Architecture Document

TEAM HEC

1.0.0 Introduction

In this document we explain how every class in each package is implemented and how they are linked together. We will use notation throughout for methods and classes for example: **getStartMapObj()**. Parameters are generally omitted for brevity, but are included when they require explanation. Attributes are generally in italics.

If you haven't set up the project yet and are looking for the setup information please check our GitHub wiki.

It may be useful to access the JavaDocs for our game, for a more detailed description of structure and specific definitions. It can be accessed at the following link : LocommotionComotionJavaDocs

The backend and the UI frontend are connected in a variety of ways ia the use of singleton classes for large screens such as shop or within the backend creation itself, such as Station UI elements that are created within a Station or Train constructor when needed. In this document, we go through each package individually, explaining the key concepts of their implementation and how they link to each other the UI.

1.1.0 The libGDX Library

Our game uses the libGDX library which provides us a framework that allows for easy creation and organisation of asset. We use the Actor class extensively throughout our implementation (Sprite extends Actor, see Sprite documentation within this document). Additionally to giving us a way of representing assets it allows us to group them on Stages and Screens. libGdx can also allow us to deploy our game to Android, iOS and web platforms, but we did not implement this as it was not required by the brief.

1.2.0 In General…

The project in general has two "ends". The front, which deals with UI, and the back, which handles the game object and player data.

CoreGame is the main class in the back end. It keeps track of the **Player** objects and monitors the current state of the trains and stations in the **WorldMap**.

Player has resources including **Gold**, **Fuel** and **Card**s. **Fuel** is used by the **Player**'s **Train**s to keep them moving, **Gold** is spent in the **Player**'s **Shop** and on **Station**'s. **Player** also has **Goal**s generated from the **GoalFactory**.

Goals define objectives for the **Player**'s **Train**s to achieve. They have set start and end **Station**s.

Cards are generated in the **CardFactory**. They create random effects to benefit the player who owns them.

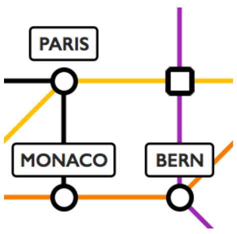
WorldMap is a singleton defining the stations and junctions. All of which are **MapObj**s, characterised by their UI **Actor**s. **MapObj**s can be connected as defined in a **Connection** instance.

To read in more detail about the front end see the Graphical User Interface document.

2.0.0 Packages

2.1.0 Map - com.TeamHEC.LocomotionCommotion.Map

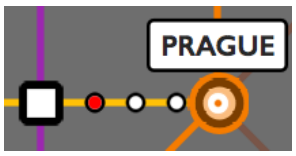
The backend of the map implementation defines Stations and Junctions, their adjacent connections and the lines they belong to. These are all initialised within the WorldMap singleton.



2.1.1 MapObj

The superclass MapObj is extended in both **Station** and **Junction**. The objects contain key information used to calculate the relative position of the station/junction in the map UI and are used to define connections between each other. MapObj contain x and y coordinates to indicate their position, a **Game_Map_MapObj** UI element for drawing the MapObj to the screen and an **ArrayList**<**Connection**> of adjacent connections to that MapObj.

2.1.2 Connection
A connection contains information about two adjacent **MapObj**. They define the route a train can take from one **MapObj** to an adjacent one. The constructor contains two **MapObj** parameters, **startMapObj** and **endMapObj**. Using their coordinates, the difference between them is calculated and a vector is created to indicate the direction and distance (length). The vector is then normalised to a unit length 1. These can all be accessed using getters such as **getStartMapObj()** and **getDestination()** for **endMapObj**.



This allows a **Train** to calculate its position by using the coordinates of the **MapObj** it is leaving and the vector for the connection it is travelling down. A train can scale the vector by the length it needs to travel. We currently use the **Train** speed for this. Connections are also used to define a train's **Route**, which is an **ArrayList<Connection>** for the train to follow. The vector is also used to create a series of white blips to indicate a connection on the UI. The scaling of this vector allows us to place them a set distance apart. A red blip can then iterate through the white blips, indicating direction.

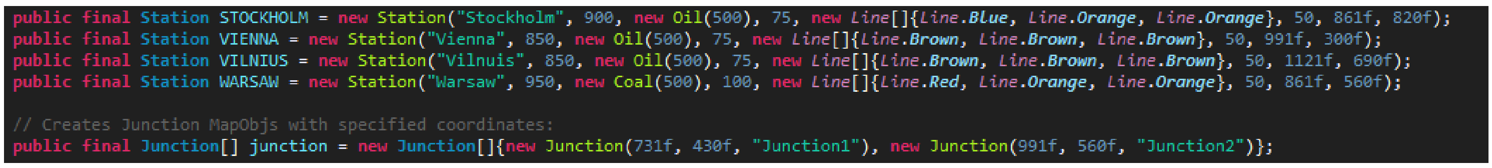
2.1.3 Line
An enum of possible Line colours (Black, Blue, Brown, Green, Orange, Purple, Red, Yellow).

2.1.4 Junction
Extends MapObj and is initialised in WorldMap. It creates a specific **Game_Map_Junction** UI element with its own coordinates.

2.1.5 Station
Extends MapObj, creating a Station with a name, **Resource** type, resource output, cost and coordinates. A station can be purchased within the **Player** class and its owner changed using **setOwner(Player)**. Every station sets up a listener notification detected by **StationListener**. You can add a listener using **register(StationListener)**. Upon a station being purchased all listeners are notified, allowing appropriate changes to be made such as updating UI elements. In **WorldMap**, all of the stations and junctions are created and their connections are initialised within the constructor.
Station uses a **Line** array of size 3 to store which lines it is on. This allows stations to be on 1 to 3 lines simultaneously (this can easily be changed by editing the size of the line array). When creating a station if a station is on less than 3 lines the remaining spaces in the array must be assigned the last unique colour in the array. For example a station on the blue and red line could have array [Red, Blue, Blue] or [Blue, Red, Red] the order is irrelevant so long as the second space is repeated. This is to simplify how **Player** checks a stations lines, by allowing it to loop through the line array if it is not the first element it compares the current line colour to the previous line colour only using the line colour if is different to the previous colour.

2.1.6 StationListener
Contains an **ownerChanged(Station, Player)** method which, if implemented by a class and registered appropriately within **Station**, can be used to make appropriate changes to the UI and other elements.

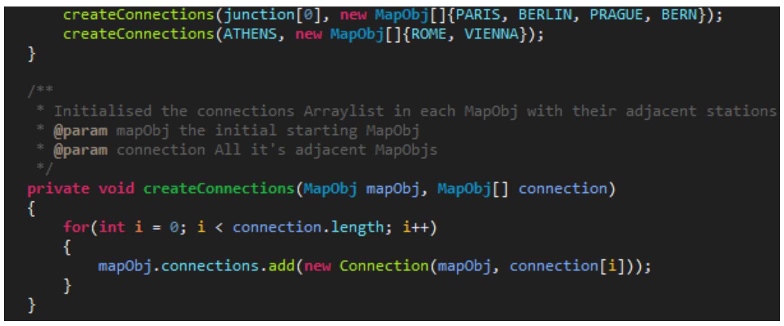
2.1.7 WorldMap
A singleton class which initialises all the **Station** and **Junction** objects, with hard coded coordinates and parameters.



```java
public final Station STOCKHOLM = new Station("Stockholm", 900, new Oil(500), 75, new Line[]{Line.Blue, Line.Orange, Line.Orange}, 50, 861f, 820f);
public final Station VIENNA = new Station("Vienna", 850, new Oil(500), 75, new Line[]{Line.Brown, Line.Brown, Line.Brown}, 50, 991f, 300f);
public final Station VILNIUS = new Station("Vilnuis", 850, new Oil(500), 75, new Line[]{Line.Brown, Line.Brown, Line.Brown}, 50, 1121f, 690f);
public final Station WARSAW = new Station("Warsaw", 950, new Coal(500), 100, new Line[]{Line.Red, Line.Orange, Line.Orange}, 50, 861f, 560f);

// Creates Junction MapObjs with specified coordinates:
public final Junction[] junction = new Junction[]{new Junction(731f, 430f, "Junction1"), new Junction(991f, 560f, "Junction2")};
```

These are added to an ArrayList<**Station**> so they can easily be accessed.

The **ArrayList<Connection>** for each MapObj is then initialised using **createConnections(MapObj, MapObj[])**.



```java
        createConnections(junction[0], new MapObj[]{PARIS, BERLIN, PRAGUE, BERN});
        createConnections(ATHENS, new MapObj[]{ROME, VIENNA});
    }

    /**
     * Initialised the connections Arraylist in each MapObj with their adjacent stations
     * @param mapObj the initial starting MapObj
     * @param connection All it's adjacent MapObjs
     */
    private void createConnections(MapObj mapObj, MapObj[] connection)
    {
        for(int i = 0; i < connection.length; i++)
        {
            mapObj.connections.add(new Connection(mapObj, connection[i]));
        }
    }
```

2.2.0 Train - com.TeamHEC.LocomotionCommotion.Train

2.2.1 Train

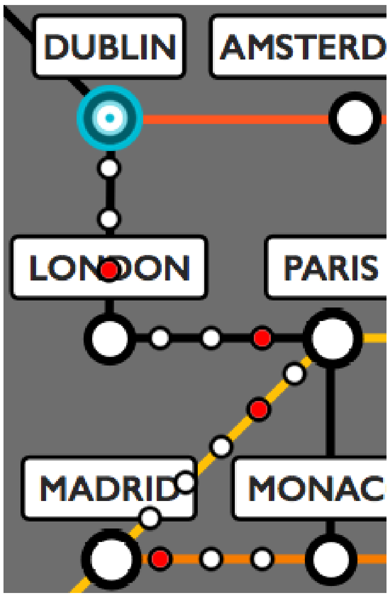
An abstract superclass for all **Train** types. The **Train** class creates a specific train with attributes defined in its constructor. These include the type of fuel the train uses, its base speed, speed modifier and its value.
The train is also assigned an owner **Player**, a **Route** and an associated UI blip, **Game_Map_Train**. The **Route** constructor is passed the starting **MapObj** of the train.
The train can be upgraded using the **TrainUpgrade** class (only partially implemented) which can alter the trains speedMod and fuelPerTurn to improve speed and fuel efficiency, in which a lower fuelPerTurn attribute can increase the distance a train travels using the same amount of **Fuel.** Train upgrades can be added using the ***addUpgrade()***, and **removeUpgrade()** methods in **TrainUpgrade** which can be overwritten by any subclass to perform unique upgrades. If you want to alter the route of the train, access the train's *path* attribute. A train progresses along its assigned path by as much as its total speed using the **moveTrain** method, which allows the **Game_Map_Train** sprite to update its position for animation.

2.2.2 CoalTrain, OilTrain, ElectricTrain, NuclearTrain

Here, **Train** attributes such as *speed* and *fuelType* are hardcoded into the super constructor call of each subclass, with **CoalTrain** being the slowest but the cheapest to run and **NuclearTrain** being the fastest and most expensive.

2.2.3 Route

When this **Route** object is instantiated, a MapObj, representing the starting position of the train upon creation, is passed into the constructor. This is saved in the *currentMapObj* attribute of **Route** to track the **MapObj** the train is at or last passed through. The route the train follows is stored in **ArrayList**<**Connection**> *path*. The train's position within this **ArrayList** and its progress for that **Connection** is tracked using the attributes *routeIndex* and *connectionTravelled*.



2.2.3.1 - Adding and removing a connection

Recall every MapObj has an ArrayList of adjacent connections created in WorldMap. The method **getAdjacentConnections()** looks within the existing **ArrayList**<**Connection**> *path*. If *path* is currently empty, it returns the **ArrayList**<**Connection**> from the *currentMapObj*. If not, it returns the ArrayList of connections from **getDestination(),** the last connection in the route ArrayList.

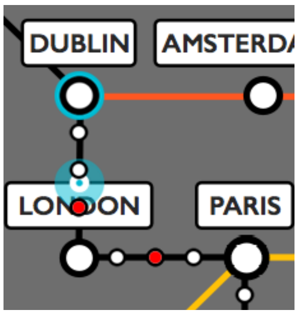
These adjacent connections can then be added to the *path* using **addConnection()** and **removeConnection()**. These methods also contain code to highlight the route on the UI which will be covered in the UI report. Using this method means only adjacent connections to the current route can be added and also allows us to toggle the state of stations if clicked within the UI.

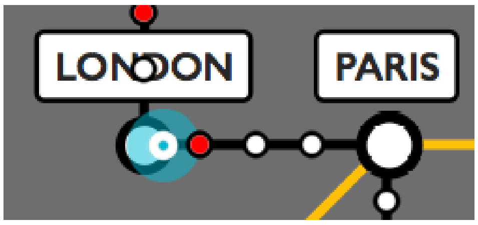
2.2.3.2 - Moving the Train

As stated earlier, we can track the trains position within the ArrayList<Connection> using the *routeIndex* to refer to the index of the connection the train is currently on, and the *connectionTravelled* to refer to how far down the length of that connection the train has progressed.
We can move the train by a specific amount along its route using the **update(moveBy)** method in which we add the moveBy value to the *connectionTravelled* attribute.Now if the new *connectionTravelled* value exceeds the length of that current **Connection**, we can progress onto the next Connection in the Route by increasing the *routeIndex* by 1 and progressing down the new connection by how much it exceeded the previous as follows:

Here, the **ArrayList**<**Connection**> route contains (Dublin>London), (London>Paris) and (Paris>Monaco) connections and the train has moved down the connection by 70.

- *routeIndex* = 0 as it is the first connection in the Route.
- *connectionTravelled* = 70 (which < *connectionLength* 130 so no overflow)





If we take another turn the train moves another 70 and passes london, overflowing by 10 (70 + 70 > 130, therefore 140 - 130 = 10). We therefore move onto the next connection by increasing the route index:

- *routeIndex* = 1
- *connectionTravelled* = 10

If the *routeIndex* exceeds the route.size(), we've reached the end of the route and can clear the route ArrayList, resetting the *routeIndex* and *connectionTravelled* to zero.

2.2.3.3 - Calculating Train position

As connections contain vectors and we know which connection we're currently on using the *routeIndex*. We can calculate the exact coordinates of the train by scaling the direction vector in the connection by the *connectionTravelled* variable and adding it to the vector of the *startMapObj* of that Connection. This is done in the **getTrainPos** method, which returns a Vector2 of the coordinates:

```
Vector2 pos = new Vector2(startMapObj.x, startMapObj.y);
Vector2 vect = route.get(routeIndex).getVector().cpy();
vect.scl(connectionTravelled);
pos.add(vect);
```

2.2.3.3 - How to implement collisions

As the connections have vectors we should be able to detect collisions by adding the vectors that represent the two trains together and checking for 0. This would imply they are travelling in opposite directions. Similarly using the **isReverseOf** method which returns true if a connection is the reverse of another (London>Paris) and (Paris>London).

2.2.3 - Route Listener

An interface for classes to implement, contains a **stationPassed(Station, Train)** method which notifies listeners that a Train has passed a station while on its route. This allows us to check the owner of the station compared to the owner of the train and charge the player a station tax if necessary. It also allows us to complete Goal validation in which a train has to pass through a series of stations to receive an award.

2.2.4 - SpeedUpgrade

Extends **TrainUpgrade**, providing constants for price and overriding the **addUpgrade()** and **undoUpgrade()** methods with their new functionality - in this case increasing the speedMod of a train by 10.

2.3.0 Player - com.TeamHEC.LocomotionCommotion.Player

2.3.1 - Player

The **Player** class contains relevant data unique to each player such as name, points, resources, cards, goals, trains, stations and line data. They are first created in **CoreGame** and are used throughout the duration of the game. Fuel resources are stored within a **HashMap**<**String**, **Fuel**>, this allows a quick and easy access to get relevant information about the player and edit it accordingly using **addFuel(String, int)** and **subFuel(String, int)**. The Strings represent the fuel type.
The **purchaseStation(Station)** method is located in player. The method takes the station to be purchased, validates that the purchase is legal then calls station **setOwner()** and **subGold()** to set the station to be owned by the corresponding player and adjust the players gold total respectively.
It also changes the players **lines**[], which is an array of size 8 each slot corresponding to a different colour. It checks the stations line array and adds one to the player array for each unique colour in the station line array. This array is used in determining line bonuses, as explained below.
The **sellStation** method is also located in player however is currently commented out as it is not implemented on the UI. The method checks that the player owns the station, removes the lines that the station is from the players line array, refunds the player 70% of the base value to the player using **addGold**, sets the station's owner to null to signify it is unowned then removes it from the player's list of stations.

The **lineBonuses** method is used to calculate how much each stations' output should be increased by, based on how much of a line the player owns. For each station the player owns, the method checks each colour of the station to see how many stations on that line the player owns, then gives a 5% increase using **setRentValueMod()**, **setValueMod()** and **setResourceMod()**. There is also a check to see if the entire line is owned, this is simply hard coded and provides an additional increase to the rent, value and resource mods. Rent is not currently charged in this implementation but can be added at a later date.

2.3.2 - Shop

The **Shop** class takes an instance of the **Player** class, as each instance of **Shop** is assigned to one player. Upon creation, it assigns the received player instance to *customer* and creates a new instance of **CardFactory**, passing in its *customer*.
buyFuel()takes the fuel type (*fuelType*) being purchased, the quantity being purchased (*quantity*) and a boolean (*testCase*) to determine if the run is a test case (*testCase* has to be included as the **WarningLabel** class will break a test if it tries to run it, the boolean is used to skip that section). The method then checks if the player has enough gold to ensure they are making a legal action before calling **addFuel(),** passing *fuelType* and *quantity*, then calling **subGold**() passing *quantity* multiplied by a sell price that is declared in **Shop**. The sell price passed to **subGold()** is dependant on the *fuelType* passed to **buyFuel()** . If the player is found to not have enough gold a warning message will be displayed informing the player that they do not have enough gold (assuming *testCase* == false).
sellFuel() works almost identically taking the same parameters (*fuelType, quantity* and *testCase*) however instead of checking if the player has the required gold to buy the resources it instead checks if the player has enough of the resource they are trying to sell to prevent them from performing an illegal action and having negative amounts of that resource. After confirming the player has enough of the chosen *fuelType*, **subFuel()** is called passing *fuelType* and **quantity**. **addGold()** passies the *quantity* multiplied by a sell price. Sell prices are declared in the **Shop** class for each fuel type, they are also only 70% of the prices used when buying and the sell price used is dependant on *fuelType*.
buyCard() takes a boolean (*testCase*). The method checks if *customer* has enough gold and if they have less than 7 cards (max hand size is 7). The method then calls **cardFactory.createAnyCard()** as a parameter for **addCard()** creating a new card and adding it to the customers cards.It also calls **subGold()** passing the price of a card. If they fail gold and hand size validation and *testCase* is false then a warning message will be displayed informing the player they don't have enough gold or room in their hand.

2.4.0 Goal - com.TeamHEC.LocomotionCommotion.Goal

2.4.1 - Goal Factory

GoalFactory generates all the fields necessary for Goal class. When called, **GoalFactory** will be passed an integer value, this integer value represents the turn count of the game. This is so that **Goal** can show which turn it was generated on, however this has not yet been implemented due to time constraints. Secondly, it gets the current instance of the **WorldMap** and from this populates an **ArrayList<Station>** called *stations* with all the stations from *stationList* in **WorldMap.** *stations* is used in the method **newStation()** which simply returns a random station object from *stations*.
To create a new goal the method **CreateRandomGoal()** must be called. This function takes no parameters and returns a Goal object. **CreateRandomGoal()** first generates two random stations and ensures they are not the same, then selects the type of cargo and finally generates the reward. The method **genReward()** takes two parameters, both Station types, creates a new instance of **Dijkstra** and utilises the **computePaths()** method to find the distance between the start station and end station. The value returned will be a double and will be rounded to an integer value.

2.4.2 - Goal

New **Goal** objects are only generated from **GoalFactory** (outside of tests). The variable *special* is set to false inside the initialiser. Three booleans, *startStationPassed*, *stationViaPassed* and *finalStationPassed*, are required to record how far a train is to completing a goal and as such are set to false during initialisation. The accessors for the Stations associated with a goal return the name of the station. That is they return a string from **Station.getName()** rather than the instance of the generated station that is associated with the created **Goal**. The **assignTrain()** method is used to assign a **Goal** to a player's **Train** once they begin route planning. It also checks to see if the players train is at the start station, if it is *startStationPassed* is set to **true**.The method **stationPassed()** is called when a train arrives at any station on its route. It checks if the station the train is currently at is either the start, intermediate or final station then changes the boolean variable associated with start/via/end. Finally it checks to see if all stations are passed and calls the function **goalComplete()** if they have been.

goalComplete() is a simple method that adds the rewards to a players wallet. It then removes itself from a players goal list and sets all **passed** variables to false.

2.4.3 - Special Goal

SpecialGoal is used for more interesting goals. These will be outside the mundane Goals. **SpecialGoal** inherits from **Goal** but modifies its *special* attribute. No special goals are currently implemented but the idea is that they give more interesting rewards and have more depth to them. For example, a **SpecialGoal** might be to take tennis players from three cities to Wimbledon in London. When implemented **SpecialGoal** should give greater rewards, perhaps even cards.

2.4.4 - Dijkstra

2.4.4.5 initialiseGraph()

Initialises a graph of **Node** objects connected by **Edge**s. Each node represents a MapObj and each edge its **Connection**s. It is used to generate a form to which Dijkstra's algorithm can be more easily applied for use when calculating rewards in **Goal**.

2.4.4.5 LookupNode()

To find the length of a path we need to pass a Node to the computePath() function. Passing a MapObj in will crash the program. **LookUpNode()** returns an instance of a node that represents the **MapObj** passed. We can then use this to access the node's *minDistance* attribute.

2.4.4.6 ComputePaths()

This function uses **Node** and **Edge** objects to compute Dijkstra's algorithm which in turn computes the reward for a goal. The reward is stored in the *minDistance* attribute of a destination node. All **Node**s now contain the minimum distance from itself to the start node defined previously.

2.4.5 - Node, Edge

To compute Dijkstra's algorithm an abstract graph that represents the map is created. To help with this we define a **Node** class. A node represents any significant point on a map i.e. stations or junctions. Hence a **MapObj** is passed in the constructor.
Every station has a list of stations it is connected too, hence each Node has an **ArrayList<Edge**> aptly named *edges*. **Node** implements **Comparable** because Dijkstra uses a priority queue which requires a comparator to ensure nodes are ordered in a specific way. This makes it easier to order nodes within the queue and also gives support for the **getShortestPathTo()** function.
An **Edge** itself is a standalone object, with two parameters target and weight. *target* refers to the **Node** that the edge points to and the *source* of **Edge** is the node object that created the **Edge**. The weight refers to the length of the vector between the two **MapObj** (represented as nodes) as computed inside the **Connection** class.
Direction is unimportant, hence a node can be considered to only have a next node and not a previous node. For instance, if it was directed, **Node(a).***next* would be **Node(b)**. and **Node**(b).*previous* would not equal **Node**(a).

2.5.0 UI Elements - com.TeamHEC.LocomotionCommotion.UI_Elements

Used to abstract away from the LibGDX Actors and reduce the lines of code by using anonymous Java classes.

2.5.1 - Sprite

Here, the Sprite class extends the **Actor** class and provides a simple platform to create and display an image on the screen. The constructor, allows us to quickly create an **Actor** with desired coordinates and preloaded **Texture**. If we want to alter the position or texture of the **Sprite**, we can simply use the getter and setters within. Once the **Sprite** instance has been created, it can then be added to the LibGDX **Stage** and displayed

2.5.2 - SpriteButton

Extends the **Sprite** class but adds some functionality including eventlisteners for **onClicked(), mouseOver()** and **mouseExit().** Now if we want to provide functionality for the **onClicked()** event, we can simply override this method using **anonymous Java classes** like such, saving space and coding time.

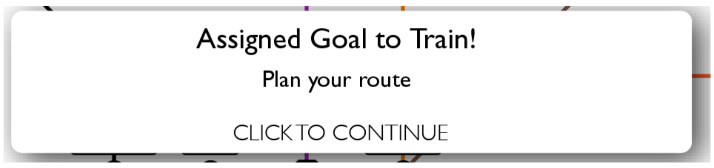
```
SpriteButton buttonExample = new SpriteButton(10, 10,
TextureManager.getInstance().button){

	@Override
	public void onClicked()
	{
		// Anything in here is now executed specifically for this button
	}
```

```
        };
```

2.5.3 - WarningMessage

Used to display a warning message on the screen. The **WarningMessage** can be displayed anywhere and will be layered on top of everything and closed with a click.
Simply call this with the desired message.

```
        WarningMessage.fireWarningWindow("Assigned Goal to Train!", "Plan your route");
```


Assigned Goal to Train!
Plan your route
CLICK TO CONTINUE

2.5.4-7 Asset Managers

The next few UI elements are structures that create and handle multiple **Sprites** and **SpriteButtons**. We have not got into great detail because they mostly consist of series of instantiations of its **Spites** and also there is more documentation in the GUI report document.

2.5.4 - Game_PauseMenu

A interface for the user to save the game, change in-game preferences and exit to the start menu.

2.5.5 - Game_Shop

The in game shop is a window that overlays the game screen allowing the user to select between the buy section or the sell section of the shop. Selecting buy or sell sets all the labels and value to the corresponding modes. The items (fuel or cards) are collected in single classes because they have so many different features (add, minus and buy button) in common we just create them once per item and swap between buy and sell.

2.5.6 - Game_StartingSequence

Simply takes users through selecting stations. Allows us to hide elements of the game screen we don't want the user using before we have set the stations and initialise the **CoreGame**.

2.5.7 -GameScreenUI

The game screen UI manages all the game screen buttons and resource labels.

2.5.8 - Game_TextureManager

A singleton class which is used to store key game image **Texture**s used in the game.

2.6.0 Resource - com.TEAMHEC.LocomotionCommotion.Resource

2.6.1 - Resource

The superclass of resources used by the player such as **Fuel** and **Gold.** Contains getters and setters for altering their value and a string for indicate their type.

2.6.2 - Fuel

The superclass for all fuels, **Oil, Coal, Electric** and **Nuclear** - adds the cost attribute to the extended class **Resource.**

2.6.3 - Coal, Oil, Electric, Nuclear

Extends Fuel, with each class initialising its cost from Coal to Nuclear in terms of expense and a string to indicate its type.

2.7.0 Card - com.TEAMHEC.LocomotionCommotion.Card

2.7.1 - Card

An abstract class inherited by all subclasses of Card. Contains **Player, Texture** and name within the constructor so assign an owner and image to use for each card and an **implementCard()** method which can be overridden by a subclass to give specific functionality. **implementCard()** is used in the **Game_CardHand** to make the card perform its action. Shortly afterwards the UI disposes of the card.

2.7.2 - CardFactory

Used to generate **Card** instances. The card factory initialises all the different types of **Card** and adds them to an ArrayList to group each type together, such as resource cards and magic cards. A **CardFactory** is assigned a **Player** in the constructor, which can be null if used within the **Shop.** The **createAnyCard()** methods groups all the ArrayLists together and returns a random one from within the list, while the other **createResourceCard()** and **createMagicCard()** return Cards from within each individual ArrayList.

2.7.3 - ResourceCard

Extends the **Card** class, overriding its **implementCard()** method so the player is given a random amount of fuel depending on the *fuelType* set. FuelType is passed into it's constructor along with **Player** and **Texture** as before.

2.7.4 - CoalCard, OilCard, ElectricCard, NuclearCard

Extend the **ResourceCard** class, providing the *fuelType* accordingly - the constructor parameters only contain the **Player** to which the **Card** belongs.

2.7.5 - GoFasterStripesCard, TeleportCard

Both extending the **Card** class, these are ideas to implement within the magicCards ArrayList of the **GoalFactory** which provide random effects, such as the **GoFasterStripesCard** which creates a **SpeedUpgrade** instance upon being implemented, or the **TeleportCard** which was designed with the idea in mind of teleporting a selected train to another position on the map. The framework has been provided but the implementation is incomplete.

2.7.7 - Game_CardHand

Used to implement a players **Card** hand in the UI. Gives a platform for the user to view and use their cards - holding up to 7 cards at once which can be raised up when selected and implemented on click.

2.8.0 Scene - com.TEAMHEC.LocomotionCommotion.Scene

This package was designed to tidy up and refactor the UI scene management side of the project by creating a **SceneManager** and a series of **Scene** extended classes. Scenes could then be loaded and unloaded and all their resources and buttons grouped together in a single place, simplifying the process and making it easier to add new scenes to the project, while providing key methods within the superclass to deal with backend management. The **StartMenu** scene was successfully created but the rest of the project proved too time consuming to restructure. We left the implementation included to provide a platform for following teams to utilise if they wish. We would strongly recommend using **Scene** if new screens need to be implemented.

2.8.1 - SceneManager

A singleton class where all the **Scene** instances would be instantiated and stored. The *currentScene* could be stored and new scene loading could be done within this class.

2.8.2 - Scene

A superclass for other scenes to extend. The **Scene** class creates everything needed to display UI Actors on the screen and provides an **Array**<**Actor**> for other scenes to add their **Actor**s too. Methods such as **addToStage()** and **removeFromStage()** could then be used to add **Actor** instances to the stage to be rendered or their properties such as setting them touchable or visible could all be altered at once using **setVisibility()** and **setActorsTouchable().**

2.8.3 - StartMenu

An subclass of **Scene**, the **StartMenu** initialises all the UI actors needed within the **StartMenu** scene, such as all the buttons and menu navigation resources. These are all added to the **Array**<**Actor**> within the **Scene** class, which then can be added to the stage calling **addToStage()** within the **SceneManager.**

2.9.0 Game - com.TEAMHEC.LocomotionCommotion.Game

2.9.1 - CoreGame

CoreGame is used to control the back end in its entirety. It represents the current game in progress and keeps track of **Player** and **Map** objects. The front end of the application accesses most of its properties through an instance of this class.

The constructor contains Strings for each **Player** *name*, their initial starting stations and the *turnLimit* set on the game - these are all created in the **StartMenu** and this is where the **Player** instances are created and assigned their initial resources using **getBaseResources()**.

The class also contains JSON functionality for saving the current game state to later be reloaded if the player quits mid game. This is outputted to a JSON file which can be formatted on https://www.jsoneditoronline.org/ for readability. Loading games has not yet been implemented but the **saveGameJSON()** method currently saves everything that should be needed to recreate a **CoreGame** instance. The main challenge will be recreating the UI.

3.0.0 Improvements

3.1.0 Goal Factory Improvements

There are a flurry of potential improvements for **GoalFactory**. A lot of features were present during initial implementation but were removed due to time constraints. An example is rewards. Alongside giving the player a sum of money they could also receive a **Card** generated by **CardFactory**().

At the moment **GoalFactory** has a hardcoded value for turnCount. We would like to add the turn a goal was created as a parameter for a created **Goal** such that it is displayed as the "start date" for a **Goal**. A feature not used in the implementation of Dijkstra is **getShortestPathTo()**. This returns an **ArrayList**<**Node**> ordered with the shortest path to a given node. This could be implemented to show the player the most optimal route for their task. Currently, **GoalFactory** does not generate a 'via' **Station** and therefore does not take the intermediate stations into consideration when assigning rewards. Finally, and possibly most excitingly, very little **SpecialGoal** code has been done so far. We would really like to see that developed further in the next generation. We'd love to see lots of interesting and funny tasks being set.

3.2.0 Train and Station upgrades

There is currently very little code used to upgrade trains and no code for purchasing more trains which contradicts our original plans, it is a feature that can be easily added. At the moment it is possible to purchase a station and if a player owns all stations on a line then they receive a larger bonus from each station. However this is not fully implemented within the GUI and therefore could be improved. On a similar note, owned Stations were designed to charge rent to another player who lands on that station. Listeners have been set up in the **RouteListener**, but this is a feature yet to be implemented. We would also like to be able to sell stations a player currently owns. Whilst it exists as code and is working it has not yet been implemented in the GUI and is not accessible by the user.

3.3.0 Gameplay

At the present time the implementation only supports the standard game mode "Turn Timeout". This contradicts the design to have two game modes, the second being "Station Domination". The aim of Station Domination is to buy as many stations as possible, completing Goals not for points but cash to buy more stations, so the game is more like monopoly. "Turn Timeout" also needs some work. We currently don't have a points system and the game will not end after the *turnLimit* expires. This will need to be implemented in the next stage.

3.4.0 Saving and Loading

The methods used for saving are fully supported within the game however they are not attached to a button in the GUI. This would be a simple improvement as a save menu exists in the GUI but has not been assigned to the save methods. Conversely loading a game has no methods which will need a full implementation.

3.5.0 Events

Unique events within the game that disrupt the regular flow of gameplay or force the player to change their tactics have yet to been implemented. These could be anything from random monkey attacks, gorillas on the line, collisions or even just the distribution of randomly given cards or prizes. The implementation of random events will keep the game fun and refreshing by keeping the player on their toes. Moreover, magic cards were not implemented. These are cards that can be given to a player that trigger events on the map and can lead to a more tactical gameplay. Adding these features would be straightforward as it would just require creating subclasses of Card that trigger events rather than give a player bonus resources.

Events such as collisions have yet to be implemented but the existing framework could be added by checking if two trains are on the same connection, either using the **isReverseOf(Connection)** method within the **Connection** class, or making use of the vectors available and adding them to see if Trains are travelling towards each other. The **abortRoute()** method with the **Route** class could then be used to send each Train back to their previous stations, with another possible side effect such as Train repairs or loss of Gold.

Other features could be implemented to prevent collisions, such as a special **Card** that lasts a certain number of turns making the selected train immune to collisions or conversely rewarded for colliding with the opposition's train. Our existing framework allows the implementation of any of these ideas without too much work within the backend of the project and provides the perfect platform to expand any of your creative ideas.

4. User Requirements We Have Not Managed To Meet

As in any software development we measure progress against the goals we set out at the planning stage of the development process. Though we have met most of our User Requirements we have not yet met 9 of the 29 requirements. This section will give the reasons why we did not meet them.

Firstly **User.GP.2.4**, **7.1**, **7.2** and **8** were all relating to scoring points. Points are one thing we are not allowed to implement in Assessment 2. So if you take up this game in Assessment 3 you need to make sure goals give points as a reward and that a player can complete goals to get these points and win/lose/draw the game based on the points earned.

Secondly we haven't implemented events yet. This means that **User.GP.4 and 6.3** are not met because **GP.4** is for random events and **GP.6.3** is for 2 obstacles (other than junctions) which need events to affect the train.

Thirdly we did not meet **USER.GP.9** for multiple game modes because it was not a priority and we simply did not have time in assessment 2 however our game modes are an important USP of our game especially the Station Domination mode.

Fourthly we have **USER.GP.11.1** and **11.2** that are partially implemented. These requirements involve station and line bonuses that have not been fully implemented.

Finally **USER.GP.5.4** the user requirement that says that the player should be able to halt and restart trains whilst on a route and **USER.GP.12.1** the user requirement for upgrading trains were not met again because there was not enough time and they were not a priority.