TaxE Game

Extension Report

1. CHALLENGES ENCOUNTERED

1.1 Documentation

Even though the code we had inherited for the 4th assessment was inherited from the same source in the 2nd assessment, the deviation of changes that we had made in comparison meant that we did not fully comprehend all of the changes that were made. Having access to the documentation created by team XYG (the last maintainers of our current codebase) made it easier to get to grips with the new functionality introduced.

1.2 Train Route Movement

When the team received the required changes from our clientele, we got together and brainstormed ideas for how we would implement them. One of the required changes was to add the ability for players to add and remove tracks from the map. Immediately we saw the value that could be added from additional tracks (that is, making routes potentially shorter, so they can complete more goals.). But the ability to remove tracks didn't seem like a very obvious use case. We struggled to see why a player would want to remove a track. After some time we thought of giving tracks health, which could be depleted by passing trains. It would then make sense for a track to be removed when it had no health left. This in-conjunction with a repair track feature seemed to give purpose to the removal of tracks.

However, the way that trains were routed made this a problem. In our final implementation, players could remove their additional tracks, but only if that track was unoccupied and no other trains were routed to pass that track. This was necessary as the moveActions (LibGDXs way of buffering future movement of "actors" used by trains) were set at the same time as the route was set, and not actually when the trains moved. If we simply removed tracks without checking, trains would appear to travel without tracks - a feature we didn't want to include in our game.

Given more time, we would have liked to change the train routing system so that train moveActions could be refreshed dynamically, responding to track removal. We felt that this would have been a much more organic way of handling the removal of tracks. It also would have given players an additional incentive to repair their tracks, as they would no longer be able to earn rent on a track that has been removed due to no health.

Nevertheless our final implementation works well. As we could not remove a track after it had reached 0 health, we decided to reduce train speed linearly with track health. Furthermore, the concept of tracks having health was also used for the repair system, and calculations for rent (more rent is demanded the healthier a connection is). Furthermore, as tracks may not be built over existing tracks, players can come across situations where tracks need to be removed in order to allow new tracks to be built. Although there is not as much incentive to remove tracks as we had hoped, there is enough freedom in our map such that track creation has become a very diverse and innate function of our game.

1.3 Poor MVC Architecture

Upon inheriting the codebase for this assessment, it was obvious that the game was intended to have a Model View Controller architecture. This is evidenced by the plethora of controller classes for game objects such as trains and stations, as well as the separation of a gamelogic package, which is distinct from the fvs.taxe package. When implementing our replay functionality, we saw a conflict with the events we intended to play (created from the perspective of playing the game, for example clicking trains and other game objects), how we would replay these in a replay mode.

This was because game logic was tied to the dialog boxes. This was problematic as we want to replay game events systematically instead of through dialog boxes, which the players would interact with during play. Game logic needed to be independent of its invocation, whether it is via dialog boxes or systematically via a replay manager/controller. To alleviate this problem we extracted all game logic from dialog boxes, and encapsulated it within the appropriate controllers.

```
@Override
public void clicked(InputEvent event, float x, float y) {
    if (Game.getInstance().getState() != GameState.NORMAL) return;
    System.out.println("train clicked");

    // current player can't be passed in as it changes so find out current player at this instant
    Player currentPlayer = PlayerManager.getCurrentPlayer();

    if (!train.isOwnedBy(currentPlayer)) {
        context.getTopBarController().displayFlashMessage("Opponent's " + train.getName() + ". Speed: " + train.getSpeed(), Color.RED, 2);
        return;
    }

    if (train.getFinalPosition() == null) {
        context.getTopBarController().displayFlashMessage("Your " + train.getName() + ". Speed: " + train.getSpeed(), Color.BLACK, 2);
    } else {
        context.getTopBarController().displayFlashMessage("Your " + train.getName() + ". Speed: " + train.getSpeed() + ". Destination: " + train.getFinalStation().getName(), Color.BLACK, 2);
    }
    DialogButtonClicked listener = new DialogButtonClicked(context, currentPlayer, train);
    DialogResourceTrain dia = new DialogResourceTrain(train, context.getSkin(), train.getPosition() != null);
    dia.show(context.getStage());
    dia.subscribeClick(listener);
}
```

Figure 1 The clicked method of the original implementation of a dialog box, invoked when a train is clicked.

Figure 2 The revised implementation of dialog's clicked method.

```
@Override
public void clicked(InputEvent event, float x, float y) {
    context.getTrainController().selected(train);
}
```

Note how the new dialog box simply forwards the fact that the train has been selected to the relevant controller. Moving the game logic into the relevant controllers made making changes to the code easier for two reasons, the code concerning a particular game object could be found in one place (E.g. Train game logic goes into the TrainController), and there was no code duplication, which would be required as replay events need to use the same logic again.

Having game logic encapsulated as appropriate now, made our replay system implementation quite straight-forward, as it was easy to create listeners for events that were fired, which trigger methods on appropriate controllers in the same manner. This means an event will be responded to identically whether it came from a dialog or the EventReplayer.

```
EventReplayer.subscribeReplayEvent((event, object) -> {
        if (event == GameEvent.SELECTED_GOAL) {
            Goal goal = (Goal) object;
            selectedGoal(goal);
        } else if (event == GameEvent.SELECTED_GOAL_DISCARD) {
            Goal goal = (Goal) object;
            discardGoal(goal);
        }
});
```

Figure 3 An example of the replay events being used by the goal controller.

2. EXPLANATION AND JUSTIFICATION OF MODIFICATIONS MADE

2.1 Track Modification

2.1.1 Adding/ Removing Connections **Tests:** I1, I2, I3, I6, I7, U13, U14, U23

In order to achieve the functionality required as part of the update brief, we implemented the ability for players to be able to add connections to the in-game map.

In the interests of playability, we decided to restrict the ability to place new connections in a few ways. The first was making sure that any new connections did not overlap any existing ones. This decision was made because we thought it would be confusing if connections overlapped, but had no way to traverse from one to another (as there is no functionality to add junctions or stations). Without this restriction there is also the possibility to add so many connections in a small area that the map starts to look less aesthetically pleasing.

Adding this restriction also makes it so that the map designer does not have to create a list of valid connections, which could take a long time. We have implemented a blacklist of connections however, to make sure that connections crossing the sea cannot be made. Having a blacklist also means that creating certain connections can be prevented if we feel that it would affect the fairness/playability of the game.

Players can also only create one connection per turn. This decision was made for balance purposes, as it prevents players from simply out purchasing their opponents in an effort to win.

Once a player has added a connection, it is owned by that player, and can only be removed by that player. The default connections of a map cannot be destroyed. There is an additional restraint on connection removal, so they cannot be removed whilst a train has that connection in its route.

2.1.2 Damaging Connections **Tests:** U4, U5, U6

When trains move over connections, they damage that track. The damage inflicted is a function of the trains speed (faster trains will inflict greater damage) and the connections strength. The more damaged a connection is, the slower any train will be when travelling across that connection. Players can check the health of their connections when it is their turn, and repair them as they see fit.

2.1.3 Repairing Connections **Tests:** I4, U7, U15, U16

As connections can be damaged, we thought that it would be unfair if we did not allow players to repair them. We also think that needing to maintain connections will force players to consider the amount of connections that they add to the map, as having many connections could be a burden.

Players can use the money they have to repair the connections that they own. We allow the player to upgrade their connections in increments of 33%, which may be helpful in the case that a player does not quite have enough money to fully repair a connection. The cost of repairing a connection is a function of the length of the connection, and how much they want to repair it.

2.1.4 Connection Materials **Tests:** I9

There are now three types of materials in this game; gold, silver and bronze. Every connection must be made out of one of these materials. All of the default connections in the map are gold. Gold connections are the best material in the game, as they are not damaged by passing trains. Connections made out of silver or bronze can be damaged. While bronze connections are weaker than silver connections, they also cost less to repair.
Connections are shown in different colours based on their material.

2.1.5 Connection Upgrade **Tests:** I5, U1, U2, U3, U8, U17, U18

As different connections have different properties, we allow players to be able to upgrade their connections from one material to another. The cost of upgrading a connection is a function of the length of the connection, and the material that they want to upgrade it to.

2.1.6 Connection Management User Experience **Tests:** I1, I2

Creating a connection is done by selecting the two stations between which the track should run. These selected stations are highlighted, and rendered as red if it is invalid, otherwise it is green. The colour coding should make for an intuitive experience. The "Create Connection" button, used to confirm the creation of the connections only appears when it is valid.

Buttons that represent the tracks that a player has created are shown below their trains during their turn. The button has text describing the two endpoint locations, as well as the health of the connection. Hovering over a button will highlight the corresponding connection, making it easier to identify. A connection can be selected by clicking the button, invoking a dialog with the options to upgrade, repair or remove the connection. Certain options will be hidden when inapplicable (e.g. a Gold connection has no upgrade option, and a track at 100% will have no repair button visible).

2.1.7 Connection Rent **Tests:** I8, U11, U12, U19, U20, U21

When playtesting the game after we added connections, we noticed that while all players in the game benefited from the creation of the additional connections, the connection owner gained no benefit and had to take on the responsibility of repairing those connections. This meant that other players in the game could 'freeload' from other players efforts (and money invested).

To alleviate this, we added a rent system. If a player's train travels on a connection owned by another player, that player has to pay them rent. The amount of rent payable is proportional to the length and material of that connection. Longer connections cost more to rent, as do more expensive materials. This adds another layer of strategy to the game, as players must now gauge if the payoff of a shorter route is still worth it, if that route contains connections owned by another player.

While players cannot ordinarily perform actions that mean that they will be left with negative funds, the payee will always be deducted the amount of rent payable. This is so that players cannot travel along other's connections at no cost, since they would still benefit at the end of their journey as a result of their goal pay-out.

2.1.8 Money **Tests:** I10

As managing connections requires funds, we decided to make the amount of money the player has their score. This works quite well, when you consider that contracts have monetary pay-outs, as is the case in real life. We felt that having a score distinct to money would favour players who do not necessarily consider the amount of money they spend, as this decision forces players to regulate their spending.

Players are given a certain amount of money at the beginning of the game, so that they can create connections before completing any goals if they wish. A small amount of additional money is granted each turn.

2.2 Replay Mode

2.2.1 Replay Mode Invocation **Tests:** I11, I12, I13, I14

- Once the Game Over Screen appears, there is a prompt to view replay
- User presses play to get started
- User can pause in between the replay
- User can also change replay speed

These changes were made to incorporate the requirement changes after assessment 3. We have discussed below the impact and challenges encountered while making these changes.

2.2.2 Additional Classes and Implementation

A **GameEvent** enum was added that represents all the things that can happen in a game. It also includes the delay while rendering the event. Figure 4 shows this.
Each event can have a distinct delay. This is so we can make certain operations slower or faster. For example, the assignment of trains and goals are instantaneous, but the end turn event has a longer delay so the players can see the trains at rest before they begin moving.

ReplayEvent is a class containing a GameEvent and an Object. The type expected for object varies from event to event. For example the SELECTED_TRAIN event has a Train for its object, whereas ADD_GOAL has an object which is a list, consisting of the player that the goal was assigned to, and the goal itself. Not every event needs an object, as the subject can be implied. This is because our replays always play linearly without moving backwards or skipping turns. For example the CANCEL_PLACE_TRAIN event doesn't need to save the train as an object, as it is obvious from the previous events what the selected train was. This decision saves space.

The **Event Replayer** is responsible for recording and subsequently replaying all the **ReplayEvents** saved while the game was played. It includes the following functions:

saveReplayEvent: called by appropriate controllers to log the events, for example trainController : saveReplayEvent(GameEvent.SELECTED_TRAIN, train). This saved replayEvent is then added to a list for recall.

subscribeReplayEvent: controllers can listen to these replayEvents when they are fired. Their listeners will recreate the appropriate action. In almost all cases the listener simply calls the method that originally generated the ReplayEvent with the same arguments

provided in the object, meaning the game is identical to that of before, and there is a minimal amount of changing how the game is run when it is being played.

```
EventReplayer.subscribeReplayEvent((event, object) → {
        if (event == GameEvent.SELECTED_GOAL) {
            Goal goal = (Goal) object;
            selectedGoal(goal);
        } else if (event == GameEvent.SELECTED_GOAL_DISCARD) {
            Goal goal = (Goal) object;
            discardGoal(goal);
        }
});
```

Above is an example of code used in the goal controller that listens to events concerning goal selection and discarding. These methods are the same methods called when a goal button is clicked in normal gameplay.

Exceptions to listeners calling their original methods include:

- Goal generation: original functionality must be suspended during replay mode, or else new goals will be created. A new method must deal with reassigning the goals, and also resetting the goal to its original, uncompleted state.
- Train generation works in a similar way to the goal generation. Actors belonging to the train must be reset, along with other its other attributes such as, its current station and history so that the “new” train object is identical to when it was originally assigned.
- Connections: When these are re-added they must be reset to full health, and added in the same material they were bought in, so that gameplay remains identical.
- Junction Failure: Again, since junction failures are random, we had to suspend original functionality, since we did not want junctions other than those that failed in the game happening.
- Most dialogs listen to the events corresponding to those originally fired when its buttons were pressed. They then close on receipt of these events. This is necessary as the dialogs aren't actually clicked in replay mode, as the dialogs would remain on screen otherwise.

2.2.3 Limitations of the replay system

There is only one chance to view a replay, which is right after the game has ended. This is because we could not find a way to access our instance of the game from the Main Menu.

The movement of trains on screen is always the same as the speed they originally moved, regardless of the replay speed. This was done to guarantee that the physics of the game were stable, preserving the deterministic nature of the game. As our replay system is event based (rather than screenshots of the game being recorded), a train completing a goal at a different time could affect the outcome of the game.

2.3 Miscellaneous Changes

2.3.1 Dialog Box Reduction

We found the number of dialog boxes to be quite annoying. Trying to complete simple tasks required far more clicks than necessary. As a result we made some changes to how processes work to reduce clutter.

Clicking a train now goes straight to train placement mode instead of a dialog. The option to discard a train is now an ‘X’ next to the train. This is to eliminate dialogs when placing a train. However, there is a dialog for discarding a train to avoid user annoyance as they might click the ‘X’ by accident.

When selecting trains at stations, a dialog box would appear even if there was only one train at the station. This has been changed so that where there is only one train at a station, clicking the station will select that train.

Selecting trains on the map would normally bring up another dialog box, where the user can choose to enter routing, discard the train or cancel their selection. This dialog box has been removed. Routing is now the default action upon selecting a train, and the discard and cancel buttons have been moved to the top bar.

2.3.2 Removed Jellies

When we inherited our current codebase, Jellies were an obstacle in the game. They would increase in speed as the game progressed, and the player had to avoid them, so their trains would not collide and be destroyed.

Initially we thought that this feature was good, but after playing the game for longer periods of time, the jellies quickly began to be a frustration. Their random movement does not complement a turn based game very well, as it is not possible for players to react and change the route of their train whilst they are already moving. We felt this lack of control would make for a less enjoyable game.

2.3.3 Map Changes

The first thing we noticed was that the map image felt old fashioned and outdated, and looked odd alongside the simple design of the GUI.. This was replaced by a cleaner, two-colour map image to fit in with the rest of the GUI. Station and junction positions needed to be moved accordingly.

After we had implemented user track creation, we found that the map was very cluttered. Stations were connected to almost every possible surrounding station which meant that track creation was an expensive process which was irrelevant.

To start, we removed a large number of existing tracks. We decided that there was no need for all stations to be linked to each other. The map has now been split into four distinct zones: North, East, West and Central Europe. These sections of the track are not connected to each other, which means that trains are unable to travel between zones unless tracks are built by players. As a result, we had to make our method of calculating the fastest route between destinations more sophisticated. It had to be able to detect where there was no connection between zones and increase the reward accordingly. Goals which cross different zones will give a higher reward if there is no

connection between them. These changes should encourage players to build tracks which will add depth to the game, as it allows for a much more diverse map which will have major sections built by players as opposed to being pre-set.

2.3.4 Route Highlighting and Train Selection **Tests:** I15

After playtesting the game we discovered that there was no way to check the current route of the train. We also discovered that clicking a train would always enter the routing mode, clearing the player's previous route set. This caused a lack of clarity within the game, as it was very difficult to distinguish which trains were assigned to specific goals, and where each train was heading.

Hovering over a train will highlight its current route. This allows players to have a greater understanding of their trains. In conjunction with this addition, we have changed the responses to clicking trains, so that the routing mode is only entered without information if the train has no route set. In the case where the train does have a route, a dialog box which will pop up asking the player if they would like to change it. This dialog also lets the players know the current final destination of their train.

2.3.5 Current Player **Tests:** I16

The bar at the top was used to display information about current player, goal completions, train collisions, junction failures and other warning messages involving routing and track creation. With so much information being displayed through a single output, we found it difficult to tell what was going on.

When nothing was happening, the bar was used to display the current player and the amount of money they had. The main issue caused by this was that there was often a lack of clarity as to whose turn it was. Both players and money were already being displayed on-screen at all times, so we decided to remove the information from the bar at the top and instead highlight the current player in white text. This made it far more obvious who the current player was, and also reduced clutter at the top of the screen. As a result, the rest of the information also became much clearer.

3. EXPLANATION AND JUSTIFICATION OF SOFTWARE ENGINEERING SOLUTIONS AND APPROACHES

3.1 Using Commits with Git

As a team we made an effort to reduce the size of commits so that only one set of related changes were made at a time. This is because we have realised from previous assessments that it can be frustrating working on a large feature, only to rollback a large amount of uncommitted changes because something has went wrong.

Committing early and often gives you the benefit of being able to roll back to a known stable state that is not too far in the past, which can be invaluable when trying to find the source of a bug. The smaller changesets mean that there are less lines of code to scour for changes, and also means that the commit messages are more informative as they can be more succinct.

3.2 Git Branches

As we would have more than one feature being developed at one time, we decided to use branches in our git repository. This meant than one (or more) person(s) could work on distinct features. Not only does this mean that our commits are easier to digest, as logically related commits would appear together, but it also meant that the master branch was always stable, which is important as we can use it as a point of reference.

Merging branches could occasionally result in merge conflicts, that need to be resolved manually, but in most cases the changes were trivial to make. All in all merging code after implementing a feature is easier than potentially having to resolve them mid-implementation, as could be the case if two unrelated, substantial changes were made on the same branch.

3.3 Managing Dependencies with Gradle

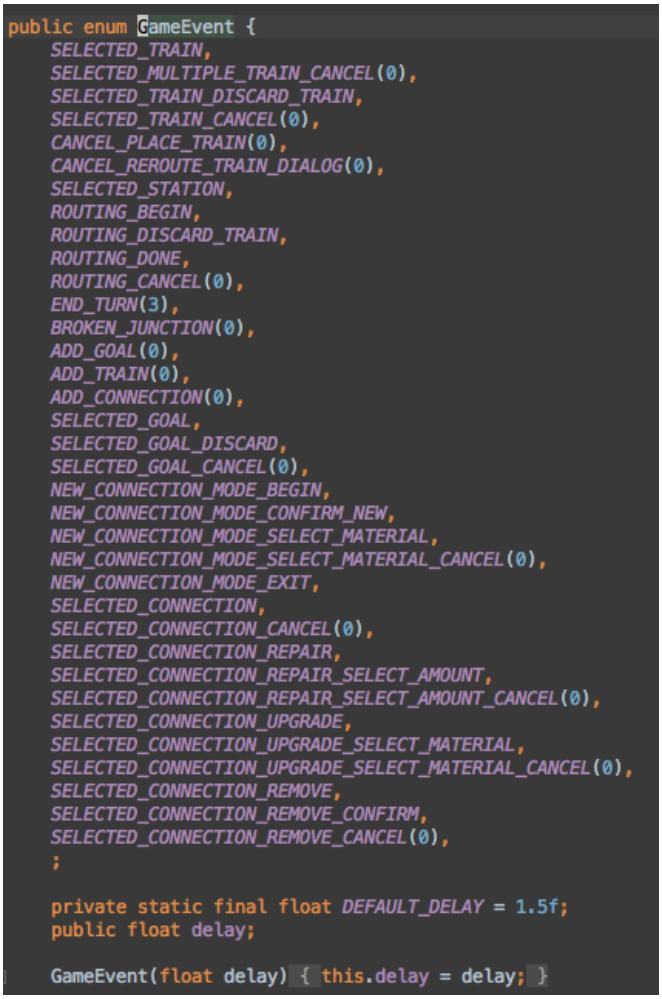
The codebase that we have inherited used Gradle for dependency management. We have continued to use it as it works well, negating the need for any team-member to individually download libraries necessary for our game to run. As the code-base is now quite mature, we have not needed to add or modify any of the libraries in use for our project, so we have not really had a need to make any changes to the Gradle configuration.

Most of our team use Intellij IDEA, which is a IDE that offers (limited) Gradle support. One small gripe throughout this assessment is that we cannot build our game via IDEA with Gradle, so we have to use Eclipse for this task. As exporting jars is not a task done very often is has not been a large problem.

3.4 Refactoring

As many new features were necessary for the completion of this assessment, lots of new code was being frequently committed to our repository. This of itself is not a problem, but we found that our initial implementations of features were not always written in the most efficient way. Having cycles in which we alternately wrote new functionality and then took time to refine it improved the quality of the code greatly.

Refactoring in conjunction with pair programming also makes it easier for other team members to be aware of changes made by other team members. While having specialised knowledge of a certain portion of the code base (that you have probably written) is good, it is also important to know details about other parts of the code. The more team-members know about the code-base the better, as troubleshooting bugs can be made much easier, as different people approach problems differently and the chances of one person knowing a solution increases. Also refactoring with someone who did not write a certain section of code is useful as they can highlight areas which are poorly commented (if they cannot grasp how a block of code works).



```
public enum GameEvent {
    SELECTED_TRAIN,
    SELECTED_MULTIPLE_TRAIN_CANCEL(0),
    SELECTED_TRAIN_DISCARD_TRAIN,
    SELECTED_TRAIN_CANCEL(0),
    CANCEL_PLACE_TRAIN(0),
    CANCEL_REROUTE_TRAIN_DIALOG(0),
    SELECTED_STATION,
    ROUTING_BEGIN,
    ROUTING_DISCARD_TRAIN,
    ROUTING_DONE,
    ROUTING_CANCEL(0),
    END_TURN(3),
    BROKEN_JUNCTION(0),
    ADD_GOAL(0),
    ADD_TRAIN(0),
    ADD_CONNECTION(0),
    SELECTED_GOAL,
    SELECTED_GOAL_DISCARD,
    SELECTED_GOAL_CANCEL(0),
    NEW_CONNECTION_MODE_BEGIN,
    NEW_CONNECTION_MODE_CONFIRM_NEW,
    NEW_CONNECTION_MODE_SELECT_MATERIAL,
    NEW_CONNECTION_MODE_SELECT_MATERIAL_CANCEL(0),
    NEW_CONNECTION_MODE_EXIT,
    SELECTED_CONNECTION,
    SELECTED_CONNECTION_CANCEL(0),
    SELECTED_CONNECTION_REPAIR,
    SELECTED_CONNECTION_REPAIR_SELECT_AMOUNT,
    SELECTED_CONNECTION_REPAIR_SELECT_AMOUNT_CANCEL(0),
    SELECTED_CONNECTION_UPGRADE,
    SELECTED_CONNECTION_UPGRADE_SELECT_MATERIAL,
    SELECTED_CONNECTION_UPGRADE_SELECT_MATERIAL_CANCEL(0),
    SELECTED_CONNECTION_REMOVE,
    SELECTED_CONNECTION_REMOVE_CONFIRM,
    SELECTED_CONNECTION_REMOVE_CANCEL(0),
    ;

    private static final float DEFAULT_DELAY = 1.5f;
    public float delay;

    GameEvent(float delay) { this.delay = delay; }
```

Figure 4: showing the list of events along with their distinct delays in brackets.