Implementation Report

Incorporating the requirement changes

At the start of this assessment, we were asked to implement two major changes. The first being adding another type of gang member, in this case postgraduates and the second to add punishment cards to the game. These are covered in the ‘Final Requirements’ document [1], and associated requirements will be referred to by their codes (e.g. *F19*).

In order to facilitate the addition of postgraduate gang members to the game (relating to requirement *F19*), changes were made to the way in which sectors store the number of units. Instead of just an integer value to indicate the number of units, sectors now store an ArrayList of GangMember objects, which may be of type Undergraduate or Postgraduate. Methods for accessing and setting the number of units in a sector were modified and added as appropriate to support this change, with there now being a respective method for both Undergraduates and Postgraduates (fulfilling requirement *F20*). On top of this, the method that is used to calculate the outcome of attacks has also changed to accommodate the new ArrayList, with an additional method being added to handle the postgraduate attack. Postgraduates attack by dealing a random amount of damage between one and five (inclusive) to the targeted sector. They are also limited to one per sector, and are destroyed when all the undergraduates in the sector are lost. The difference between undergraduates and postgraduates was significant enough such that requirement *F19* was fully satisfied.

We felt that creating a general GangMember object gave the most potential for extensibility, whilst adding minimal complexity to the architecture. This allowed both Undergraduates and Postgraduates objects to inherit the same features but allow the difference for postgraduates to only attack once, implemented through the *castUsed* attribute.

Postgraduates are allocated at a base rate of one per turn, but players may obtain more by capturing specific sectors that give the player postgraduates to allocate. When a player is allocating units, they select a sector in the same way, but now may select the number of each type of unit to allocate to that sector. This will then be reflected in the modified indicator for the number of units in the sector, which shows the number of undergraduates, followed by a slash, then the number of postgraduates. We felt that this changed indicated to the player in a clear manner the number of each type of unit in each sector, whilst making minimal changes to the existing code.

Overall the UI and processes of allocating troops remain very much in line with the original product, yet provide enough additional functionality to ensure that postgraduates are able to be fully utilised by each player. This ensures that our implementation is consistent with requirement *F20*, but also faithful to the original implementation we inherited.

The second major addition to be made was punishment cards that a player can earn on attacking sectors and then use on other players (in order to fulfill requirement *F21*) . Once we’d decided on the different types of card we were going to implement we started creating the graphics for the cards and adding the necessary new classes.

The main new class is Card, which is a superclass storing the type of the card as an enum (declared separately) along with two boolean flags. First ‘playerRequired’ denotes whether the card needs to be applied to a specific player and ‘affectsNeutral’ denotes whether it affects the neutral player or not. Two ‘act’ methods are declared ready to be overridden by the subclasses, one which takes the Player that the card is being used on as a parameter and one which does not for when ‘playerRequired’ is false. Also implemented are the relevant accessors, but no mutators are provided as the card type and the boolean flags should not change once initialised.

There is then a new subclass of Card for each type of card implemented, meaning that if desired more cards can easily be added in future simply by adding additional subclasses. These subclasses override the relevant 'act' method of Card with code specific to the effect of each card. These methods return 'true' if the card has been used successfully or, for certain cards, return 'false' after displaying a DialogBox to explain to the user that they can't currently use the card due to the effect already being active on the chosen player. This return then prevents the card being removed from the player's card hand.

At the start of the game an ArrayList of Cards 'cardDeck' is initialised in GameScreen and cards from there are then removed from 'cardDeck' and added to each player's 'cardHand' as they win cards (thus fulfilling requirement *F21*). When a player uses a card it is likewise removed from the 'cardHand' and added back to 'cardDeck'. Various UI elements are set up and swapped around in GameScreen in order to display the cards in various states along with the menu to choose the player to use a card on. All of these contribute to giving the player the clear impression that they 'play' a card, which helps to satisfy requirement *F22*.

Other changes to the final product

After reviewing the save and load feature we inherited, we felt that the quality of the code was substandard. We have therefore made significant modifications to the save and load classes, stopped excessive data being saved, save data regard the new features and added a save and load screen. Save and load classes are in their own package because they are completely separate from the game and only gather the necessary data to save the game state out to file. The SaveLoadManager gathers all of the data and returns the data when it has been loaded. It also has methods to save and load data out to file using the JSONifier to transfer the datatypes of the gamestate into a JSON string. It transfers back again into the correct data types during loading. The SaveLoadManager has a variable that stores a possible four different game states to be loaded from or to save to file. Saving the game state out to file required the intermediary class Gamestate.

To make the save and loading process more user friendly, a save/load screen has been implemented where the user can see a possible four different game states. These different states can be overwritten if necessary. They can also be loaded from when a game is already in progress, but this overwrites the current game in progress.

References

[1] Sid Meiers, "Final Requirements ," [Online]. Available: http://sidmeiers.me/documents/FinalRequirements.pdf [Accessed 01st May 2018].