# *Architectural and Design Report*

## Game Interactions

As detailed in our project specification document, we intended to implement our Trains across Europe game with an entirely mouse-operated user interface. Whilst this remains true for the majority of the game's interactions, we chose to use a keyboard input for panning the map. This interaction is achieved by pressing the up, down, left, and right keys. Initial group conversations about panning the map described an interaction where the player would move their mouse cursor into an invisible border around the edge of the game play space. The camera would then pan in the direction corresponding to the specific edge that the cursor was occupying. We felt, however, that using the arrow keys would offer finer control and be more intuitive.

The only other implemented interactions that make use of a keyboard are text input boxes such as selecting a company name. Keyboard input is the obvious choice for entering text into the game; alternative solutions like an on-screen keyboard were not considered appropriate, as it was decided that this would only serve to complicate the text input process.

When considering the rest of the user interface, it was our goal to make interacting with the game as intuitive and simple as possible. Since a mouse cursor is now the primary method for navigating and interacting with most common operating systems (Windows, Mac OS and Ubuntu, for example), we saw it sensible to mirror this in our game and use the mouse as the primary interaction method. By choosing to implement menu and game button event triggers and resource selection using the left mouse click button, we have eliminated the need to use the middle and right mouse buttons, hence decreasing the number of possible interactions in the game. With a smaller number of interactions, it follows that a player will be able to learn to play the game in a shorter amount of time.

Another reason for the decision to implement the majority of the game's interactions using the mouse cursor and left mouse button was extensibility. Whilst there is no current intention on our part to produce an Android or iOS version of Trains across Europe, we understood that this was an obvious direction to consider for future developments of the game. The increased extensibility comes with the realisation that it would be very practical to map mouse left clicks to touch screen presses, thus enabling support for mobile devices (as well as touch screen controls on a laptop/all-in-one/desktop monitor).

## Game Architecture: Overview

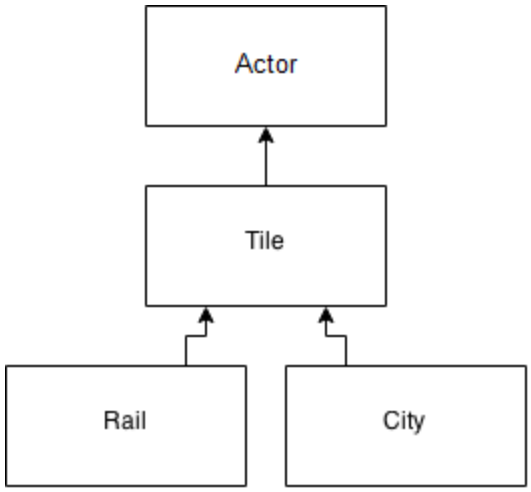
After the completion of our group's project specification documentation, we promptly engaged in conversation regarding the programming language for our implementation. We considered

Actionscript and Java, however quickly decided that Java would be the more appropriate choice as all the members of our group had a solid grounding in the language. In order to prevent an unnecessary amount of time being spent on the game's implementation, we chose to make use of the libGDX[1] game development framework, a common choice for Java developers wanting to make use of OpenGL. As a result of this, some of the classes detailed in this architecture report are subclasses of predefined libGDX classes; these instances will be explicitly stated.

## Game Architecture: Map and Tiles

The inclusion of towers has the requirement of implementing the game map as a grid of tiles that each have an attribute that determines whether or not a player can build a tower on the tile. This implementation also gives extensibility in that it provides the potential for another team to implement a procedurally generated random map on every game start. However, this is not something we intended to implement at this time in the interest of time constraints.

To initialise the structure of a grid tile, we extended libGDX's Actor class, as this class contains the x-coordinate, y-coordinate, width, and height attributes that we required in our Tile definition. An actor is also an interactable object, including methods for hovering over the actor and for clicking into the actor. As a result of having different tile types (blank, city and rail), we were also able to extend the blank tile class, known as Tile, into the City and Rail classes. These relationships are demonstrated in this simple inheritance diagram:




On top of the four inherited attributes from Actor, Tile also has an attribute *neighbours* for storing a tile's four adjacent tiles, this is useful as it lays the groundwork for implementing the attack range of trains and towers. We also use neighbours when generating rail, to select the

[1] Available open-source at: http://libgdx.badlogicgames.com/index.html

correct rail texture to be used for each Rail instance. Tile does not need an explicit unique identifier, as every tile will be distinguishable from its *x* and *y* coordinates.

Implementing cities required extending the Tile class because a City object also has a specific name associated with it (e.g. London, York), a 3 character identifier and a list of trains currently occupying the city, whilst the Rail class has to define a boolean for testing if the Rail is in a city, and have a specific texture depending on the direction of the rail. On top of adding additional functionality to the subclasses, it was also necessary to distinguish between tile types for the sake of rendering the map; a rail should look different to a city should look different to a blank tile. As such, this implementation allowed use of Java's *instanceof* operator for checking which class an object belongs to.

The Map class is a concrete class that handles the generation of a 2D array *MapArray* that stores the contents of the game map. The class' constructor takes an *x* dimension and *y* dimension, then populates *MapArray* with blank Tiles to that size. Rails are hard coded into the Map class where an appropriate texture is applied to all the rails. The ChooseRailTextures function is called to pick the correct rail texture to match the rail tile to its adjacent rails.

## Game Architecture: Buttons

Two classes of button have been implemented thus far: GeneralButton and GameButton. The only difference in attributes of both classes is that GeneralButton has variable dimensions whereas GameButton is fixed to 16 by 16 pixels. The reason for this is that buttons on the menu screen will need to vary in size so that they can be fitted into the menu UIs. We decided in discussion that instances of GameButton should all be the same size for consistency, the graphical design, the in-game tutorial and in-game manual will indicate the outcome of the button click event. Because GameButton is a special case of GeneralButton, GameButton is defined as a subclass of GeneralButton. The isPressed() method handles activation of a button.

## Game Architecture: Train

Trains are objects that can only move on tiles occupied by either a Rail or a City. When given a move command a pixel level animation occurs to move the train sprite along the movement path in order to follow the coordinates the object holds. Trains are given routes by players during the deployment phase, a train in a City can be traversed to an adjacent City once per turn. Currently, only one type of train exists in the game, and the texture for the train is based solely on the player who owns the train. A Train, like the Tiles, is also an Actor as it requires drawing and updating on the game screen. Possible train routes are stored in a list of lists of Tiles. These possible routes determine the Cities that a train can traverse to. A specific route is selected by the train when a player clicks on one of those Cities. This lines up with the use cases detailed in our specification document.

## Game Architecture: Player and Goal

Goal is a class that randomly generates a one of two kind of goal for the player to complete, the class also calculates the rewards associated with completing the goal. We could have had the reward generating function within the Player class but after discussion felt that it made more sense and was more more cohesive to have goals that generate their reward on creation.
The Player class has been designed around keeping track of both players' progress separately. There are two instances of Player object one for each player where we can easily store some integers representing resources - which are updated when resources are earned/lost. Also there is a list of trains owned by the player, although in this version of our game each player will only ever have one train, this has been left in for extensibility.

## Game Architecture: Screen

When using libGDX, application displays are rendered using extensions of the Screen class. A screen has support for stages and cameras, which are implementations of a viewpoint for the player. These viewpoints require appropriate manipulation to render animations on the game's screens. The libGDX Screen class has a render method that is called when the screen needs to render (and update), however we override this method in all of our screens, so that we can render the appropriate GUI elements. Making calls to new screens is achieved using the following code:

```
game.setScreen(new Screen(game));
```

The MainMenuScreen, CustomisationScreen and HowToScreen are all just interactable, non-movable text and images on a static camera. All of the GUI elements on these screens are rendered to the same camera. The first screen displayed is MainMenuScreen. This screen instantiates objects of the GeneralButton type in order to display the main menu options "Play Game", "How to Play", and "Quit". The update method for this screen checks if any of the buttons have been pressed. If one has, the correct screen is called. The CustomisationScreen is implemented using the same methods as the MainMenuScreen, with only the assets and buttons being different. HowToScreen contains only a rendered image showing the game controls, it also listens for a mouse click which returns back to the main menu.

The main game screen is significantly more complicated than the menu screens and makes use of multiple stages and cameras:

- baseStage
- cityStage
- trainStage
- uiCamera

Having multiple viewpoints allows assets to be in and out of focus (determines whether or not it is interactable) and also the order in which assets are rendered. The rendering order controls the z-position of assets. The base stage is rendered first and is where the map and rail is drawn. The cities are rendered above this on the cityStage, and the trains above the cities on the trainStage. All of these stages are dynamically pannable with input using the arrow keys; they move together as to prevent assets across the stages becoming misaligned. The reason for these being stages instead of cameras is because all of the elements on them are actors and as such are interactable. You cannot add actors to a camera, a stage is required for this. The uiCamera, however, has no interactable assets (excluding the nextPhaseButton), only text and images that are displayed on the screen to inform the player of their resources and goals. This camera does not pan with the game map, this ensures the text and images always remain in a static location within the game window where they can always be viewed by the players.

Implementing the setup and deployment phases of a player's turn required algorithms that read a phase number and select an appropriate stage for input. In the setup phase, the baseStage is interactable. In the deployment phase, the cityStage is interactable. In the movement phase, when the trains process their traversals, no stage is interactable, however the nextPhaseButton remains active as it belongs to a uiCamera, not a stage.

## Game Architecture: Tower

A tower is another object that extends Actor. It has an owner to determine the correct texture to be rendered and coordinates for drawing the tower on the correct map tile. Again, as an extension of the Actor class, a Tower is interactable and can be added to a stage in order to be drawn to the game screen. In the current implementation of the Trains across Europe game, towers can be placed on the map, however are otherwise non-functional. There is also a known bug apparent where the towers delete themselves from the map at the end of a phase.