Team LYS

TaxE Game

Extension Report

TABLE OF CONTENTS

1. CHALLENGES ENCOUNTERED

Some of the problems that made extending the game more challenging include:

- Lack of comments and documentation within the code. Hence, the only way to understand the code was to either contact the developers or read the entire documentation and understand the overall code structure.
- Game requirements were not made available with the initial documentation on the website making it difficult to start further development.

1.1 Specific Challenges

1.1.1 Drawing Architecture

The image objects that represented map objects such as junctions were not developed such that the images representing could be changed in response to events in the game. This was a challenge because we had no way of communicating to the player if an event was broken (aside from the notification that flashes once on screen). Later changes were made to the codebase that allowed the sprites drawn to be changed at runtime, depending on if they were broken or not.

1.1.2 City Tooltip

When trying to play/test the game, the name of a city only appears when the mouse is positioned over the city, this can lead to a frustration as it is not reasonable to expect that players know and recall where the cities specified in the goal actually are. Interface testing became significantly more time consuming because of this.

Attempted to get around this by having the tooltips appear above appropriate stations when mouse hovers over a goal. This proved difficult as the goals were being rendered again with every frame. They were buttons being removed and readded which made it difficult to add a listener to them as the listener would be lost on the next frame.

When the implementation was changed to render the goal buttons only when they need to be changed (new turn or goal complete) we attempted to display the tooltips above the cities for the stations. At first this did not work and we were unable to figure out why. We decided to highlight the stations by changing the image. Eventually this caused a bug. We went back to tooltips when we figured out how to get it working.

1.1.3 ResourceManager

There is a lengthy call to the train Constructor “return new Train (train.getFirst(), train.getFirst().replaceAll(" ", "") + ".png", train.getFirst().replaceAll(" ", "") + "Right.png",train.getSecond());”.

The code is uncommented and so it is hard to tell what its’ purpose is. It manually copies the attributes for a particular train, and uses them as arguments to construct a new one. This simple usage can be simplified via the use of a copy constructor, which knows how to copy instances of

trains. We have created a copy constructor which removes ambiguity and reduces code duplication.

1.1.4 Method Parameter types

In many parts of the code a method requires the name of an object, and not the actual object. For example the Map method “doesConnectionExist” takes the name of two stations. This is inappropriate as a connection between two stations has nothing to do with the name of the station. It also means that the method is made longer than necessary, as the actual object must be found based on its name. These lookups are usually only O(n) (looping through a list of elements) but are unnecessary.
Calling these methods where this is the case also becomes verbose as you have to pass in “object.getName” rather than just “object”.
Due to the way the train/station json deserialization was implemented, it is not very easy to refactor some of these methods (as they really on using strings representing objects, and not the objects themselves).

1.1.5 LibGDX

As previously we had only used swing/awt for our graphical interface, we had to learn how to use the LibGDX library. As it is a purpose-built game library it has many features (such as actors representing objects on screen) that were useful, but meant that the architecture of the game was very different to our previous solution.
It was difficult to comprehend the binding of actors to their gameLogic objects, as in the code, the attributes of these actors/objects, were set/changed with no documentation to say what was happening. Sometimes this could be confusing as we were unsure whether to use a particular actor or object attribute as canon if it existed in both places. An example of this are the trains, when they are not at a station their location is set to (-1,-1), which meant you would have to look at the actor’s location to find its true position.

1.1.6 Route Editing

Originally, it wasn’t possible to change the path of a train once the route had been confirmed. The calculation of how far the train would travel each turn was calculated as soon as the route was created, and no methods to change this have been provided.
This made it tricky when implementing broken tracks and junctions, because there was no way to check whether a track or junction had been broken half way through the route, and make the appropriate changes to the train path. As a result the train would travel across broken tracks and junctions.
In order to work around this issue, we have altered the path creation method so that it will only build a route until it finds a broken track or junction and ignores any stations after this. We found that the train would

reach the broken track or junction and just jump to its final destination. This was due to the fact that the final destination was set separately to the route. This issue was sorted by setting the final destination to the most recent non-broken station during the route building process. We still had the problem that if a junction or track broke after the route had been created the train would still follow the initially set path. To fix this, the route build method runs at the start of every turn.

2. EXPLANATION AND JUSTIFICATION OF MODIFICATIONS MADE

2.1 Constants

We have chosen to use static final variables were appropriate because they are superior to “magic numbers” (where seemingly arbitrary values are employed), as their descriptive names convey their intended purpose, and as they as static and situated at the top of the class, they can be very easily modified, if developers want to change the behaviour of certain objects.
For consistency with standard Java practices we have ensured our constant names are in Uppercase, with words separated by an underscore.

2.2 Breaking CollisionStations **Tests:** I1, I2

The addition of a simple Boolean variable “broken” was added to the CollisionStation class, along with appropriate methods to get and set the whether or not a CollisionStation is broken.

We wanted junctions to be randomly broken or fixed after each turn. As the code to handle the transition between turns was found in the GameScreen class, it was sensible to add the code to this class. An alternative would have been to create a separate class, but since breaking tracks would only happen after turns it could have made the code harder to understand.

Four new variables have been added to the GameScreen class to ensure CollisionStations are modified as desired:

- JUNCTION_BREAK_PROBABILITY: This constant is compared to a randomly generated number, after which a decision is made to break a track depending on if the random value was above or below the value of the constant.
- JUNCTION_FIX_PROBABILITY: This constant is also compared to a randomly generated number, after which a decision is made to fix a track.
- BREAK_OR_FIX_EVERY_X_TURNS: This constant specifies the minimum number of turns that must pass before a CollisionStation is broken or fixed.
- lastBreakOrFix: This variable stores the turn in which a CollisionStation was last broken.

Hooks to the game's alert system were used so that players are notified when the state of a CollisionStation is changed.

2.3 Breaking Connections **Tests:** I11

A Boolean variable "broken" - along with the relevant functions to get, and set, the state of the "broken" variable for a particular station - has been added to the Connection class. This is the easiest way for the game to keep track of which connections have been broken, or fixed.

Similarly to junction breakages, connections were to be randomly selected and broken, or fixed, at the end of each turn. The code was added to the GameScreen class, as this is where the transition between turns was handled, and it would have been more confusing than helpful to create a new, independent class for the breakConnection and fixConnection methods.

Six new variables have been added to the GameScreen class to give the methods breakConnection and fixConnection the functionality they require to be used, and the ability to easily change some core aspects of the methods.

- CONNECTION_BREAK_PROBABILITY: This constant is compared to a randomly generated number, which is regenerated at the end of each turn, which gives the decision whether the game is going to break, or fix, a connection or not.
- LAST_CONNECTION_BREAK_OR_FIX: This variable stores the number of turns that have elapsed since the last breakage, or repair, of a junction, and is compared to the CONNECTION_BREAK_OR_FIX_EVERY_TURN variable to decide whether to break, or fix, a connection.
- CONNECTION_BREAK_OR_FIX_EVERY_TURN: This constant is used as a comparison for the LAST_CONNECTION_BREAK_OR_FIX variable, and specifies the minimum numbers of turns to have passed before a connection is broken, or fixed.
- BROKEN_CONNECTIONS: This is a variable that records the amount of broken connections in the game at any one time, increasing by one when a connection is broken, and decreasing by one when a connection is fixed.
- MAX_BROKEN_CONNECTIONS: This is a constant to limit the amount of broken connection in the game.
- LAST_BROKEN_CONNECTION: This variable keeps a record of the last broken connection to make the game decide which connection to fix, if indeed it decides to fix a connection.

Hooks to the game's alert system were used so that players are notified when the state of a connection is changed.

2.4 Setting the image of a CollisionStation **Tests:** I10

Previously the image representing a CollisionStationActor (the class that graphically represents a CollisionStation) was fixed. This meant that a

CollisionStation could only assume one image for the entirety of a game. As we needed players to be able to distinguish between broken and unbroken CollisionStations, this had to be changed.

Now, two variables for the location of the broken and unbroken image were created. This necessitated the creation of a factory method to handle the creation of a CollisionStationActor, as in Java you cannot use a super constructor in conjunction with a conditional (which would have been the simplest method, as we could change the image based on the value of the CollisionStation.isBroken Boolean). The super constructor was a call to an Image object that requires the path of the image to display.
The factory method now performs the conditional expression, and invokes the CollisionStationActor with the correct variable containing the location of the appropriate broken or unbroken Junction image.

Being able to set the image of a CollisionStation was arguably superfluous, as we could assume that the map/game always starts out with no broken CollisionStation, but we thought it would be wise to consider the state of the Station and set its' actor's image appropriately.

2.5 Updating the image of a CollisionStation **Tests:** I10

Previously, the CollisionStations in the game were rendered once, as they only had one, static representation. Now that a CollisionStation could be either broken or not whilst the game was running, there needed to be some way to update the look of the CollisionStation.

In the interests of performance, we did not want to naively call the method "stationController.renderStations()"; on every frame the game was updated, as the representation of a CollisionStation would not always be changed with such high frequency (and even if this was the case, not *every* Station would be changing state). Instead an updateImage method was created in the CollisionStationActor class, which would update the image depending on the state of the corresponding CollisionStation. This method would be called only when the broken state of a CollisionStation is toggled.

Adding this updateImage method required knowing the current state of the CollisionStation so it could change the image appropriately. A reference to the CollisionStation was added to the CollisionStationActor on construction for this reason (as previously the CollisionStationActor did not know what Station it was representing).

2.6 Goal Generation and Goal/GoalManager Refactoring **Tests:** I3–I9, U1–U3

Multiple related changes are described here, as it would not be appropriate to describe them separately since the changes were all made to fulfil one of the updated requirements (Quantifiable goals).

Random goals were already generated by the GoalManager class, so we created our new goal generation methods in there also. We chose to have three goal difficulties; easy, medium and difficult. Random numbers were used to generate goals of varying difficulties.

```
if (randomGoalDifficulty >= 0.8) {
    goal = generateDifficultGoal(turn);
} else if (randomGoalDifficulty >= 0.5) {
    goal = generateMediumGoal(turn);
} else {
    goal = generateEasyGoal(turn);
}

return goal;
```

The above code snippet replaced the code in the method “generateRandom”. As it still returns a Goal object, this change is transparent in respect to the rest of the system.

Three new methods to create goals of varying difficulties were made. In order to reduce code duplication, these methods were chained together, so for example the generateDifficultGoal method gets the goal returned from generateMediumGoal (which in turn used the goal generated by generateEasyGoal) and adds more constraints to it.

```
public Goal generateMediumGoal(int turn) {
    Goal easyGoal = generateEasyGoal(turn);

    easyGoal.addConstraint(resourceManager.getRandomTrain());

    return easyGoal;
}
```

The actual Goal class was modified slightly. Previously the “requiredTrain” variable was a string that contained the train’s name. This variable is used for comparison in the Goal “isComplete” method to check that a goal was completed by the correct type of train. The type of “requiredTrain” was changed to a Train, because we did not feel that representing a train as a different, unrelated object was not idiomatic object orientated code. Checking only the name for equality can also be misleading as there could potentially be two distinct types of trains with the same name.

The “addConstraint” method was refactored and split into two methods, taking advantage of Java method overloading. This is superior to the previous implementation of this function, which took two String arguments, the first of which represented the type of constraint object, and the second represented the name of that object to add as a constraint. If statements determined what to do with different types passed in. As with the aforementioned “requiredTrain” variable, checking objects via a string representation is not a good practice, as in this case, simply typing the type “train” wrong would result in “addConstraint” silently failing, as it would not know how to deal with the misspelled string. With our changes your IDE will notify you that your arguments are invalid when calling the function.

```
public void addConstraint(Station via) {
    this.via = via;
    score *= CONSTRAINT_SCORE_MODIFIER;
}

public void addConstraint(Train train) {
    this.requiredTrain = train;
    score *= CONSTRAINT_SCORE_MODIFIER;
}
```

A constant was used to limit the maximum distances for Goals in the GoalManager. This was so that the two randomly chosen start and destination points for a goal are never extremely far apart. A similar constant is used to ensure that the "via" destination is between the start and end point. These methods are flawed however, as we have not implemented any pathfinding algorithms to determine how close two Stations are. The distances are simply calculated as a straight line, which means that the reported distance is not always representative of the connections that actually link the Stations. The base score of a goal is equal to this distance, and every additional constraint added increases the score by a constant factor.

Previously the "isComplete" method for a goal looped through the history of a given Train to check for certain conditions. We have extracted this loop to the Train class as the method named "historyContains" which takes a Station as an argument, returning the result as a Boolean. This means that we can check a Station is in the history of a train as a one-liner method call elsewhere, useful as it is required three times in the "isComplete" method alone.

2.7 More about Quantifiable Goals **Tests:** I3–I9, U1–U3

- Easy goals are of the form "Send any train from station A to station B", where A and B refer to distinct, random stations.
- Medium goals are like easy goals, but have a train constraint and so are of the form "Send a specific train from station A to station B", where the specific train is a randomly chosen train.
- Difficult goals are like medium goals, but have an additional route constraint. They are of the form, "Send a specific train from station A to station B via station C" where C is another random station.
- Easy goals are generated more frequently than medium goals, which in turn are more frequent than difficult goals.

2.8 Player Scoring **Tests:** I12-I15

Calculating a players' score is easy as a result of the new Goal implementation. Once a player has completed a goal, its' score is added to the player's score. When the game has been finished, the player with the highest score is declared the winner.

2.9 ResourceManager Refactor

The ResourceManager class was restructured in the interests of future maintenance and extensibility.

As there is no variation in the trains at runtime, all of the methods were made static, as there is no need for the ResourceManager manager to be instantiated as it has no state (the resources generated would always be the same no matter what instance generated them). For the same reasons the list of trains was made static.

The list of trains “trains” now actually contains train objects, instead of Tuples containing the name and speed of each train. This change was made as it is self-documenting, and means that it is easier to return new train instances from this list.
getRandomRescource was renamed to getRandomTrain as that is the only thing it can return (since trains are the only resource implemented)
A Train copy constructor was added, which is used when returning random resources. This is more readable than creating a new train by passing in all required arguments to the constructor.

2.10 Vowel Class **Tests:** I16

This class is used to check if a string starts with a vowel. This is used in the Goal toString method to ensure that the correct article (a/an) is used depending on the name of the train. A separate class was created for this since it may be reused by another toString method, maybe if describing power-ups etc.

2.11 Goal Rendering **Tests:** P1

Goals were previously rendered every frame. In the interests of performance this was changed so that goals are only re-rendered when the goals change or new turn begins. This change also allows a ClickListener to be added to the buttons. (See Highlighting Stations)

- Added a Boolean in Game class to represent when the goals need to be re-drawn
- Set Boolean to true in turn listener in Game and afterAction() in TrainMoveController (goal completed after turn has changed)
- In the render() in GameScreen, call showCurrentPlayerGoals() if Boolean is true, then set to false.

2.12 Displaying Station Names **Tests:** I17

The first attempt was to display tooltips above the stations similar to when the mouse hovers over a station. When this didn’t work we tried changing the station image, this went well but eventually caused an unknown bug, we removed it when we figured out how to fix the tooltips.

- Added ClickListener to buttons displaying goals for displaying the station tooltips on hover.
- New image was created for stations to highlight them (yellow border instead of black).
- Created methods in StationActor to change the image between normal and highlighted.
- Added code in ClickListener.enter() and ClickListener.exit() to change the Station image when hovering over the goals.

2.13 Route Management of Broken Junctions and Tracks **Tests:** I1, I2, I11

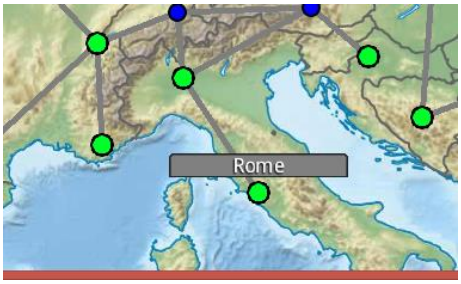
Broken junctions and tracks had been implemented such that the game was aware which routes were unavailable, and wouldn't allow invalid new routes to be created. This didn't work properly because if a route had already been created before a junction or track broke, the train would continue on its' course.
The initial idea was to check if the next station or track in the route was broken, and then to pause the route until it was fixed. We discovered that this wouldn't be possible because once a route was created the train movement was hard coded and couldn't be changed.

- Edit the addMoveActions method and rename to "refreshMoveActions"
- Iterate through train route to check for the first broken track or junction
- Remove any stations later in the route
- Set the trains final destination to the final station of the edited route
- Run refreshMoveActions every turn, to rebuild the route and check for new broken junctions or tracks
- Add if statement to ensure track builder doesn't run after the train has reached its' final destination and has nothing in its route.

2.13 Path Finding **Tests:** I18

When playing the game, we would often come across situations in which the "via" station was not in-between the origin and start destination. This was especially a problem if the "via" station was actually positioned at the edge of the map with only one connection.

As you can see in the screenshot (right), having Rome as a "via" destination is not a very good choice from a gameplay perspective, as although a player can route in, and then back out of the "cove" the GUI presents ambiguities, in the sense that there is no indication of routes that contain connections more than once.

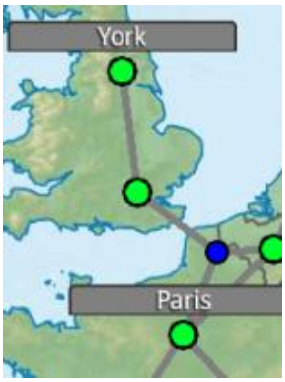


These types of goals could be generated as the GoalManager only considers the straight-line distance between two points, only making sure that points are an acceptable distance from each other, which without considering the connections in-between leads to these poor decisions.
In an effort to make these routes better we decided to port some of our pathfinding code that we wrote for assessment 2. It was relatively easy to do so (since our previous project was also written in java). The only large conceptual differences between the two implementations is that this game considers routes to start and end at a Station, whilst the previous implementations considered Track objects to be components of the route. The relaxed routing requirements of this game (compared to our previous solution) meant that the ported code did not even need to consider the angles between Stations (as our previous game mandated that trains may only

move to adjacent junctions providing that the angle of the tracks fell within a certain range).

With this implementation generating goals became much simpler. As the map computes and stores the shortest routes between all Stations on instantiation, we could use this information to ensure that a goal has a minimum number of stations. This helps balancing as we no longer have scenarios when the origin and destination are right next to each other.
In addition to this choosing via routes is incredibly simple as we can just pick a random station on that route (that is not the origin and the destination). This means that confusing situations such as the one depicted above will never occur.

The score of each goal is now the length of the shortest route between the origin and destination, rather than the straight line distance between the two. This makes for fairer gameplay.



In this simple example (right), it is easy to see disparity between the straight line distance between York and Paris, and the actual distance between them if travelling station by station. This disparity can grow as the distance between the two stations become further and further apart.

3. EXPLANATION AND JUSTIFICATION OF SOFTWARE ENGINEERING SOLUTIONS AND APPROACHES

Working on this project we continued to use some approaches we used on our last project and used some new solutions used by team FVS.

The new techniques we acquainted ourselves with were Gradle dependency management and libGDX game development framework. With Gradle we used plugins for the IDE's to work with it. Any other queries were looked at online or with the help of FVS. LibGDX was relatively simple to get the hang of, however we didn't have to worry much as FVS had most of the framework set up as needed. Any adjustments needed were done by looking at libGDX documentation and mimicking code FVS wrote.

We continued to use gitHub to manage the project as FVS had also used gitHub which allowed us to create a fork of their project. This allowed us to look back at how their work built up to the project we needed to work with and help understand their implementation.

The tasks for this project were split up and allocated to team members. However we used pair programming whenever anyone was stuck on a problem, and discussed it as a group if it was still difficult to solve.