FVS

Architecture

Our three primary aims were:

1. Create a working implementation which meets all requirements
2. Our code should be clean and readable
3. Our code should be easily extensible

We managed to complete our first aim with little sacrifice to the second and third aims. Our third aim was achieved mostly down to design decisions which we made early on and stuck to even if it added some extra development time in the early stages. We believe the extra effort in the early stages will result in higher quality code, and quick implementations of features in the latter stages of development.

Project Split

The project is split into packages, the first package being gameLogic, this contains all the entities that exist in the game, Trains, Goals and is completely separate from the client side implementation. “fvs.taxe”, is the core code for all the front end.

Finally there is a Desktop project, this package/project is given by default by libgdx and only contains one class. The library supports launches for mobile and web, if we wanted to support this they could also have their own separate project, and this split allows for any platform specific code to remain separate, for example android notifications could go in an Android package if we wanted to support that platform.

Front End (fvs.taxe)

Figure 1 shows a high level overview of the front end package of our project. Illustrating the hierarchy and interactions between, Screens, Controllers, The Stage and Actors.

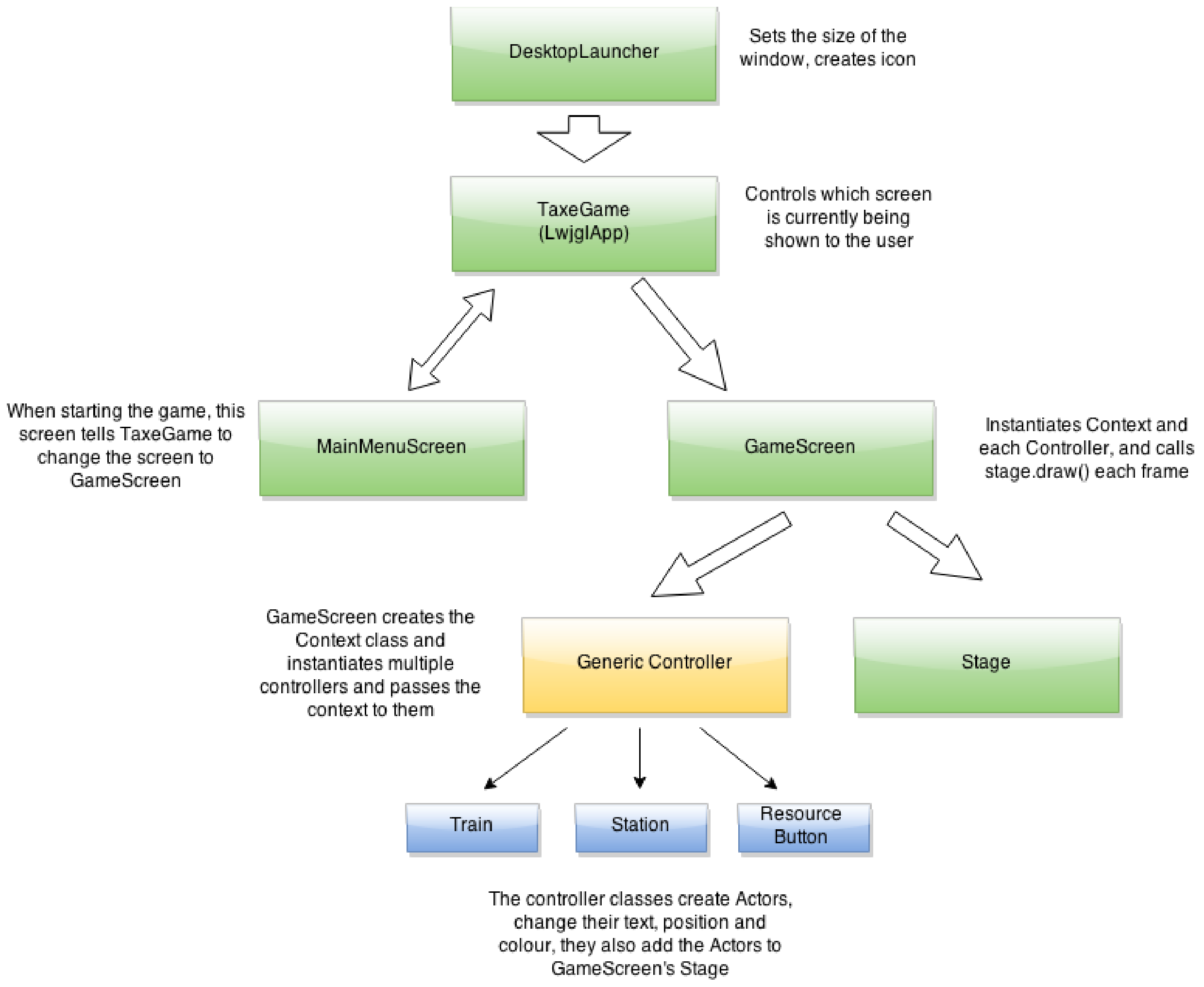
DesktopLauncher
Sets the size of the window, creates icon
TaxeGame (LwjglApp)
Controls which screen is currently being shown to the user
When starting the game, this screen tells TaxeGame to change the screen to GameScreen
MainMenuScreen
GameScreen
Instantiates Context and each Controller, and calls stage.draw() each frame
GameScreen creates the Context class and instantiates multiple controllers and passes the context to them
Generic Controller
Stage
Train
Station
Resource Button
The controller classes create Actors, change their text, position and colour, they also add the Actors to GameScreen's Stage

Figure 1 - Front End diagram

Library Choice

Originally we had intended to use Slick2D as the game engine, as we were aware of it's wide usage in the past for similar projects. We however noted that Slick2D's development and maintenance had ended for the main part which gave us some concerns. We instead opted to use LibGDX as the main game engine as it has gained widespread usage in recent years within industry and so will be more suitable to our needs to be provide a game which can be built upon and supported. The main concepts of the game however do not need to be altered as both Slick2D and LibGDX are wrapped around Lightweight Java Game Library (LWJGL) so the underlying principals in each library are very similar.

LibGDX Components

Screens span the whole window, the current screen is set via TaxeGame, and this screen has a render() method which is called every frame. Our first screen, MainMenuScreen is very simple, two rectangles, one of them exits the game, the other changes the screen to GameScreen, which is the only screen used to play the game.

Actors are the view components in LibGDX. Which in our use case is mainly for buttons and images. Actors have built in functionality for handling clicks and mouse hover events which we have used.

Actors are added to a **Stage**, which is stored in the GameScreen. Each passing frame calls GameScreen.render() which calls Stage.draw(). This then iterates through each actor stored in the stage container, and calls Actor.act(). The stage knows how much time has passed since the last frame, and passes this information on to the actors which can then use this to work out how much they should move by (if they happen to be moving).

Controllers

In general our controllers are responsible for:

- creating and manipulating Actors
- dealing with user interactions
- displaying textual information on screen

GoalController and ResourceController provide information about the current users goals and resources (trains) on screen for the player to see and interact with. StationController provides information about all stations and junctions on the map, generating the actors which represent each, whilst TrainController handles the display of trains at stations and on routes around the map. TrainMoveController, however, handles the movement of trains between stations by applying actions to them, and RoutingController handles the user interactions necessary to route the train to the correct location. TopBarController handles the display of the bar seen at the top of all screens and provides an interface for other controllers to utilise its functionality.

Context

This is created in the TaxeGame class, and passed through to all of the controllers. It is global state so I've tried to be very weary about adding too much to this class.

I would argue the following are justified in being in the class because of their extensive use in a wide variety of locations:

- TaxeGame - as this class contains batches for writing text and drawing images[1] on the screen, it is also used in a few places
- Stage - many controllers need access to the stage to add the variety of actors which they create
- Skin - Dialogs, buttons and tooltips all need access to the skin file to define their colouring and fonts, we used the default libgdx skin
- Game - Game is a singleton (justified in the Back End section of this report). My reasoning behind adding this to the context class rather than letting callers get access to it via its getInstance() was to reduce calls to that method, although this doesn't stop the

[1] Ideally all images, font and lines drawn would be Actors, and then every GUI component added or drawn could be added to the stage without the need for the context to store a reference to the TaxeGame class, but we didn't find a way of doing this

class being a singleton, it has made it easier to refactor Game into a normal class, which I like to think we would have done if we had more time

The two controllers which are stored in Context do have reasons for being there, but it is still a bit of a code smell. TopBarController is in Context because it contains a method for giving feedback to the user (the text which appears in the top bar for ~2 seconds and then fades), which is used in a variety of places: goal completion, clicking an invalid route, changing the turn, showing the speed of a train.

The other controller stored is RouteController, our justification for this, is given that the way that routing begins is complex far removed from the controller itself:
Currently the station actor has a click listener which opens a dialog asking which train you want to manage, the train buttons in this dialog have click listeners attached to launch the train's manage dialog. Then if you click the route button, it informs the RouteController to begin routing mode.

An improvement could be the click of the routing would publish some StartRouting event in a location "high up" enough that RouteController could have access to it and subscribe.

Dialog
Dialogs are used to present the player with information and then take one of multiple actions based on it. For instance, at a Station with trains at it the user will see a list of all trains at the station and then be able to select a train to interact with. Separate classes are used for each in-game dialog, for easier maintenance, e.g. DialogEndGame and DialogResourceTrain. Dialogs are usually instantiated within a click event which is attached in a Controller, for example when StationActors are created, a click event is attached which will instantiate a DialogStationMultitrain.

Back End (gameLogic)

Our game entities are defined in the gameLogic package, e.g. Goal, Player, Resource, Train, Station.

GameState
This is an enum which is stored and modified in the Game singleton. The four states are Normal, Animating, Routing and Placing. This allows for the same parts of the game to act differently depending on what state the game is in. For example, TopBarController is responsive to the GameState as it uses this to determine which buttons to show or hide. Also, station dialogs are only launched when the game is in its Normal state, you wouldn't want the dialog appearing when you're trying to place or route a train. There is also an event which fires off when the GameState changes.

Player
Player objects store their current and completed goals and their resources. PlayerManager is used to initalise the Player instances, retrieve the current player, and fires and event when the turn changes (and therefore the current player) and another event when a property of a player changes (e.g. within a turn, a player might drop a resource). The UI uses these events so it can update the scene when something changes. Further use of these events could include a feature which saves the player state into a file each time it changes.

Goal
Currently too much of the Goal implementation is in the base class Goal and GoalManager. Goal should contain shared functionality for any possible type of goal (e.g. boolean flag for if goal is complete). GoalManager shouldn't be dealing with how to generate a particular *type* of goal. In the existing implementation we have one type of Goal of the form "send train X from Y to Z". This should be renamed to something like DestinationGoal, and should inherit from Goal. It should also contain it's own logic for the constraints of this goal type (X, Y and Z). With this structure the next type of Goal to be implemented would also inherit from Goal, and GoalManager would somehow choose how to generate a goal of type A or type B without needing any knowledge about how these goals are formed.

If one wanted to extend goals to provide a score value, they could then change Goal to an abstract class, move the "DestinationGoal" code from the base class, and require that the new child class implement some method to return their score value.

Map
The gameLogic.map package contains all information regarding the generation of the gameMap, but not displaying it. The main Map class contains ArrayLists of all stations and connections. It also parses the JSON file which contains the different stations and connection between them. The inclusion of station and connection data as JSON data files allows it to be configured using different tools, including the HTML editor that is included within the game scripts for developers. This makes adjusting the configuration of the map a lot simpler than having to manually enter all information and therefore saves precious development time.

Station objects contain information about their name, location and if trains are allowed to stop there. CollisionStation inherits from Station and currently adds no extra functionality, but this subclass does allow other parts of the code to check if a train can collide at this particular station/junction (by checking if it's an instance of CollisionStation). The inheritance allows junctions to share the functionality of stations (e.g. trains can be routed through them).

The Connection class represents a connection or "the track" between two Stations. This class could just be a Tuple but we have defined it as such to allow future extension. For example there could be a variant of the game where a connection has a weight limit because there could be a bridge, and certain Trains can't go through certain connections.

Resources

In the gameLogic.resource package, all resources used in the game are defined, including Trains and Power-ups. Train objects contain their location, where it is being routed to, as well any stations it has historically visited. This data is used to animate the train and determine whether the player has completed any goals. Player objects store an array of type Resource, and Train extends Resource.

PowerUp was implemented as a holding class to allow the possibility that trains could be powered up as we discussed in assessment 1. However, due to time constraints it was not implemented and Power-ups are not accessible in game. We left the stub class there as it's likely to be implemented in the future and this clearly shows our design intentions to future developers.

ResourceManager is responsible for reading in the different types of trains which are stored in a JSON file and then randomly generating a new resource each time a new one is needed by a player. Once this random resource is added to a player it can then be placed at a station and routed to a destination. ResourceManager does contain code specific to the Train subclass of Resource, this is not ideal, as with GoalManager, it's better for the Train to define how it is created via methods it specifies.

Game singleton

This was our only singleton class, which gave us the following benefits:

- Very quick to implement
- Ensured only one instance of this class was created
- Gave very easy access to components we decided could be global: PlayerManager, ResourceManager, Map and GameState

The disadvantage of having this class is the coupling that so many classes across the game now have on this singleton. Which means a lot would need to change if this game class ever changed. A better approach would have been dependency injection. We could have specified interfaces for the aforementioned global components. Then we configure a dependency injection library[2] [1] by telling it which interfaces map to which concrete implementations.

Model-View-Controller (MVC)

In the assessment 1 plan, I stated that I would use an MVC architecture to improve code separation. The main success we had with this was our Model code is contained in the gameLogic package and has a nice separation from our controllers.

Here are few examples of where we have achieved this:

- While routing, our RouteController asks the Map class (in gameLogic) whether a connection exists between the current station and where the user just clicked

[2] Apologies for this link being to a PHP DI library, I haven't researched Java DI libraries and I have experience with this library and it demonstrates the concept I'm trying to illustrate

- The (gameLogic) Map class, parses a JSON file containing the Stations and Junctions and stores them in an ArrayList. The Station controller then uses this data to add multiple StationActor instances to the stage
- The end turn button is created in TopBarController, and has an anonymous click listener attached which calls the turnOver() method in gameLogic's PlayerManager class. This publishes two events playerChanged and turnChanged, the listeners of which will go on to give players resources, update the GUI, increment turn number etc.

We were not as successful with separating the Controllers from the View code. In our project, our View objects are Actors. These contain very little logic of how they are display because this has been done in the controllers. For extra separation, our controllers could deal with user input and game events exclusively, and not modify Actors, we could then have some other class which deals with ViewLogic, but this would add some code duplication and more complexity (in terms of another layer) for little benefit. An answer on Stack Overflow [2] generalises our case by saying "a pragmatic balance between an idealized MVC and strict enforcement of DRY", where we have chosen to lean more towards the DRY side.

Events

We have used events in multiple places to reduce coupling. In many GUI applications, something will need to react based on an event happening elsewhere. This is a great opportunity for coupling to occur. Situations where an event happening at A will cause B, C and D to change in some way has been represented by A having a list of listeners/subscribers, but it doesn't need details on who they are, as long as they conform to the relevant listener interface. This has resulted in us having loose coupling [3].

Figure 2 is an example of how simple it is to specify the contract between the publisher and subscriber. There parameters of the event show the data that the listeners will receive when the event is raised.

Figure 2

```
interface SomeListener {
  public void methodListenerMustImplement(Station s);
}
```

Interactions

We have carefully considered how the user will interact with the game, including what they would expect certain buttons to do as well as how the interaction can advance the game faster to keep the players interested in the game. Much about the design decisions is addressed separately within the GUI Design report.

Main Menu

The Main Menu screen provides a good introduction to the game. It also requires very little processing to display meaning that the initial load time of the game is very little. It also provides a very clear route into the main game for fast play. The screen uses big buttons so the players can spend as little time on this screen as possible.

When the "Play Game" button is pressed it does not move immediately to the game window as it takes a couple of seconds to load the game assets. During this time, however, the screen is not redrawn so there is content visible on screen. It would have been ideal to preload this content to allow the game to transition faster to the map window. This behaviour is more complex to operate and would require multiple windows to provide a smooth experience. Therefore this approach was not justifiable.

Top Bar

The Top Bar is used to perform many different functions, but provides a clear central information point if it is not necessary to open a dialog message or the overlaying the information on the map would cause other information to become unusable. It also provides a convenient location for the user to access options, meaning there is a much shallower learning curve as less interaction patterns are needed as supposed to each option having a different system.

This area could be improved upon by having some permanent information such as the current player and the score information, however as scoring is beyond the current scope of the assessment it would not provide any real benefit and would have required several changes elsewhere in the code that time did not permit.

Placing Trains

This interaction is very intuitive by allowing users to drag the train to the station they wish to place it at. This method works well as listeners can be attached to station click events to facilitate the process without any additional interaction from the users. It also allows the user to use the familiar map to place trains. From an engineering perspective, this was simple to implement and by allowing users to place the train at the station means their cursor is ideally located to begin the routing interaction, which is a likely next action.

When trains are at stations it was easy for the station to become cluttered due to the number of trains in the area. This also meant only the most recently placed train could be clicked. This was redesigned to allow the trains to stack at stations, with a number above the station icon indicating the number of trains at it. This could be improved by allowing the number within the station icon, as there is space, however LibGDX does not allow this to happen with strings, drawn directly onto the stage, to be drawn over actors. This was therefore the best solution, with the text being black to stand out against the green and blue of the map. Clicking on a station displays a dialog giving all trains at the station and allowing one to be selected. If only one train was at a station then this screen could be bypassed, but constraints on time meant the implementation was applied to all stations with any number of trains at them.

Players are given two new trains (resources) on each turn. Whilst the idea of using an in-game currency was discussed to allow the player to purchase resources this was rejected as it would constitute a form of scoring, which is beyond the scope of the assessment.

Routing
Routing is a simple interaction which displays the station the train is currently at and the route which is to be taken. It is triggered via the dialog which appear when a train is selected whilst at a station. The user can then select any station connected to that one to continue their journey. The route chosen is highlighted on screen. No changes occur to any stations or junctions and the change in colour of the lines between stations provide a sufficient visual queue, although it is the station that is clicked on to add it to the route.

When the route is complete the “End Routing” button in the top bar provides a way to finalise the route and store it within the game. To increase playability, the routing does not make any suggestions based around goals, it is up to the user to make the choice. This particular interaction method also provides a facility for the player to cancel the routing before it is stored, meaning that any accident on the players part is not necessarily punishing to them.

Changing turns
When completing a turn (all trains have been routed as desired) a player presses the “End Turn” button in the top bar. This then changes the display to being relevant to the next player. This occurs whilst the trains are still animating, although no actors or buttons on-screen can be interacted with by the user. This simplifies the code greatly by firing the EndTurn and PlayerChanged events at the same time. It will also allow the next player a few seconds to consider their moves in their turn whilst the game is updating the screen by animating the train movements.

Goal Completion
The user has no direct interaction with the goals which are automatically assigned to each player and tested automatically by the game. The game tests trains arriving at their final destination to ensure they are of the correct type and have passed through the origination station since the goal was issued. When the goal is completed the use is provided with a visual message and the goal hidden from view. Hidden goals do not count towards the goal cap therefore they can be issued with a new goal on their next turn.

Goals are displayed on-screen as buttons to allow for extension of their function as the game grows to include quantifiable goals and more information about an individual goal will need to be accessed. It would, however, seem out of place with the game in it’s current form to display any dialog regarding a goal as all information about it known by the game is displayed within the textual description of the goal.

Trains used to complete a goal are also removed from the player once a goal has been reached. This decision was made to increase the playability of the game by having an ever changing pool of resources and preventing a player from stockpiling faster trains and gaining an advantage over the other player that way.

Summary
Although there is room for improvement in the code base, we have managed to create a working implementation of the game, as per the specification within the time frame with little

sacrifice to the code quality, much of the code base is extensible and should cope with future changes in requirements. We have also set a good precedent for how future features should be implemented. We get some benefit from using events with this small project, but even more benefit will come when future developers, hopefully, add further listeners, which may not have happened had we not spent the time making these design decisions.

References

[1] Symfony, “PHP Dependency Injection Introduction,” [Online]. Available: http://symfony.com/doc/current/components/dependency_injection/introduction.html. [Accessed 18 January 2015].

[2] tvanfosson, “How much logic is allowed in ASP.NET MVC views?,” StackExchange, 14 January 2009. [Online]. Available: http://stackoverflow.com/a/443213/138251. [Accessed 18 January 2015].

[3] Wikipedia, “Event-driven Architecture,” Wikimedia Foundation, 2 December 2014. [Online]. Available: http://en.wikipedia.org/wiki/Event-driven_architecture#Extreme_loose_coupling_and_well_distributed. [Accessed 18 January 2015].