Final Architecture and Traceability Report

A full UML class diagram has been created to document fully how each class in our project relates to one another. [1] This covers not only the dependencies of classes but also the methods and variables that are stored within each class. As far as the overall architecture is concerned, for the most part it has been left untouched. We as a group nearly fully agreed with the architecture that we inherited from the previous project group. It was all well set up with only the necessary data being shared.

All of the classes representing the different screens for the game were well thought through and constructed. These classes have methods that traverse the different available screens to the user from the current screen. These screens also contained all of the data that pertains to the specific screen with data that is needed from other screens available through accessors.

The functional classes of this game (the classes that don't have GUI elements) all interact well with the screens so that the correct data is being displayed to the user at all time. A key set of classes to note are the phase classes. These classes all uniquely define the stages of a turn within the game. Giving all of the necessary functions of each phase, an example of this would be the reinforcement phase which correctly gives the units the player has collected over their previous turn to allocate in this new turn.

It was a great base for us to develop the requirement changes on. Changes to the save and load method were the only part of the architecture that was heavily edited from the previous group. This is purely because too much data was being saved compared to what was necessary to restore the game after it had been saved.

In response to this we adapted the code to save what we thought was necessary for the purposes of resetting the game to the state when saved. Although the game could be saved and loaded, it was only one save game. Saved data also didn't have the turn timer or the PVC tile stored. This has been changed. An available four slots for saving the game have been implemented. We have done this because possibly more people would be using the same computer so to allow for more people to play on one device we've given four slots to save in.

The process of the data being saved has also been changed to account to the unnecessary data that was removed. All of the necessary data including the new elements introduced in this assessment are all saved within each game state. The save file is a collection of game states to make it easy to process the data to display data to the user to identify the game they're looking for. To make it more readable, the order and formatting of the code has been changed and commented so it is easy to follow.

There are three classes pertaining to saving and loading. These are the *SaveLoadManger*, the *GameState* and the *JSONifier*. Each three have been heavily edited due the to changes to the entire processes. To start with *GameState* is used to store all of the information about the game as a whole that is required to restore the game to a playable state. This was a much bigger class when we inherited it. It has been reduced to hold only necessary information about the games state for example, whose turn it is and what phase of the game they are in.

Next the *SaveLoadManager* has been changed. This used to have a lot of methods that gathered the data from all of the games classes to be put into the *GameState* class as well as returning all of the data when it is loaded back in. A lot of data conversion to be the correct type for saving and loading was required for this process. Due to this large overhead, code that retrieved and saved unnecessary data has been stripped from the class. Although there is now a comparatively small amount of data being saved, there is still a large

overhead for saving and loading. The format for saving requires a conversion of Java data types to a JSON string and vice versa when loading the game.

JSONifier takes care of changing all of the data into a JSON object and converting this back into Java data types. Again, this class has been heavily edited. This is due to the reduction in data being converted to and from a JSON compatible type. For all of the new elements we added to this architecture and all of the edits we've made, it made the most sense to change most of the types to conform to the all of the changes we've made throughout the game. This means that a lot of data has become arrays/lists or some data have gone from primitives to objects.

Punishment cards are one of the new features to be added to the project. These are implemented in their own package. Each card is a subclass of the main class Card. This class contains the information on what type the card is; whether a player is required to use it on; and if it affects the neutral player. Each card then has its own effect which is stored within each class. This then affects the target player. The punishment cards are kept modular because there is no reason for them to be within the main package as they don't directly influence any GUI elements, the display of the effects are taken care of by the *GameScreen* primarily. The effects themselves take place within the card classes. It does this by passing the player affected and the game screen to edit the necessary data that the card affects. An example would be *FreshersFlu* which cuts the strength of the units in half. It does this by taking the current units in all the sectors, cutting that amount in half and replacing it but keeping the old amount of units in a separate variable stored in the player to replace when the turn is over minus the amount that was defeated times 2.

Units of the original architecture have been split into two different types of units: Undergraduates and Postgraduates. These units have been, again, made in their own package and then reintegrated to the *Player* class and the *Sector* class as is necessary. The Player can have a combination of any amount of undergraduates and only one postgraduate in any one sector. Postgraduates are a lot stronger than undergraduates and this is the reason for there only being one allowed in any one sector. They can also only attack once, again this is due to their considerable strength over undergraduates.

To facilitate this, the *GangMember* class was added, with classes *Undergraduates* and *Postgraduates* extending this. GangMember contains a name attribute for the unit, and this is set appropriately in the subclasses *Undergraduates* and *Postgraduates*. Further to this, the *Postgraduates* class contains an attribute that indicates whether the unit has attacked this turn or not. We felt this provided a good level of extensibility to the program, allowing additional new types of gang member to be easily added to the game by extending the *GangMember* template. This also allows information to be kept based on the type of GangMember object (such as the *attacked* attribute of *Postgraduates*).

Due to this change in the implementation of units, the *unitsInSector* attribute of *Sector* has been changed from an integer to an ArrayList of *GangMember* objects. This allows us to determine the type of each unit in a sector by looking at its name attribute, and for postgraduates whether they have attacked this turn by calling the method *getAttacked()*.

References

[1] Sid Meiers, "UML Diagram," [Online]. Available: http://sidmeiers.me/documents/UML4.png [Accessed 01st May 2018].