Architecture Report

Introduction

This document details the architecture of Off the Rails- a turn-based strategy game by SEPR team JKG. In this report, all non-test classes are expanded with methods contained (other than accessors and mutators), with justification of architectural implementation and class interactions design choices.

Class List:

Graph design:

MapGraph

- **CreateMapArray()**
- **GetJunctionList(String file)**
- **MoveTrain(int TrainID, int Location, int Destination)**
- **AddTrain(int TrainID, int Location)**
- **RemoveTrain(int TrainID, int Location)**

Station

- **(Inherits Junction)**

Junction

- **GetID()**
- **AddTrain(Integer TrainID)**
- **RemoveTrain(Integer TrainID)**
- **GetConnectedJunctions()**
- **GetTrains()**
- **IsPresent(Integer TrainID)**
- **FindNext(int Destination)**

Checkpoint

- **RepairFault()**
- **CauseFault(String Message)**
- **(Inherits Junction)**

GUI/Game Engine:

GameEngine

- **create ()**
- **render ()**
- **nextPlayer()**
- **incrementTurn()**

PlayerInfo

- create()
- render()

MapGUI

- create()
- updateTrainList(Player player)
- moveTrain()
- render()

Coordinates

PlayerShop

- create()
- render()

Goal Engine:

GoalEngine

- getGoalDescriptors()
- getGoalDescriptor(int i)
- endTurn(ArrayList player1Trains, ArrayList player2Trains, int goalToDestroy)
- newGoal(int goalID)
- getCityName(int junctionID)
- destroyGoal(int goalID)

Goal *(Abstract)*

- getGoalID()
- getNumCarriages()
- getDestLocID()
- getRewardMoney()
- getRewardPoints()

GetToDestinationViaStationGoal

- getStartLocID()
- getTurnLimit()
- reachedDestinationStation(int turnCount, int trainID, int noCarriages)
- goalStartedForTrain(int turnCount, trainID)

GetToDestinationGoal

- reachedDestinationStation(int turnCount, int trainID)

Other:

Train

- getTrainID()
- getEngineType()
- getOwnerID()

- **getCurrentJunction()**
- **getFaultRate()**
- **isFaulty()**
- **getNumCarriages()**
- **getTier()**
- **getSpeed()**
- **moveTrain(int destinationID)**
- **upgradeTrain()**
- **breakTrain()**
- **repairTrain()**

Player

- **getPlayerTrains()**
- **buyNewTrain(int cost, int engineType, int ownerID, int trainID, int faultRate)**
- **getPlayerScore()**
- **getPlayerWealth()**
- **increasePlayerScore(int numPoints)**
- **increasePlayerWealth(int numPoints)**

Class Justification:

GameEngine

The GameEngine class is an extension of the ApplicationAdapter class from LibGDX, which is the default class for applications in LibGDX. This is run on startup, and it creates new instances of the goal engine and GUI classes, creates instances of the Player class to represent each player and assign them some trains, and handles the passing of turns between players.

MapGUI

The core of the architecture is the class MapGUI which is an extension of the game class from LibGDX. This handles both the GUI and the input for the game. The GUI and the input system are bundled together because LibGDX is an event driven system. We decided to use LibGDX because the game ruleset means that it never needs to update unless the player(s) has interacted with it in some way. MapGUI has the function create(), which is used to make a new instance of the game in the initial state, this creates an array called stationCoordinates, this stores the data used to determine the location of the buttons used to interact with train stations. This data is then used by the MapGUI class to place the station and junction buttons into the window, and attach event listeners to them to handle input. The function updateTrainList retrieves the list of trains owned by a player - and their attributes- in order to display them as buttons and be interacted with by each player. All of the player commands are executed through buttons which are operated by the MapGUI class.

Player
The Player class contains all of the information about each player's money, assigned trains, and score. It also contains the methods used to change these values used by the shop and the goal engine, this is useful because it means that all classes which need to alter the player attributes use the same methods, and no class can access the player attributes without going through the player class.

Train
The Train class contains the information about trains in the game, the methods used to retrieve this information, and the methods used to change their attributes. In future versions, when the shop has been implemented and trains can be upgraded, the PlayerShop will call the methods of the Train class to alter the train's values. This is useful because it means that all classes which need to alter the train attributes use the same methods, and no class can access the train attributes without going through the player class.

PlayerInfo
The PlayerInfo class is used to display the current money, turn, and any other relevant information for the player, this information is retrieved from the Player class.

MapGraph
The MapGraph class is used to store the information about the state of the map, including which junctions are connected to one another and where trains are. It also contains the method used to move trains from one junction to another, and methods to add and remove trains to and from junctions. The map works as a standard graph, where each station, junction or checkpoint is a node and any connections between nodes represent the track. Finding the next station or junction in a track is done by following a edge of the graph through checkpoints until the next Junction ID that corresponds to a junction or station.

Junction
The Junction class is used to store the attributes of any point of interest on the map, it is extended to checkpoints and stations. It stores the junction ID, information about trains at the junction and which junctions can be reached from the junction. Having

the information about the reachable junctions immediately available means that it doesn't have to be calculated in real time. Stations are the named junctions which are featured in goals. Checkpoints are junctions which have only two connected junctions. They cannot be selected as the destination for a train, only entered as intermediate points between other junctions, they are also capable of having impassable faults. Checkpoints exist because they can be used to make different weightings on the routes between junctions in a way that allows us to place faults on the route, preventing the player from travelling past the fault. Checkpoints also allow the player to travel partially along a long route each turn.

Goal
The Goal class is the class used to represent the game's objectives. It is extended by GetToDestinationGoal and GetToDestinationViaStationGoal, which are specific goal types with specific purposes. Goal attributes can't be changed once the goal instance has been created, this is reasonable as changing the goal once it has been displayed will likely confuse the players. The Goal class has functionality that will apply to all goals, which can then be extended to the needs of specific goals. This was chosen because it makes it simple to add new types of goal at a later stage. Unfortunately, due to time constraints, in this release of the project, only GetToDestinationGoal was finished by the deadline, this is why we only have one type of goal in this version.

GoalEngine
The GoalEngine class is that which handles the creation, storage, and completion of the game's objectives of 'Goals'. It contains private methods to create and destroy goals of different types and ensures that there are always exactly three goals for the entire runtime of the game. It also checks which goals have been completed at the end of both players turns, allocates the rewards assigned to the completion of each goal, using the methods in the player class, and repopulates goals with new ones at random. As one of the assessment specifications was to give the players a new goal each turn, if no goals have been completed at the end of a given turn, the Goal Engine will delete a random goal, and create a new one in it's place. This is really helpful as it means Goals are all self-contained within this engine and the GameEngine class can deal with goals abstractly, simply by ensuring that the GameEngine class is initialised and called to check on their status at the end of each turn.

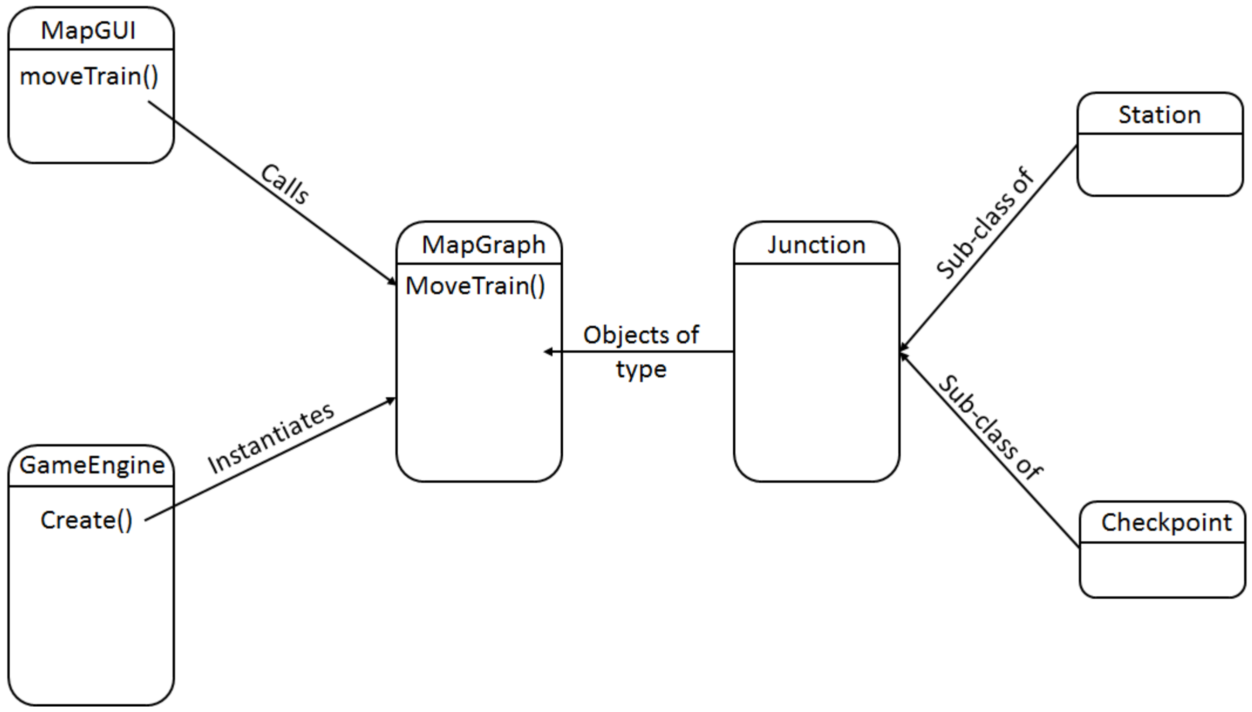
Coordinates

The class Coordinates is a simple class for storing 2D coordinates, which are used frequently throughout the game. It was decided that it was simpler to have a dedicated class, rather than defining a separate variable for each x and y coordinate, every time one was needed.

<u>Class Connection:</u>

Graph design:

MapGraph is the backend representation of the map state, it contains the method used to move trains around the map, which is called by the player input from MapGUI. The MapGUI object for the game session is instantiated by GameEngine on startup.

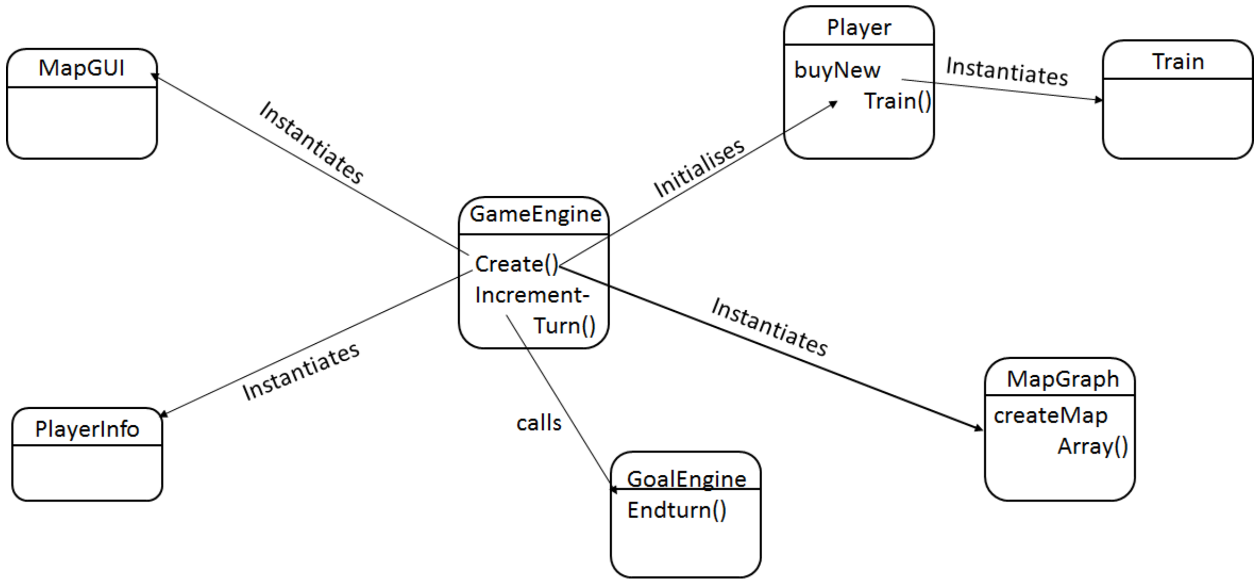


MapGUI:

MapGUI Handles all the input, so it calls methods from MapGraph and GameEngine when buttons are pushed.

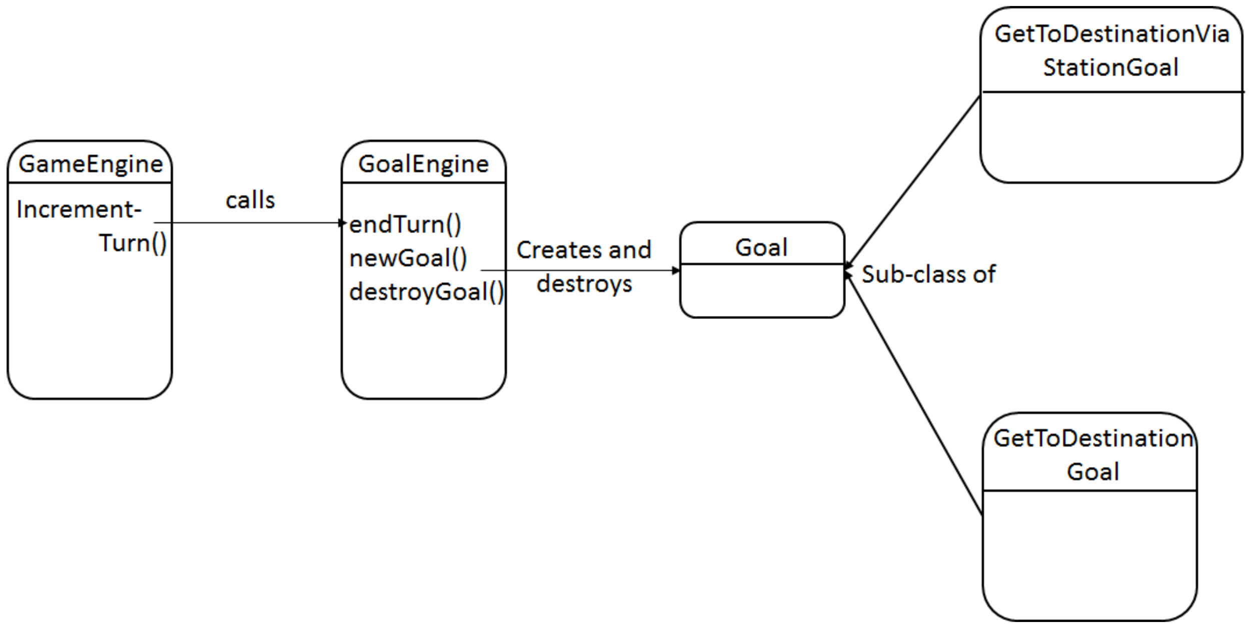
GameEngine:

The game engine creates instances of MapGUI, MapGraph, Player, and Playerinfo, on startup

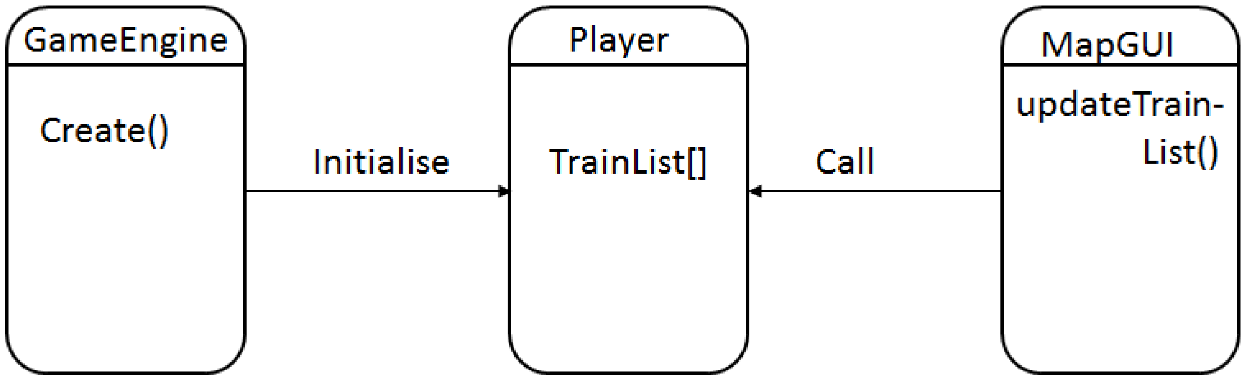


Goal engine:

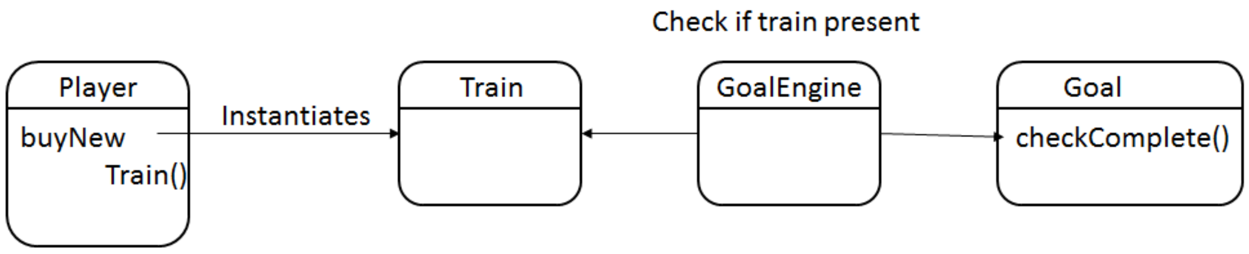
GoalEngine ensures that 3 instances of Goal exist at all times. Goal is extended by GetToDestinationGoal and GetToDestinationViaGoal. Goal itself is never instantiated. GoalEngine is called by GameEngine at the start of the game.



Player:
Player contains a list of trains assigned to the player. Player also passes the list of trains that are owned by each Player instance to MapGUI. Player objects are created by GameEngine at the start of the game.



Train:
Train is connected to GoalEngine as GoalEngine needs to know if a certain train fulfils a goal's criteria. Trains are created by the Player class, as there will never be trains which don't belong to a player.


Player
buyNewTrain()
Instantiates
Train
Check if train present
GoalEngine
Goal
checkComplete()