Architecture Report

UML Class Diagram

Due to the size of the UML class diagram, it is hard to read in detail within this pdf document. Therefore it has also been posted on our project website. It can be found directly by using this link.

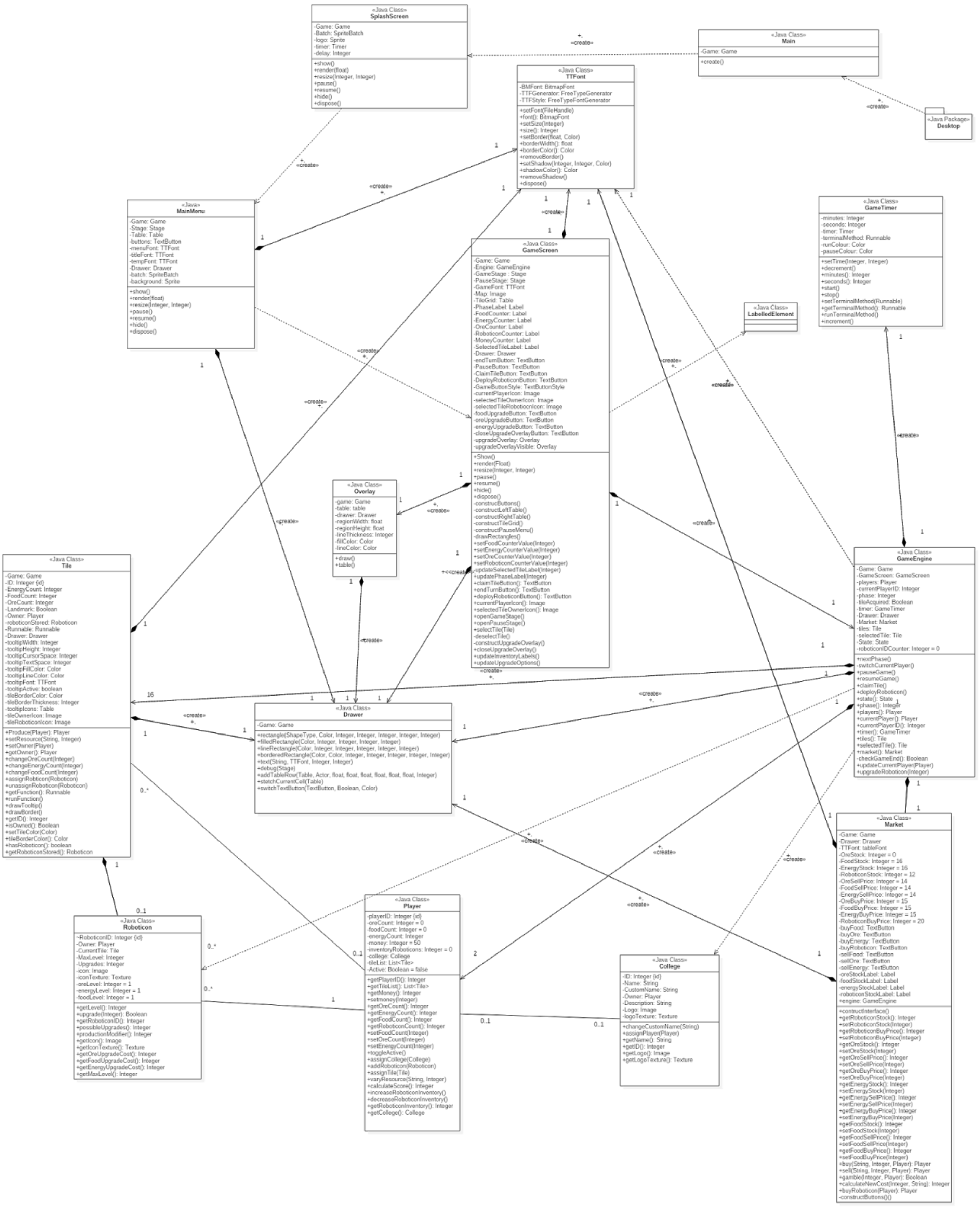
https://jm179796.github.io/SEPR/Images/UMLClassDiagram.png

Languages and Tools Used

In order to create the concrete architecture, we decided to use StarUML. This is because it allowed us to be a lot more specific when describing our architecture. For instance it enabled us to specify the stereotype of each class such as <<Java Class>>. It is free software and therefore could be used by all group members whenever they wanted to edit the diagram. It also provides tools specific to the UML standard that we were using, UML 2. All relationships between classes are association(denoted by no arrow), dependency(denoted by an arrow) or composition(denoted by a black diamond). For certain dependencies there is a <<create>> stereotype which signifies that the dependency is due to an object being created.

The library that we used to develop the game was the libGDX library[1]. Java was the programming language that, as a group, we felt we were the most familiar with. As a result we decided to develop the game using Java and libGDX was the library we thought would give us the most support in doing so. One of the main advantages of this open source library is the documentation that it provides. All of the powerful features that it enables are fully explained on their website. It also has the advantage of being abstract by not forcing a specific design upon the developer but rather to simply provide them with the tools they need to create a variety of programs. All of the rendering of the game was provided by the library, we simply needed to provide it with what to render and when.

UML Class Diagram



Justification for the Architecture

Note that references in bold signify a requirement reference number that can be found in the RefactoredReq1 document here https://github.com/jm179796/SEPR/blob/Assessment2_Docs/Updated%20Assessment%201%20docs/RefactoredReq1.pdf

Classes Implemented From Abstract Architecture

Tile

The only removal that we made from the abstract model is the isAdjacent method. This is because there is no longer the requirement for a player to only be able to acquire tiles adjacent to their own **[9.a.ii]**.

The most substantial change to the Tile Class is that it now extends the button class from the libGDX library. This is necessary as the 'Listener' functionality is required in order to detect when the user clicks on the tile. Each tile needs to be able to distinguish which player owns it based on its appearance**[3.a.i]**. Therefore the setBorderColor function is required to change this using the tileBorderColor attribute. The player's college icon in the UI dictated by the tileOwnerIcon attribute further distinguishes who owns a tile. A tooltip has also helps the player distinguish the tile than the player is hovering their mouse over. This is implemented with the drawTooltip method. Future builds will show more information on the tile such as the resources being generated**[11]**. As the tile implements libGDX functionality, it requires to know the game state which is stored in the Game attribute. All the tiles are created and stored by the gameEngine class in an array of size 16**[2b]**.

Player

All functions and elements specified in the abstract UML diagram for the player class have been carried over to the player class in the concrete architecture. The inventoryRoboticons integer stores how many roboticons have been purchased from the marke. When a roboticon is deployed this value is decreased and a Roboticon object is created. This is because roboticon objects are created when they are deployed, because having a tile in their constructor means they must be assigned to a tile**[8c]**. There are only ever two player objects active at one time **[3]** and both are created and stored by the gameEngine.

Roboticon

The roboticons are not stored in the GameEngine, instead they are held in the roboticonStored attribute of a Tile. Therefore the roboticon must have a corresponding Tile and thus a Player when created . The roboticons also store an image showing their current upgrade level**[1cii]**. The cost of the upgrades for the Roboticons are calculated by the respective Roboticon object based on it's level, it is then returned to the GameEngine.

Market

The Market extends the table class as the Market is presented as a table in the HUD**[16b]**. Because of this, the Market can store libGDX buttons and labels. These are added using the constructInterface and constructButtons methods. It is created and stored within the gameEngine.

College

As effects haven't been implemented yet, there is no effect stored in the class. The only addition is the description attribute that will be displayed at the college selection screen. The GameEngine creates all of the colleges, they are then assigned to players if they are selected by the player**[13a]**.

Classes Not Implemented From Abstract Architecture

Phase

The phase class has been made redundant by the GameEngine and GameTimer class. This is due to the GameEngine storing the current phase and the GameTimer class managing the time..

Effect

Effects weren't part of the minimum requirements for this stage in the project development and due to time constraints they weren't implemented. This also applies to random effects.

Landmark

Not having an effect class also meant that the landmark class couldn't be implemented at this current time, This is due to them requiring bonus effects**[2.a.ii].**

Classes Not Contained Within Abstract Architecture

Main
This class is required by the libGDX library to initialise the game state, it then simply initialises the SplashScreen class.

SplashScreen
A splashscreen is presented to the user as the game opens displaying the team logo. This class, which extends the screen, renders the splash screen before opening the main menu.

MainMenu
The main menu is implemented through this class**[13]** with the extension of the libGDX screen class . It uses the TTFont class to generate the varying fonts that are displayed. Buttons are displayed using the addTableRow function from the Drawer class. It then creates the GameScreen class.

GameScreen
All of the GUI that is presented to the user is created, manipulated and edited here**[2]**. It presents the game map, the tile grid and the HUD for each player **[2,14,16]**. It also allows the user to pause the game with a pause button**[19]**. All of the labels and buttons of the game interface are created by this class using the Label and TextButton objects from the libGDX library respectively. The GameScreen also manipulates the objects on the screen, for instance updateInventoryLabels function updates the labels displaying the player's inventory when their values change. The GameScreen object initializes the GameEngine and calls it to determine what values are to be displayed on the screen.

GameEngine
Because of the lack of depth of the abstract architecture, as well as the uncertainty of the format that the game would take(i.e. would it be event driven are a main loop), there was no class present that would maintain the state of the game. Upon starting to write the program, we quickly realised that this would be essential so we introduced the GameEngine class. As shown by the UML diagram, the concept of the class is to create and store the different objects within the game. These include the two players that are participating, the market, the 16 tiles on the grid and the market. We decided that the program would be event driven, so the GameEngine also handles what happens when different interactions between the user and the game take place, such as a button being clicked on or a tile being moused over. The GameEngine has the responsibility of advancing the phases of the game. Because the game is event-driven, the GameEngine advances phases by actions such as the player pressing the end phase button or the timer ending. This is implemented using the nextPhase function.

GameTimer
Certain phases are timed**[9]**, therefore a game timer is required to record the time left of the specific phase. This class uses the timer object from the libGDX library and adds it to a libGDX label. The timer decrements internally and sends its output to the GameEngine object that it's associated with. It also has a parameter that determines it' colour, this is so it can be hidden on phases that aren't timed**[9]**. This is created and stored by the GameEngine.

Drawer
Several methods must be executed in order for objects to be rendered on the screen, therefore to avoid repeating code a drawer class is needed to create objects that are frequently used such as rectangles. The methods work by taking in parameters such as dimensions and color before rendering the object on the screen using the shapeRenderer from the libGDX library.

TTFont
LibGDX requires fonts to be in bitmap format in order to be displayed within the game. Therefore this class is required in order to convert fonts from .TTF format into bitmap format. It simply takes the .TTF file as well as parameters such as color and size before generating an object that can be used as a font that libGDX objects can display, All classes that display font make use of this class.

LabelledElement
A class that allows labels to be recognised as a libGDX actor, the only class that implements it is the GameScreen class.

Overlay
This class implements the upgrade popup screen used to upgrade a roboticon. It extends the stage class due to it being a popup window rather than an additional screen.

References

[1]"libgdx", *Libgdx.badlogicgames.com*, 2017. [Online]. Available: https://libgdx.badlogicgames.com/. [Accessed: 22- Jan- 2017].