between them. The users should also be able to see how many resources and points they have, as well as the time remaining in the game.

2.2 Each turn, the user should be able to select 1 new goal and buy 2 resources.
Rationale: The specification stated that users should be able to select goals to achieve and buy resources to use in achieving those goals. The specification of the game said that users should only be able to select 1 new goal and buy 2 resources per turn.

2.3 The user should be able to change the performance of trains.
Rationale: The customer stated that it would be good if the users could change the performance of trains by using “power-ups” or a “pimp my train” idea. This allows the users to improve different aspects of the performance of trains such as speed, fuel efficiency, number of carriages or reliability. This makes for a more varied and exciting game.

2.4 The user should be able to disrupt the opponent’s game
Rationale: The customer likes the idea of destruction and said that there should be ways for the user to disrupt the opponent’s game. These ways could be things such as blowing up an opponent’s train, blowing up a section of track on a route that an opponent is using or it could be a financial/resource penalty. The customer also said that some obstacles could last a certain number of turns and then expire.

2.5 The user should be able to select the routes for their trains.
Rationale: The customer said that users should be able to manually select a route for a train to take in order for them to satisfy goals. They said that the user should be able to select a starting station, a finishing station as well as stations that the train should visit on the journey. This will increase the strategy element of the game.

2.6 The user should be able to save games to return to at a later date
Rationale: Following on from U.R 1.2, the customer stated that there should be a way for the user to save a game and resume it from the same point at a later date. This allows more flexibility in playing the game and a longer game length.

2.7 The user should be able to return to the main menu whilst in-game
Rationale: The customer said that whilst the user is playing a game, they should be able to return to the main menu at any time. This allows the user to leave their game, start a new game or load a different game without having to close the game and reopen it.

2.8 The user should be able to exit the game whilst in-game
Rationale: The customer stated that the user should be able to exit entirely from the game from the main game screen. This allows the user to leave the game entirely directly from the main game screen and continue with another activity on their computer.

2.9 The game should be 2 player
Rationale: The specification stated that the game must be 2 player with each player taking alternate turns to make a move in the game.

2.10 The users should choose their starting positions of trains
Rationale: The customer said that he would prefer the starting positions of trains to be chosen by the users rather than being randomly assigned. This will ensure that no situation occurs where a player can't make any move.

2.11 The users should be able to select the same goals as each other
Rationale: The customer said that the users should be able to select the same goals as each other. This would mean that users would be selecting the same routes to complete goals and would therefore bring a more strategic element to the game.

2.12 The users should be able to select the same routes for trains as each other
Rationale: The customer said that to make the game as competitive as possible, it would be good if a user could select the same route for their trains as their opponent.

2.13 There must be at least 5 cities in the game map
Rationale: The brief states that there must be at least 5 cities on the game map for the users to visit.

2.4. System Requirements

1. Functional Requirements

1.1 From the Main Menu

1.1.1 The system should allow the user to start a new game.
From the main menu, the system should allow the user to press a button "Start New Game" which will allow them to proceed in starting a new game. This will load the next screen where they can enter their player names and individual difficulties for the new game.
Satisfies: Starting a New Game (UR 1.1)

1.1.2 The system should allow the users to enter their player names and individual difficulties when starting a new game.
The users should be able to textually enter their player names and then select the game difficulty for each player from checkboxes. i.e Easy, Medium or Hard. Once they have inputted this data, the system should highlight the "Continue" button to allow the user to continue to the main game screen. There should also be a "Return" button for the user to return to the main menu if they wish.
Satisfies: Entering individual player names and difficulties (UR 1.5, UR 1.6)

1.1.3 The system should have a persistent storage system to allow users to load a previously saved game.

From the main menu, the system should allow the user to press a button “Load Game” which will allow them to load a previously saved game. A persistent storage system is needed to enable saved games to be accessed at a later date from disk. The system will also need to be able to distinguish between different saved games so to be able to load different saved games. Once a game is re-loaded, all parts of the game must be loaded to the same state that they were saved in.
Satisfies: Return to a game at a later date (UR 1.2)

1.1.4 The system should allow the user to exit the game at any stage.

The system should allow the user to do this safely, without any potential loss of data and allow them to leave the game and use the machine they were playing the game on. If the user is in the middle of a game, the system should give them the option to save the game before quitting.
Satisfies: Exit the game (UR 1.4, 2.8)

1.1.5 The system should allow the user to view the game leaderboards.

The system should allow the user to do this from both the main menu and and the main game screen. There should be a button “View Leaderboards” on both screens which when pressed opens the Leaderboards screen.
Satisfies: View other Player’s scores (UR 1.3)

1.2. Gameplay

1.2.1 The system should utilise a persistent storage system for saving games

A persistent storage system is needed to enable saves to be saved to disk and accessed at a later date, possibly after the game has been shut down and reopened multiple times. The system will also need to be able to distinguish between different saves.
Satisfies: Saving of games (UR 2.6)

1.2.2 The system should save user progress

A method of saving current resource and obstacle values, map status, player status and overall game state is required. This system must also be readable at a later date.
Satisfies: Saving of games (UR 2.6)

1.2.3 The system should utilise a game restoration system

A method of restoring the game state, players state including resource values, difficulties and map status from the saved game system is necessary.
Satisfies: Loading of games (UR 1.2)

1.2.4 The system should utilise a persistent Leaderboard database

A database should be used that stores the previous top 20 high scores, the player names related to those high scores as well as the date and time that the game of played.
Satisfies: Viewing other Player’s scores (UR 1.3)

1.2.5 The system should allow different levels of game difficulty

The game will vary goals, the number of obstacles, financial payouts and the performance of resources to change the difficulty between easy, medium and hard.
Satisfies: User selection of difficulties (UR 1.6)

1.2.6 The system should scale scores based on each player's game difficulty
Before the game, each player will select their level of difficulty to compete at. The scores of each player should be scaled depending on the difficulty they are playing at to ensure a more competitive and closely fought game.
Satisfies: User selection of difficulties (UR 1.6)

1.2.7 The system should allow resources to be purchased by both players
The game will have an in-game 'shop', from which in-game currency is used to purchase more resources. The user will select "Buy Resource" from the game screen, then a pop-up screen shows all available resources with their corresponding costs. User selects "Buy" and amount is removed from Player's current money and is given the resource purchased.
Satisfies: Resource purchase every turn (UR 2.2)

1.2.8 The system should provide trains with different performance
Each train will have different aspects of their performance - speed, fuel efficiency and capacity. A fast, fuel efficient and small train will perform best however it would be expensive to buy. A slow, large and less fuel efficient train will perform worst however it would be cheap to buy.
Satisfies: Upgrading of trains (UR 2.3)

1.2.9 The system should allow trains to be upgradable
The game will allow a train's performance to be increased by buying upgrades using the in-game currency. The user will select the train to upgrade, select "Upgrade" and is then shown upgrade options and current performance of train. The user selects the upgrade to purchase, and selects "Buy". The funds are taken from the Players money, and the performance improvement is applied to the train.
Satisfies: Upgrading of trains (UR 2.3)

1.2.10 The system should allow users to select goals
Users will be given a choice of preset goals that they must complete, and select which to undertake. The user will be shown a screen on the start of a new turn to select new goals from a selection of preset goals. The system should allow both players to select the same goals. Goals will in the the format of:

- Send a <type of train> from <city 1> to <city 2>
- Send a <type of train> from <city 1> to <city 2> in <x> hours.
- Send a <type of train> from <city 1> to <city 2> carrying <x> passengers.
- Send a <type of train> from <city 1> to <city 2> via <list of cities>

Satisfies: User selection of new goal (UR 2.2)

1.2.11 The system should allow users to place obstacles on the map
When a player has an obstacle, they select a location on a track to place the obstacle to use it.
Satisfies: Disrupting Opponents (UR 2.4)

1.2.12 The system should allow obstacles to disrupt opponents

When a user hits an obstacle, a penalty is applied to the user based upon the type of obstacle they have just hit.
Satisfies: Disrupting Opponents (UR 2.4)

1.2.13 The system should allow manual selection of train routes

A user will first select a train, then "Begin Route", then an overlay will highlight the possible waypoints to take. An initial station is selected. The possible connecting stations are highlighted, then the user clicks on the next station. The user continues selecting stations until they have reached their destination. The user will then select "Finish Route" and the train will begin to travel to start destination.
Satisfies: User route selection (UR 2.5)

1.2.14 The system should show the approximate earnings, time taken and fuel required to take a route

When a user is in route selection mode, an estimation of the monetary gain, time taken, fuel required and route summary should be shown to the players.
Satisfies: User route selection (UR 2.5)

1.2.15 The system should ensure the game is 2 player

This is imperative so that the game operates correctly. When users are setting up a new game, the system must ensure both player's details are entered to allow them to continue to the main game screen. If only one player's details are entered, the system should not allow the user to proceed.
Satisfies: 2 Player Game (UR 2.9)

1.2.16 The system should allow the user to select the starting position of their trains

Once the user has selected their goal and resources for their turn, the system should allow them to select the starting station of the route they want their train to take. The system should allow them to select from any station on the map.
Satisfies: Starting Position of Trains (UR 2.10)

1.2.17 The system should allow users to select the same train routes as each other

The system should allow the user to select the same route for their trains as their opponent. This would mean that if two trains were travelling in opposite directions on the same track there would be a collision. The tougher train would make it through unscathed however the weaker train will lose a carriage and then continue on its journey. If they were travelling in the same direction, the second train cannot overtake the train in front.
Satisfies: Starting Position of Trains (UR 2.12)

1.2.18 The system should allow users to select the number of carriages on their trains

The system should allow the user to select the number of carriages that are on their trains. The higher the number of carriages, the more it will cost to purchase the train, and the train will be slowed however the rewards for completing goals with that train would be higher. Also, the higher the number of carriages, the stronger the train is and the more likely it is to come through collisions unscathed.
Satisfies: Performance of Trains (UR 2.3)

1.3 Viewing Leaderboards

1.3.1 The system should allow the user to see the top 20 high scores.
The top 20 high scores of the game should be displayed in a table format. The system should allow the user to scroll through the table to view the different scores. The leaderboard data should be loaded from a database and should include leaderboard rank, game score, the player name who played the game as well as the date and time the game was played.
Satisfies: Viewing other Player's scores (UR 1.3)

1.3.2 The system should allow the user to sort the leaderboard table.
The system should allow the user to press the column headings of the leaderboard table. This will sort the table based on which column heading was pressed. The column headings should be score rank, game score, player name and date and time of the score.
Satisfies: Viewing other Player's scores (UR 1.3)

2. Non-Functional Requirements

2.1 The system should be suitable for use by SEPR lecturers and students.
The target audience for the game are the lecturers and students of SEPR. The game should therefore be designed with an easy to use interface and be usable to SEPR lecturers and students.

2.2 The system should be testable, maintainable and reliable.
The system should be designed in a way that makes it easy to test the implementation so that potential bugs can be found quickly. It should also be maintainable so that it is easy to maintain current code and also implement new features in the future. The system should also be as reliable as possible and experience as few failures as possible because this is extremely infuriating for the user.

2.4 The system should be well documented.
The system should be thoroughly documented to ensure that future developers of the system can easily understand the implementation and continue developing the system.

2.5 The system should load quickly.
The system should load within 20 seconds.

2.6 The system should be within the store requirements.
The system should be no larger than 100mb.

3. Constraint Requirements

3.1 Design Constraints

3.1.1 The system must run on both Windows and Mac OS.

3.1.2 The system must compile, execute and run on computers in the Department's software laboratories.

3.1.3 The system must not use any hardware that is not part of the standard issue in the Department's undergraduate software laboratories.

The system is required to run on both Windows and Mac OS to ensure that it is usable by both module leaders and SEPR students. The system must also compile, execute and run on the computers in the Department's software laboratories as well as not use any hardware that is not part of standard issue in these laboratories.

3.2 Game Constraints

3.2.1 The system must ensure the user does not have more than 3 goals or 7 resources at any one time whilst playing that game.

This is defined in the initial specification. It also means that the "Buy Resources" screen is not available if the user has 7 resources, and the Goal selection screen is not shown whilst the player has 3 goals.
Satisfies: Goals and Resources (UR 2.2)

3.2.2 The system must ensure the user does not select more than 1 new goal and buys 2 resources each turn.

The goal selection screen is only available at the start of the turn (subject to above requirements) and when 1 goal is selected, the user can confirm and continue back to the main game screen. However if more than 1 goal is selected, the user cannot continue until they remove the extra goals they have selected. The "Buy Resources" screen is similarly limited with 2 resources.
Satisfies: Goals and Resources (UR 2.2)

3.2.3 The system must ensure there are at least 5 cities in the map

The brief states that the map must have at least 5 cities for the user to visit so the system must ensure this holds.
Satisfies: 5 Cities on Map (UR 2.13)

3.3 Project Constraints

3.3.1 Specification Document must be delivered by 12 November 2014

As defined by Project specification, this deadline is immovable.

3.3.2 Working implementation, website, testing and design documentation must be delivered by 21 January 2015

As defined by Project specification, this deadline is immovable.

A requirements matrices which links the User and System Requirements is available in Appendix 2.

2.5. Environmental Assumptions

The game is designed to be played on a modern computer such as those in the Computer Science department. The client will have a screen size of at least 1024x768, a computer capable of running Java along with a keyboard and mouse for control.

2.6. Feasibility, Risks and Alternatives

Most of the requirements are feasible, we know this due to prototyping and scenario based evidence. Some of the requirements however are less feasible. The requirements that are less feasible are as follows:

1.1.3 The system should have a persistent storage system to allow users to load a previously saved game

Implementing a persistent storage system may be slightly more difficult than many of the other game features. Capturing the total state of the game at a given point in the game may be a risk. As well as the difficulty in implementing the feature, it could have an effect on other requirements such as the non-functional requirement 2.6; The system should be no larger than 100mb. The only realistic alternative to this would be to not have a save system.

1.2.11 The system should allow users to place obstacles on the map

Allowing users to place obstacles anywhere on the train routes may be more problematic than some other features to implement, in particular the free choice aspect of this feature. The alternative to this would be a compromise to only allow the users to place obstacles on certain areas of the track, creating a more predictable but also possibly a less enjoyable experience.

1.2.17 The system should allow the user to select the same train routes as each other

Two trains can be on the same track at the same time meaning that collisions are possible. A collision system in the game would possibly require us to calculate momentum of the colliding trains to accurately represent the collision. The alternative to this would be to allow trains on the same route to avoid each other by being on 'different' tracks and simply passing each other.

2.6 The system should be within the store requirements.

This requirement places restriction on what we are able to achieve with the game. The 100mb restriction means some features we need to implement might be made very difficult due to space available. The code must also be clean and efficient. The alternative is to negotiate with the client to accept a trade off with space for more advanced features.

Further to this, there is a danger when using plan based methodology that more time is spent documenting than is spent in development. To mitigate this, during development we are adopting a *Scrum* methodology (see section 4.2) to avoid being held back by excess documentation.

3. Abstract Architecture Proposal

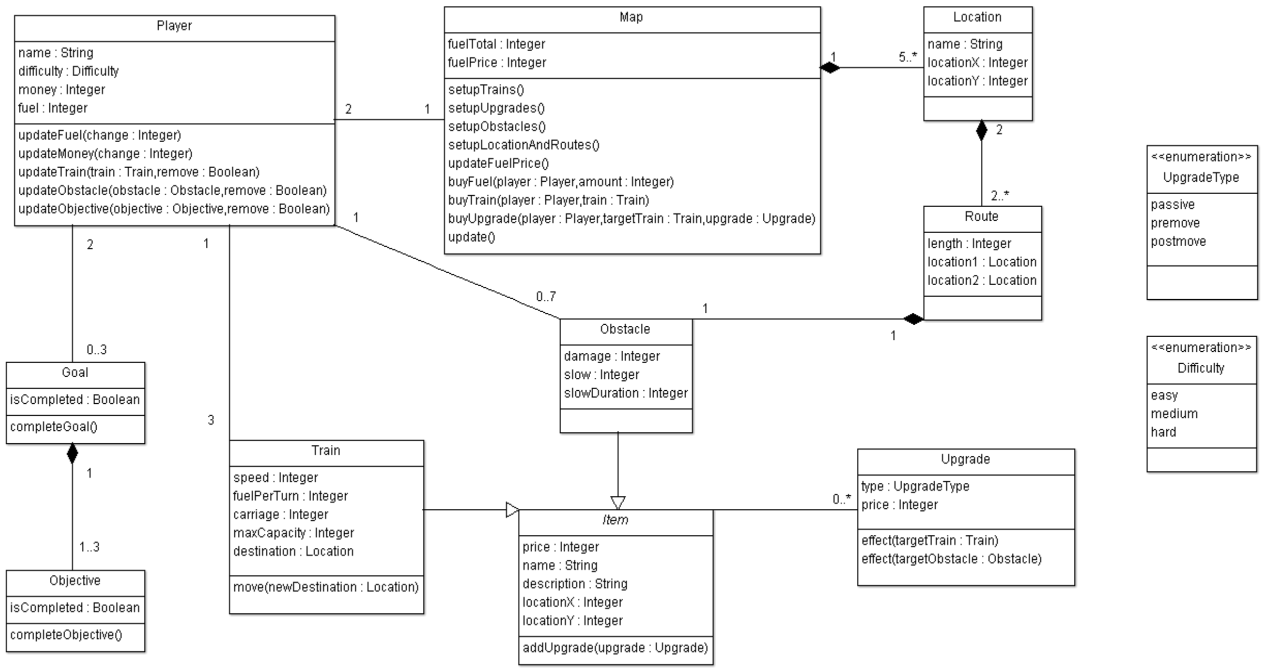
3.1. UML Use Case Diagram

As with the initial requirements, before designing the complete UML that describes the system we created a UML case diagram based on our Use Cases (described in the Requirements section and available in Appendix 1.1). This is available in Appendix 1.3.

3.2. UML Class Diagram

Once the requirements and use cases were produced it was possible to create a possible implementation of the game through a UML Class Diagram, which is available on the next page and also in Appendix 1.4.

A partial description of this structure is below and the full description is available in Appendix 1.5.



Class: Map

All of the information for our game is stored at the top level in the Map class. This contains by extension all the objects in the game world and all the information about the 2 players.

Variables

player1, player2 : Player – These 2 variables hold our 2 instances of the player class that are currently playing on this map. These are passed to the map class when the map class is created.

locations[5…] : List<Location>* – This variable holds a list of the locations that exist in the map. This is created and populated on map creation when the setupLocations method is called. In any map there will be at least 5 locations, with no cap on the maximum number of locations.
trainsInShop[2..] : List<Train>* – This variable holds a list of trains that exist in the shop. This is created and populated in the setupTrains method and is updated when a player purchases a train from the shop using buyTrain. In the map, there must be a minimum of 2 trains in the shop for there to be at least 1 for each player.
upgradesInShop[0..] : List<Upgrade>* – This variable holds a list of upgrades that exist in the shop. This is created and populated in the setupUpgrades method, and is not updated for the rest of the game, since no upgrades will be added or removed from this list (as they must be available to purchase for multiple trains). There can be as little as 0 upgrades available, and no upper cap on the number of upgrades.
obstaclesInShop[0..] : List<Obstacle>* – This variable holds a list of obstacles that exist in the shop. This is created and populated in the setupObstacles method and is updated when a player purchases an obstacle from the shop using buyObstacle. In the map, there is no minimum or maximum number of obstacles available to purchase.
Methods
setupTrains() : Boolean – This method sets up the initial trains available to purchase for both players in the shop. It creates different instances of the train class with different names, values and upgrades, and adds them to the trainsInShop list. It returns true if all of the trains were added successfully, and false otherwise.
setupLocationsAndRoutes() : Boolean – This method populates the locations list by generating different instances of the Route and Location classes and connecting them. It returns true if all routes and locations were added successfully, and false otherwise.
buyTrain(player : Player, train : Train) : Boolean – This method allows a specific player (passed as the player variable) to buy a specific train from the map (passed as the train variable). It returns true if the player's trains list and money are successfully updated (and the map trainsInShop is updated), and false if the purchase was not successful (e.g. if the player did not have enough money).
updateFuelPrice() : Null – This method updates the map fuelPrice on a regular interval. It updates to a new value based off of the current value, combined with how low the fuelTotal integer is (simulating reduced supply increasing price) and how much movement the player objectives require (simulating increased demand increasing price).
update() : Null - This method updates the game at a the end of each turn. Trains locations are updated towards their destinations, players goals are updated and the fuel price is regenerated.

Class: Player

Variables
trains[0..3] : List<Train> - This list holds the trains that the player currently owns. It is initiated as empty and updated when a player buys a train, sells a train, or a train is destroyed; but a player is limited to at most having 3 trains at any one time.

obstacles[0..7] : List<Obstacle> - This list holds the obstacles the player currently owns. It is initiated as empty, and updated when the player buys an obstacles, sells an obstacles, or uses an obstacle. A player is limited to at most 7 obstacles at any one time.
goals[1..3] : List<Goal> - This list holds the goals currently available to the player. It is initiated as empty on the start of the game, and when the game updates, each player's goals are updated.
Methods
updateTrain(train : Train, remove : Boolean): Boolean - This method updates the player's trains array. A train is passed; if the remove variable is false, then the train is added, or if the remove variable is true, then the train is removed. The method returns true if the change was successful, false if failed (e.g. if the train array is full, or the train to remove does not exist in the player's list of trains).
updateObstacle(obstacle : Obstacle, remove : Boolean): Boolean - This method updates the player's obstacles list. An obstacle is passed; if the remove variable is false then the obstacle is added, or if the remove variable is true then the obstacle is removed. The method returns true if the change was successful, false if failed (e.g. if the obstacles list is full, or the obstacle to remove does not exist in the player's list of obstacles).
updateObjective(objective : Objective, remove : Boolean): Boolean - This method updates the player's objectives list. An obstacle is passed; if the remove variable is false, then the obstacle is added, or if the remove variable is true, then the obstacle is removed. The method returns true if the change was successful, false if failed (e.g. if the obstacle list is full, or the obstacle to remove does not exist in the player's list of obstacles).

Class: Goal
Variables
isCompleted : Boolean - This variable tracks whether the goal has been completed. This is set to True when the main objective is completed.
mainObjective : Objective - This variable holds the objective that must be completed for the goal to be completed. It is created when the goal is created.
sideObjectives[1..2] : List<Objective> - This variable holds the side objectives that can be completed for extra rewards. This list is created and populated when the goal is created.
Methods
completeGoal() : Boolean - This variable checks if the main objective of the goal is completed, and then sets isCompleted to true and returns true if the objective is completed, and returns false otherwise.

Class: Objective
Different objectives extend this class. One objective's completeObjective may check if a train has visited each city, while others may check for a certain number of passengers arriving at a location.
Variables
isCompleted : Boolean - This variable tracks whether the goal has been completed. This is set to True when the objective is completed.

Methods
completeObjective() : Boolean - This method checks if the objective has been completed. This is overridden in different subclasses of Objective.

Class: Item
Item is the superclass that defines many of the functions of both trains and obstacles.
Variables
upgrades[0..] : List<Upgrade>* - This list of instances of the upgrade class holds the upgrades applied to the item. It is initiated as empty and then is updated when upgrades are added to the train.
locationX, locationY : Integer - These integers hold the vector coordinates of the item on the map. They are set when the item is added to the map, and updated when the item moves.
Methods
addUpgrade(upgrade : Upgrade) : Boolean - This method adds a subclass of the Upgrade class to the item's upgrades list. This method returns true when the upgrade is successfully added, and false if the adding fails (e.g. the upgrade already exists in the list).

Class: Train
Train is a subclass of Item. Different trains are created with different names, speeds, fuel costs and upgrades that change the trains effect both before and after moving.
Methods
move(newDestination : Location) : Boolean - This method sets the destination of the train to a new instance of the location class. This method returns true if the change was successful and false if the move failed (e.g. the newDestination is not reachable as a valid route from the current location).

Class: Obstacle
Obstacle is a subclass of Item. Different obstacles are created with different names, damages, slows and upgrades that change the obstacles effect on passing trains.
Variables
damage : Integer - This integer holds how much damage an obstacle will inflict on passing trains. If the damage is negative, the obstacle will heal passing trains. It is set when the obstacle is created.
slow : Integer - This integer holds how much a train passing over the obstacle will be slowed down for the next few turns. If it is negative, it speeds up the train. It is set when the obstacle is created.
slowDuration : Integer - This integer holds how long an obstacle can slow a train for. It is set when the obstacle is created.

Class: Upgrade
Upgrade is a superclass that is used to define the base features of various upgrade subclasses. Different upgrades inherit Upgrade, with a different type to define when the

upgrade's effect should be applied and different effects on targetTrains and targetObstacles. This means that an upgrade can have a use for a train, an obstacle, or both.

Variables

type : UpgradeType - This instance of the UpgradeType enumeration is used to define when the upgrade's effect should be applied. It is set in different subclasses of upgrade.

Methods

effect(targetTrain : Train) : Null - This method applies the upgrade's effect on a train (passed as targetTrain). This method is overridden in different subclasses of upgrade, and returns nothing.

effect(targetObstacle: Obstacle) : Null - This method applies the upgrade's effect on an obstacle (passed as targetObstacle). This method is overridden in different subclasses of upgrade and returns nothing.

Class: Location

Variables

locationX, locationY : Integer - These integers hold the vector coordinates of the station. They are set when the station is created.

routes[2..] : List<Route>* - This list holds instances of the route class that define the connections from this location to other locations. It is populated when the station is created, and has a minimum requirement of 2 routes (e.g. the station must be connected to 2 others).

Class: Route

Variables

location1, location2 : Location - These 2 variables hold the 2 locations the route connects. These are set when the route is created.

length : Integer - This integer holds the length of the route a train must travel to reach either of the locations. It is set when the route is created.

Enumeration: UpgradeType

This enumeration is used to define when an upgrade's effect should be applied. Although more may be added, it has 3 values at the moment: passive (e.g. apply immediately and then never again), premove (apply the effect each time before a train moves) and postmove (apply the effect after a train moves).

Enumeration: Difficulty

This enumeration is used to define an individual player's chosen difficulty. It has 3 values at the moment: easy, medium and hard.

There are also some associated OCL constraints that have not been conveyed in the diagram- these can be found in Appendix 1.6.

4. Project Plan

4.1. Approach to Project Plan

From the beginning of the task the team all wished to have tasks assigned in an organised manner with solid deadlines. We chose to create a Gantt chart by breaking down the tasks for each assessment. This was done in a detailed manner for Assessments 1 and 2, where the tasks to be completed (and their timescales) are well known. Whereas for Assessments 3 and 4 the tasks have been broken down in a little detail and some idea of timescales given - but the ambiguous nature of the Assessment tasks means planning in any more detail would be inaccurate and unrealistic.

The Gantt Chart shows a level of dependency between some tasks that leads to a “Waterfall” approach to the project. This is particularly true of the planning phase of the project, where certain tasks (such as completing requirements) absolutely need to be completed before the Abstract Representation can be started. Tasks that are dependent on each other are next to each other on the Gantt chart and are scheduled to run one after the other, due to the speed at which each part of the project needs to be completed, there is very little slack in the schedule and thus it does not appear on the chart. The critical path through the assessments has been considered when scheduling activities, however as a tradeoff of using an online document (where everyone can track latest progress easily), it cannot be displayed using the traditional arrows.

With regards to Assessment 2, the approach taken with the Gantt chart was to decide what order features should be completed in. Therefore within each ‘feature’ an agile development methodology will be adopted, where the feature is built then iterated quickly in short sprints to achieve the desired functionality. The advantage to using agile development is we can do ‘pair programming’, whereby each feature is developed by at least 2 people. This has two advantages, firstly that differences in language proficiency are covered and secondly, in case of illness there is always one other person who understands the code.

The Gantt Chart is attached in Appendix 3.

4.2. Software Development Approaches and Tools

Approach

We considered different approaches to software development before making a decision as a team as to which approach suited us best. Due to our brief driven project and deadlines, as mentioned above we have opted for a mostly *plan* based development method[1], having spent a significant time at the beginning of the project finalising requirements and creating abstractions.

The *Rational Unified Process (RUP)*[2] has been opted for, as we only envisage one change in requirements (at the beginning of Assessment 3). We have defined a clear set of requirements in Assessment 1 based on Use Cases and will stick to these throughout the development process. The layout of the Assessments lends itself almost perfectly the *RUP*, which requires Requirements and Analysis to be provisionally finalised before Implementation. Whereas Implementation is very much test-driven, an approach useful for a team with varying proficiency levels in Java.

Scrum[3], the most well-known agile development methodology, was considered as another option. However, as our specification has already been decided upon in Assessment 1, constant meetings with the customer to refine this specification are unnecessary[4]. The *Scrum* method does not fit into the assessment structure nearly as well as the *RUP* does.

Language

While we may later decide to implement scripts, it is imperative that we decide on a primary programming language to program our game in. Across our existing team we have significant experience in Java and Python, but little experience in other languages.

Due to the nature of games often being very object based, it is favourable to use an object orientated language. As such we have 3 main options:

Java[5] is a strong contender in that all members of the team already know the language. Furthermore; it is cross-platform meaning we would have very little issue porting the game to Linux or Mac and there are a large range of libraries for Java which simplify game development.

However, Java is not without its issues[6]. It is very easy to reverse-engineer, meaning it is simple for individuals outside of the group to access, use and change our code once the game has been distributed to the public. Java also has a memory and garbage collection overhead, meaning that is uses a large amount of memory.

C#[7] is a language very similar in syntax to Java so would not be difficult for the team to pick up. It has a large range of Game Development Environments (especially XNA). It also does not have the overheads that Java has, saving on memory. On the other hand; it is very limited to Windows operating systems except in special cases. This significantly impacts the cross-platform ability of our game. Similarly to Java, it also has reverse-engineering issues significantly impacting on the security of our code.

C++[8] is the most commonly used language in game development. There are numerous tools available for C++ to aid with game development, as well as large amounts of existing open source code to access given its popularity with developers. C++ is however, harder to port across platforms than Java, though not as limited as C#. However, it would be the most difficult language for our team to pick up and start using, impacting on development time, and also suffers from issues such as long compilation times and boilerplate code[9] (code that has to be included in many places with little change).

Our team's choice of development language is Java. The cross-platform advantages it has, alongside it's familiarity in the team, outweigh its memory issues. Furthermore, the reverse-engineering is only an issue if we are concerned about the security of the code: a lot of developers allow their code to be reversed so users can modify the game.

Language Development Suite

When developing in Java, we can use many different development environments. There are 3 main IDEs available to us that offer different advantages and disadvantages.

Eclipse[10] is the most commonly used Java IDE. The main advantage of Eclipse is that it is very extendable, allowing plugins (such as the Android Development plugin for mobile development) to change the functions of Eclipse to suit certain tasks. This option is the IDE our team currently have the most experience in.
Netbeans[11] is an open source Java IDE. There are no major differences between Eclipse and Netbeans aside from Eclipse's range of plugins, but Netbeans does tend to be updated sooner with Java developments.
JCreator[12] is a simple lightweight IDE. It can only be run on Windows, which is a disadvantage, but is very quick and low-intensity to run.

Given that there is very little difference between Eclipse and Netbeans, and that we have more experience in Eclipse, we will be using Eclipse as our development software.

Game Engine Library

There are multiple engines available for Game Development in Java. However, we intend to use jMonkeyEngine3[13]. This is an open source game engine that is cross-platform, and functions in both 3D and 2D. The primary benefit of using jMonkey is that is significantly simplifies the game graphics, meaning we are able to focus more time on developing game function and mechanics, and less time implementing basic visuals.

Code Maintenance

It is essential that we keep code maintained across our team while developing to prevent accidental forking of code. There are 2 main ways we can do this: Using a Git or a SVN[14]:

SVN, an individual repository of code with multiple users updating it. This favours central development as it is a single build, meaning that there are no inconsistencies in builds that developers are working on. However this is limiting in that it prevents developers from working on individual ideas that aren't ready for reintegration with the central repository.

Git is a repository with lots of client repositories that individual users (or groups of users) update. This means that there are multiple builds of a project at once, causing some inconsistencies, but it does allow multiple paths of development at the same time, helping to pipeline the development process.

For our project we will be using Gits, as they allow us to split our development team into individual groups with their own build forks in order to achieve multiple development goals at the same time.

(References for this section can be found in Appendix 1.7)

5. Software Engineering Risks

The following table details many of the different risks of carrying out a software engineering project. We have chosen the 10 risks that are most likely to affect our project, detailing how we're looking to mitigate them and how serious they may be. Further information on the severity and likelihood can be found in the Risk Matrices in Appendix 1.8.

5.1. Risk Register

ID	Risk	Description	Severity	Likelihood	Mitigation
1	Schedule Flaws	Inevitably not every part of the project will be completed exactly on time or in the correct order. Some tasks may take longer than predicted. A team member's other commitments may also play a role in this as the gantt chart we are using to schedule each part of the project does not take into account an individual's other commitments at a certain point in time.	Low	High	The team is working to a schedule on a Gantt chart. This helps mitigate any schedule flaws by clearly setting out task dependencies and deadlines.
2	Evolution of Requirements	As the project advances the customer may decide to make changes to the requirements. This will create problems as the client may request new features or changes that we have not previously scheduled time to implement. The time constraints imposed on the project have to be adhered to so it is possible we may lack the time to implement any new changes to the product. An almost unpreventable risk.	Medium	High	This is very hard to fully mitigate, however constant communication with the client means any requirements changes are known as soon as possible.
3	Misunderstanding of Requirements	A risk linked with the risk of ineffective communications. The failure to communicate with the client properly and understand the client's ideals will lead to the misunderstanding of requirements and subsequently a complete failure of the project.	High	Low	Make sure all team members have regular contact with the client and that requirements are discussed on a regular basis both internally and externally.
4	Lack of Client involvement	If the project team is left to develop the product for long periods of time without input or approval from the	High	Low	Contact the client in any cases where decisions are being made based on any ambiguous

		client, the project may develop in a way that the team members feel is suited to the requirements but the client does not.			information.
5	Inexperience of Personnel	Most of our team have not been involved in a large scale project of this nature before, this could lead to problems that could be easily resolved or avoided with more experienced team members.	Medium	High	Regular meetings to check progress and support if needed.
6	Ineffective communications	Failure to communicate between team members and the customer along with failure to communicate internally between team members will lead to huge problems with the project.	Medium	Medium	Structure to team and external meetings, making sure everyone attends and is kept up to date.
7	First Time Use of Technology	Inexperience with certain technologies and languages such as UML could lead to underestimating the time needed to complete certain tasks using a language that none of the team members have previously used.	Low	High	We have chosen technologies early on to give team members time to familiarise themselves with the technologies, all of which have excellent documentation available.
8	Changes or Disruptions to the project team	Unforeseen illness or absence of team members would increase the pressure on the remaining team members. As we are not able to hire extra members the severity of this risk is increased.	Medium	Medium	It is important to make sure everyone is up to date if one or more person has an issue. Work in pairs so no part of the project is left without an 'expert'.
9	Productivity Issues	A project of this nature with a fairly distant deadline has the risk of team members doing an unbalanced amount of work; little at the start and a lot at the end. This creates situations that might have to be resolved with 'heroics'.	High	Low	A plan with solid deadlines relating to task dependencies. Contact between team members to make sure that scheduled tasks are being completed on time.
10	Adding Unnecessary Features	Team members may become overzealous, adding unnecessary features, this is especially relevant to our project due to the nature of an assessment.	Low	Low	Detailed plan outlining all features to be implemented. If new features are considered, team discussions to keep everyone focused.