Table of Contents

1. Introduction

This document covers the game architecture and interactions of the TaxE game being produced by *Turkish Delight*. It first considers the overall scene structure before going into detail about the game architecture and the features yet to be implemented.

2. Game Scene Structure

A diagram is presented in Appendix 1 that illustrates the structure of the scenes in our game. It represents how the user can navigate through the different scenes in our game.

3. Architecture

3.1. Classes

We haven't included explanations and justifications for certain scene classes as we felt these classes were self-explanatory.

3.1.1. Main Classes

Class: Game

Design Explanation

The class extends the *ApplicationAdapter* class and is used to load the initial scene (*MainMenuScene*) when the game is opened. It contains a *render()* method to override LibGDXs [1] default update method to render a scene. It also contains methods to pop and push scenes onto the stack of scenes within the game. In addition, it contains the different values of the *zOrder* so that *SpriteComponents* can be displayed at the correct depth within the game.

Design Justification

The *Game* class is fundamental in the implementation of the stack that we use to layer scenes (explained below).

Class: Player

Design Explanation

The class stores attributes for each player in the game. Attributes includes player name and start location, as well as the amount of money and fuel each player currently has. It also contains an ArrayList of the trains each player currently owns. The class contains methods to buy and sell the trains of each player.

Design Justification

The most effective method to store player attributes was the creation of a *Player* class.

Class: Scene

Design Explanation

This class deals with the drawing and updating of components on the screen at any one time. It also deals with any input passed to the program by implementing the *InputProcessor* class. All scenes that are created within the game are instances of the *Scene* class. The use of the scene model ensures that individual scenes can be layered on top of each other, with scenes stored in a stack.

Design Justification

The use of a stack means that presenting and dismissing scenes is massively simplified. It ensures that any updates made inside the current scene are applied to the entire game. In terms of future developments, it means that new scenes can be added very easily.

Class: SpriteComponent

Design Explanation

The class extends the Sprite class. When creating an instance of a *SpriteComponent*, the zOrder is assigned to the sprite along with whether the sprite is clickable. A *SpriteComponent* can also be assigned to update when pause is clicked.

Design Justification

The use of sprites in our game is a fundamental part of the LibGDX graphical system. All GUI elements within the game are create as sprites so therefore this class is essential.

Class: ComponentBatch

Design Explanation

The class is used to manage all active *SpriteComponents* within the game. There are several methods within the class to ensure this. The *update()* method is called at the start of a Game's *render()* method to ensure sprites are correctly ordered before they are displayed. There are methods to add and remove active *SpriteComponents* from the game. The *Draw(SpriteBatch)* method is used to draw all active *SpriteComponent*s and the *sort()* method is used to reorder *SpriteComponent*s by their zOrder.

Design Justification

When drawing sprites in batches, LibGDX relies on the order in which the sprites are drawn to decide what depth each object will have. We therefore required a class to manage active *SpriteComponent*s within the game and ensure that they are drawn at the correct depth. The *ComponentBatch* class does this by re-ordering the draw sequence for the sprites according to the zOrder supplied when the sprite is declared.

Class: Clickable

Design Explanation

The class extends the *SpriteComponent* class and is a generic class for all clickable objects such as *Buttons*, *LabelButtons* or *Scrollers*. It has two main methods to check whether a mouse click is within the bounds of the object.

Design Justification

Our game requires multiple clickable objects, it therefore made sense that we should create a generic *Clickable* class that could be used with all of those objects.

3.1.2. GUI Objects

Class: Button

Design Explanation

This class extends the *Clickable* class and allows the creation of objects as sprites that can be added to the GUI of the game and clicked.

Design Justification

Given that our game requires the use of buttons in many different instances, we decided it was necessary to create a *Button* class. We did consider using the LibGDX Button class however given we wanted to assign textures and zOrders to buttons, we felt it was more appropriate to create our own *Button* class.

Class: EditText

Design Explanation

This class extends the *LabelButton* class and allows the creation of objects as sprites that can be clicked and have their text changed.

Class: Label

Design Explanation

This class extends the *SpriteComponent* class and allows the creation of sprites with text above them. The class contains a method *genericFont(fontColor, fontSize)* which generates the standard font used in our game (GOST type A standard) in the colour and size given as parameters to the method.

Design Justification

Given that our game requires the use of labels in many different instances, we decided it was necessary to create a *Label* class. We did consider using the LibGDX *Label* class however given we wanted to assign Bitmap Fonts and zOrders to labels, we felt it was more appropriate to create our own *Label* class.

Class: LabelButton

Design Explanation

This class extends the *Button* class and allows the creation of buttons with text above the texture. The class uses the *Label* class' *genericFont(fontColor, fontSize)* method to generate the standard font used in our game.

Design Justification

We did consider using the LibGDX *Button* class however given we wanted to assign Bitmap Fonts and zOrders to labels, we felt it was more appropriate to create our own *LabelButton* class. In addition to this, we had already created the *Button* and *Label* classes so the implementation of *LabelButton* was very similar to these two classes.

Class: Pane

Design Explanation

This class extends the *Clickable* class and allows the creation of an object that behaves like a scrollpane. A *Pane* can be assigned a zOrder and can also have *SpriteComponent*s added to it. Once a *Pane* has been created, it can be moved in either the x or y direction depending on the orientation of the *Scroller* object.

We are implementing scrollable windows in a way such that the background image *SpriteComponent* of a *Scene* must have a transparent window where the *Pane* should appear. The *Pane* will then be laid underneath the background image and hence will only be visible to the user within the transparent window. When the *Scroller* object is moved up and down, the *Pane* will also move up and own in an amount determined by the percentage value returned by the *onMove(percentage)* method in the *Scroller* class.

Design Justification

We did consider using the LibGDX scrollpane class [2], however after a large amount of research, the implementation of this class appeared complex. Our lead developer has past experience of implementing scrollable windows using separate custom *Pane* and *Scroller* classes. The combination of these factors lead us to decide it would be best to implement the scrollable window using this method.

Class: Scroller

Design Explanation

This class extends the *Clickable* class and allows the creation of an object that behaves like a scrollbar. A *Scroller* can be assigned a range; this defines the minimum and maximum positions to scroll between. The orientation can also be set depending on whether the scrolling is along the x or y axis. When the *Scroller* object is pressed down, the displacement is calculated from original position to the new position of the mouse and then the object's position is updated. The class also contains a method *onMove(percentage)* which gets the percentage that the *Scroller* object is between it's maximum and minimum positions.

Design Justification

The justification is the same as the *Pane* class (see above).

3.1.3. Routing

Class: AiSprite

Design Explanation

This class extends the *Clickable* class and is used to allow further flexibility in showing elements on a *CurvedPath*. It also stores the polygons for collisions.The class is extended by both *Carriage* and *Train*.

Design Justification

Increased flexibility allows for an easier programming experience and implementation when adding new features. Storing the polygons within the abstract class makes calculating collisions simpler.

Class: Carriage

Design Explanation

This class extends the *AiSprite* class and is used to create carriages that are connected to a train within the game. One of its main functions is to calculate the position of carriages behind a train.

Design Justification

The decision was made to use a separate class for *Carriage* due to the high complexity required to calculate the position of a carriage behind a train. It also ensures better modularity within our code. The alternative would have been to include the carriages with the *Train* class however this would have made the class extremely complex.

Class: Connection

Design Explanation

This class is used to store a *CurvedPath* and the *RouteLocation* that the path finishes at. A *Route* is made up of a start location and then a collection of *Connections*.

Design Justification

Necessary to have the *Connection* class so that trains can be given routes and makes it simple to identify whether a given *RouteLocation* is connected to another *RouteLocation* and if so by which *CurvedPath.*

Class: CurvedPath

Design Explanation

Within the game, a path is represented as a *CatmullRomSpline* [3] - this is the standard method of representing a curved path or set of points in LibGDX. *CatmullRomSpline* uses complex maths to calculate the point at which an element is along the curve using 't-values'. Therefore, when implementing carriages which are supposed to be a fixed distance behind the train, the distance between the two would vary. The decision was therefore made to create the *CurvedPath* class to extend *CatmullRomSpline.* The class samples the curve on creation and then stores the pixel distance and vector position that corresponds to the t-value. These distances, positions and t-values are stored in three separate ArrayLists. This allows all three relevant values to be retrieved at index i.

The class allows the trains to move a constant distance every turn with the carriages at a fixed position behind the train. This does mean a more realistic motion was sacrificed. Alternatively, we could have left the carriages to be a variable distance behind the train and allowed the train to move a variable distance every turn. However, this would have added greater unpredictability and difficulty for calculating collisions.

Design Justification

The decision to represent a path as a *CatmullRomSpline* was justified by the fact that there was the most online assistance to implement the function. This consisted of example code, a wiki, online forums offering support and YouTube sources.

The decision was made to store pixel distances, vector positions and t-values in ArrayLists. Although this results in a loss of fidelity of t-values, we have ensured that we sample enough so that the positions are precise to 0.01 of a pixel, this prevents any issues with movement or improper displaying of t-values. ArrayLists were used because they are variable size. Hashmaps were considered, however it would limit the ability to switch freely between distance, position and t-value.

Class: Route

Design Explanation

A route is made up of a start *RouteLocation* and a collection of *Connections*. The start location of the route must take the path of the connection to get to the target location. This target location will then take the path in the next connection to get to that connection's target location, and so on.

Design Justification

The alternative method would have been to store the start location, end location and path as a 3-tuple. This proved prohibitively difficult. The structure used here made it easy to see a route and the *Connection* class can be used by locations to store what they are connected to and via which path.

Class: Train

Design Explanation

This class extends the *AiSprite* class and is used to store all attributes about the different trains in our game. The class contains methods to update a trains position when moving along a route, upgrade a train, assign carriages to a train as well as assign routes to a train.

3.1.4. Scenes

Class: CurrentResourcesScene

Design Explanation

The class extends the *GameWindowedGUIScene* and is used to display the current resources (trains and obstacles) of the player that is currently active. The *Scroller* and *Pane* classes are used to allow players to scroll through their current resources.

Class: DialogueScene

Design Explanation

This class extends the *Scene* class and is used to display a dialog window to make the player aware of a certain event or to confirm a decision that they have made.

Class: GameGUIScene

Design Explanation

This class extends the *Scene* class and is used to display the elements that show both players game information in the toolbar when the *GameScene*, *CurrentResourcesScene*, *GoalsScene* or *ShopScenes* are visible.

Design Justification

Rather than write code to display the information in each class where it should be visible, we decided to create a class that could be re-used. This reduces code duplication and code modularity is ensured.

Class: GameWindowedGUIScene

Design Explanation

This scene extends *GameGUIScene* and contains the methods called by the *Buttons* on the game toolbar.

Class: GameScene

Design Explanation

This scene is the fundamental control of all gameplay within the system. It updates the player information, deals with train collisions, contains the paths between cities as well as leading to many different scenes. (see Appendix 2.3.). It also contains the methods to save and load a game. A game is saved as a text file where each item is separated by a comma. The format of the text file is saved for each player as: Player Name, Player Money, Player Fuel, Player Score, (TrainType#Location) if it's stationary, (TrainType#Location#CurrentProgressionAlongRoute#WaypointReachInRoute#RouteLocationsListenedInString) if it's moving. This is followed by which players go it is and the fuel reserve of the game.

Class: GoalsScene

Design Explanation

This scene extends the *GameWindowedGUIScene* Class. The scene displays a list of current goals in a *Pane* and allows selection of goals.

Class: LeaderboardScene

Design Explanation
The class extends the *Scene* class and is used to display the game leaderboard. The *Scroller* and *Pane* classes are used to allow players to scroll through the leaderboard. At this stage, given scoring has not yet been implemented, the leaderboard is not fully functional.

Class: LoadGameScene
Design Explanation
The *LoadGameScene* extends *Scene* and lets a player load a previously saved game. This Class also makes use of *JFileChooser* [4] so that the user can load games that aren't listed in previously saved games.

Class: MainMenuScene
Design Explanation
This extends the *Scene* class and is the first scene loaded when the game opens. It gives the options to start a new game, load a previously saved game, view leaderboards or exit game.

Class: NewGameScene
Design Explanation
This extends *Scene* and enables the user to select game difficulties, player names and starting cities before starting a new game.
Design Justification
The alternative to this would be to have a settings menu where the user can select difficulty and player names in game. However the starting cities, game difficulty and player names need to be chosen before the game starts for the game to be playable.

Class: PauseMenuScene
Design Explanation
This class extends the *Scene* class and is used to display a menu to the player during a game. The player can then resume, save or exit game.

Class: SelectionScene
This abstract class extends *Scene* and is used to allow the user to select routes of trains when playing the game.

Class: ShopScene
Design Explanation
The class extends the *GameWindowedGUIScene* and is used to display the shop where players can purchase trains, obstacles and resources. The *Scroller* and *Pane* classes are used to allow players to scroll through the items in the shop.

3.1.5. World Objects
Class: Junction
Design Explanation
This class extends the *RouteLocation* class and is used to allow greater flexibility in creating junctions within our game map. A *Train* doesn't stop at a *Junction.*
Design Justification

The main justification is that the class increases flexibility. It also ensures better modularity within our code.

Class: RouteLocation

Design Explanation

This abstract class extends the *Clickable* class and is used to allow greater flexibility in creating elements that have connections. It is implemented by using separate *Junction* and *Station* classes.

Design Justification

The main justification is that the class increases flexibility. Alternatively we could have simplified the structure by extending the *Junction* in *Station* however this would have reduced flexibility.

Class: Station

Design Explanation

This class extends the *RouteLocation* class and is used to allow greater flexibility in creating stations with the map of our game. A *Train* stops at a station, unlike a *Junction.*

Design Justification

The main justification is that the class increases flexibility. It also ensures better modularity within our code.

3.1.6. Goals

Class: Event

Design Explanation

The class is used to define an *Event* using four variables: *EventType*, *TrainType*, *Station* and *TrainCarriages*. The *EventHandler* contains an ArrayList of *Events* (see below).

Design Justification

Looking ahead to when quantifiable goals are implemented in the game, the *Event* and *EventHandler* classes provide a method to track a quantifiable goal. To demonstrate this, we have implemented our absolute goals using arrival events when a specific train arrives at a specific station.

Class: EventHandler

Design Explanation

When a train arrives at a station, it posts an *Event* to the *EventHandler*. The class will be used in the future when the method to track quantifiable goals is implemented. The class contains an ArrayList of *Events* and has methods to push and get the events from the ArrayList.

Design Justification

The justification is the same as the Event class (see above).

Class: Goal

Design Explanation

Goals are the main method of giving absolute and quantifiable objectives to the players. Each goal consists of 3 objectives, the main objective and 2 side objective. Each time it is a player's turn, each goal is notified and it checks to determine if any of it's main or side objectives have been completed. Once the main objective has been completed, the goal is marked as inactive and no longer available to either players.

Design Justification

The Goal class allows us to abstract our goals into main objectives and side objectives, creating scope for more creative linked objectives in the future. It also allows us to easily create as many active goals as we want at any one time. Checking the objective criteria for a player on the player's turn means that the player is notified of their interactions with goals only when they are in control of the game.

Class: Objective

Design Explanation

Objective is the superclass used to create absolute and quantifiable objectives in our game. Currently it has 3 child classes: EmptyObjective, ArrivalObjective and RouteObjective. Each turn, it checks whether the active player has completed the objective and updates itself accordingly.

Design Justification

The use of an Objective superclass means that we can use the Objective class in our Goal class for our 3 objectives, while being able to override and modify the fulfilment criteria to create different dynamic objectives using child classes.

Class: EmptyObjective

Design Explanation

EmptyObjective is a subclass of Objective. It can never be completed, as it's fulfilment criteria is always false.

Design Justification

The EmptyObjective class simply exists so that Goals are not required to implement all 3 objectives, and can leave side objectives empty.

Class: ArrivalObjective

Design Explanation

ArrivalObjective is a subclass of Objective, instantiated with a target destination. It monitors events from the Game's EventHandler, and is completed for a specific player when it detects that a train that that player owns has reached the target destination.

Design Justification

The ArrivalObjective class allows the implementation of the simplest absolute objective; Sending a train to a station. Its implementation using the EventHandler may seem over complicated, but it is built like this to ensure uniformity in design with future Quantifiable objectives that will require the EventHandler.

Class: RouteObjective

Design Explanation

RouteObjective is a subclass of RouteObjective, instantiated with a target destination and start station. It monitors events from the Game's EventHandler, and is completed for a specific player when it detects that a train that that player owns has reached the target destination, after passing through the starting station.

Design Justification

The RouteObjective expands on absolute objectives by allowing the objective of sending a train between points. It's design justification is the same as ArrivalObjective.

3.2. Class Diagrams

To show how the classes in our game are connected, we have created class diagrams for different components of our game. Those components are overall game, routing, scenes and GUI objects. These can all be viewed in the Appendix 2.1 - 2.4.

3.3. Changes to Architecture since Design

After designing our game, we realised that the jMonkeyEngine3 [5] game engine we had chosen was not appropriate. This was because after further research we found that given jMonkeyEngine3 has a native 3D setup, it had very little support for sprites. The is because the depth is described physically within a 3D world and therefore when objects are resized, the layering of those objects is an issue. We looked into the possibility of using a parallel view however jMonkeyEngine3 has not yet fully developed this and therefore is unreliable. Another possibility was to convert between 3D and 2D however this would have been extremely complex and therefore was not an option given our programming experience. We therefore decided to change our game engine and use LibGDX instead.

LibGDX has very good support for sprites and also has good online documentation to help developers. There is also an active community on online forums to help solve programming problems. The game engine must be setup in Eclipse by using Gradle [6]. Gradle is dependency management and build system which ensures an easy method of pulling 3rd party libraries into our game. This means said libraries do not have to be stored in a source tree.

During the implementation of the game, we realised it would be necessary to create more classes than we initially anticipated. This was due to the large underestimation of the game complexity and also down to the fact the team was trying to make the implementation as easy as possible. However, the increased number of classes ensures good modularity within our game and also makes sure we do not have classes that contain a ridiculous amount of code.

During the implementation, we also realised that we required more attributes and methods in the classes we had originally designed. An example of this can be seen in the *Route* class and is illustrated in Appendix 3.1. and 3.2. We initially anticipated that we would only require one attribute *length* of type integer. However, as the implementation developed, the class was required to interact with other classes and became more complex, therefore we required more attributes and methods. The class now has four attributes and eight methods.

Similarly, with the *Train* class, we initially anticipated that four attributes and one method would be required. However, we underestimated the complexity of the class and therefore after implementation, the class now contains eighteen attributes and nineteen methods. This is illustrated in Appendix 3.3. and 3.4.

4. Features yet to be Implemented

In the Assessment 3 brief, it states that obstacles, scoring and quantifiable goals should be implemented. We therefore assume that they do not need to be implemented in Assessment 2. Therefore the *GoalsScene* currently only contains absolute goals. Similarly, scoring and obstacles have not been implemented. This therefore means that the *LeaderboardScene* is not yet fully functional.