Final Architecture and Traceability Report

References to requirements are given in **[ ]** and can found here
Link to inherited architecture
Link to updated architecture

Languages and Tools Used

In order to present the system architecture of both the design that we inherited and of the final design, we used the UML diagram tool which was part of the IntelliJ IDE. The software allowed us to export the UML diagram as a PNG file. The syntax of the diagram is as follows; orange 'f's represent fields, red 'm''s represent methods and purple 'p''s represent properties(getters/setters). Also, arrows that are blue show inheritance. In addition, a green padlock represents a public attribute/method while a red padlock represents a private attribute.

Changes To Inherited Architecture

Effects

The way that effects **[6]** were implemented within the game was completely changed by our team within assessment 4. Initially, effects were implemented through a class for each corresponding effect. For example, the 'earthquake' effect was applied through the *Earthquake.java* class. These effect classes extended the abstract class *RandomEvent*. While this method of administering effects works, it provides no scope for implementation of future effects due to the lack of inheritance as each time an effect is created a new class has to be written for it. As a result, we decided to remove the 'Earthquake' and 'Malfunction' classes and add four new classes. No other classes were dependent upon the three classes mentioned therefore there were no side effects when they were removed.

The classes that we added were *PlayerEffect*, *PlayerEffectSource*, *PlotEffect* and *PlotEffectSource*. These are the same classes that we used to add effects to the game within assessment 3, we simply ported them over to this project and adapted them so that they would be compatible with the architecture we inherited. The justification for the design of those classes for assessment 3 can be found here [link].

Player effects have the function of altering the resources within the player's inventory whereas plot effects provide modifiers for the resource yields of plots. These effects appear through the use of an overlay from the overlay class and are actually implemented through the use of a runnable. The two source classes were used in order to create the desired effects that were then triggered through the GameEngine class. This meant the addition of several methods to the GameEngine class that would create, apply and manage the effects within the game.

Due to these changes, it now means that whenever an effect is to be added to the game, it can simply be created within the corresponding source class. This is a lot more efficient than creating a new class each time an effect is to be added as the inherited architecture required.

The effects system was also changed to ensure that precisely one effect (of either type) is imposed for each player on each turn. This ensures that the game remains somewhat unpredictable and interesting thereby requiring players to adjust for circumstances out of their control - while also not entirely out of players' hands.

Table Classes

Within the architecture that we inherited all of the the GUI elements, such as buttons and labels, within the main game screen were created and manipulated by the *GameScreen* class. While this class functioned as necessary, it provided little in the way of maintainability due to it being such a large 'superclass'. Therefore, we agreed that it would be best to modularise this class into several different classes. These new classes were *MarketInterfaceTable*, *PhaseInfoTable*, *PlayerInfoTable.*, *TradeOverlay*, *SelectedTileInfoTable* and *UpgradeOverlay.*.

The four classes extend the libGDX 'Table' class and are responsible for creating and maintaining all of their respective buttons and labels. For instance, the *PhaseInfoTable* creates a label describing the current phase and updates it every time the phase changes. It is also now responsible for creating and storing the game timer. As shown by the UML diagram, all of these classes are all created and stored within the *GameScreen* class. Each individual table now implements their own drawing functions; the *GameScreen* class simply invokes them. *TradeOverlay* and *UpgradeOverlay* both extend the *Overlay* class.

The advantage of this is immediately obvious as now for example if I wanted to create a second market table I could inherit the new *MarketInterfaceTable* instead of writing new code within the *GameScreen* class.

Market Class

As justified within the implementation document, we decided to move any of the GUI elements of the market, such as buttons and labels, from the Market class into the *MarketInterfaceTable* and use the Market Class specifically for the logic of the market. We also moved any of the logic that involved adding functions to the buttons into the *GameEngine* class. Because of this, the Market no longer required to make use of *GameEngine* and *Game* objects and therefore no longer required either of them in its constructor. As before, the *Market* is stored and maintained within the *GameEngine* class.

College Selection Screen Removal

We decided to remove the college selection screen from the architecture of the program. This is because it was redundant due to the fact that players are randomly assigned colleges rather than selecting them. No other classes were dependent upon this class hence there were no side effects caused by this.

Enums

One subtle change we made to the architecture was the introduction of enums for the resources **[5]**. This simply involved changing any references to resources within the architecture to reference the specific enum of the resource instead. This improved the structure by preventing type errors within the code.

The remainder of the architecture was left untouched as we felt that it did not require any further change in order to meet our requirements.

Implementation of New Requirements

Chancellor Implementation

In order to accommodate the implementation of the capture the chancellor minigame **[13]** a new class had to be created to represent the chancellor entity. The class in question, *Chancellor*, stores values such as its location, texture and also the reward that the player gains if it's captured. It also contains the logic required to move the chancellor around the map. To allow the chancellor mechanic of the game to function, the *GameEngine* will invoke the *Chancellor.Activate* method within the *Chancellor* class at a specified time. When this has commenced, the *GameScreen* will draw the chancellor on the screen through the use of the *drawer class.*

Supporting 4 Players

As the architecture we inherited already supported 9 players, it wasn't necessary to change the structure in order to have multiple players. We simply reduced the number of players by changing an integer within the *PlayerSelectionScreen.*

Justification For Final Architecture

Overall, we feel that the final architecture of the system hasn't drifted too far away from the basic abstract architecture that we designed within assessment 1 [here]. Our philosophy at the beginning of the project was that every entity represented within the game has its own class and this has been maintained throughout the process despite the changing requirements.

One aspect of the architecture that we feel we could change in the future is trying to further modularise the two classes *GameEngine* and *GameScreen.* As shown by the diagram, these two classes are by far the largest within the game and moving some of the code that they contain into separate classes would improve maintainability and give greater scope for improvement. We tried to begin doing this by creating the table classes as stated above by moving certain parts of GUI code from the *GameScreen class* into separate classes. However, given more time would have liked to further move some of the code from out of the *GameScreen* class so that everything graphically related within the main window of the game had its own individual class to create and maintain the objects it displays.

The *GameEngine* class is another class that we would have liked to, given the time, refactor further. While we feel that there isn't any code that is particularly redundant, there are instances where code could have been moved into a different class in order to shorten the 1300 lines of code currently within the class. For example, the *setMarketButtonFunctions* function could have been moved into the *Market* class. The downside of this however would be that we wouldn't be able to initiate unit tests for the market. This is because changing the button functions requires use of the *GameScreen* class, which is currently untestable due to the drawer overheads.