Implementation Report:

For this assessment, all major features were implemented as described in the following sections. All code and the related Unity project files can be found on our Github

When implementing our changes, we took into account how the game's architecture was modelled by building upon their individual elements. The following elements were added by either extending or fulfiling their previous architecture:

- *GameObject Structure:* The previous game was implemented by using solely "UI elements" (objects rendered onto a camera rather than existing in the game world). We decided to extend this architecture trend by making any objects we created also "UI elements".
- *EventManager:* The previous architecture focused on having many of their methods (scripts) attached to one controlling game element. As such, many of the methods we have implemented below are also attached to this "EventManager".
- *SaveHandler:* The SaveHandler architecture has been successfully implemented in the "SaveHandler" script and is called through the controlling "EventManager" element, thus adhering to the previous architecture.
- *BonusHandler:* The BonusHandler architecture has been implemented through the mechanic of Chance Cards, giving buffs and debuffs to the players in the game. The "ChanceCards" script handles the bonuses that they offer, while the Minigame offers the opportunity to gain more chance cards.
- *Minigame:* The Minigame architecture was successfully implemented, while the "PVC" script handles the game mechanics, the game is called and run through the EventManager as required in the previous architecture.

The following are features that have been implemented and their justification as to what requirements they fulfil. All features were implemented in accordance with the inherited requirements. Some minor, low-priority features mentioned in the requirements were not fully implemented (as described below), but the game is fully functional without them:

Demo Mode (5)

As per requirement 5, we have implemented a demonstration mode which will reset the game if a period of extended inactivity is detected. According to sub-requirement 5.1 this reset should occur after 3 minutes; we have followed this and if no meaningful action is taken for 3 minutes the game returns to the main menu. Sub-requirement 5.2 does not clearly define what point games are reset to after a period of inactivity, so we have decided that a reset will take the game back to the main menu.

Main Menu (6)

In order to create an interface for implementing requirements 5 and 6, enabling the "Demo Mode" and allowing users to Save and Load games respectively, we found it was important for our game to have a Main Menu Scene. Within the scene, we created interfaces for toggling "Demo Mode" and also a menu for loading games, with a custom grid interface to view and select any valid save files.

<u>Save and Load Function</u> (6)

As per requirement 6, features were added to the in-game menu that allowed game states to be saved and loaded. Users can save the current game state, as specified by sub-requirement 6.1, by using the in-game settings menu. This settings menu will also pause the in-game turn timer as specified by the same sub-requirement 6.1. Saved game states can then be restored from the main menu. As per sub-requirement 6.2, the game supports storing multiple save games at once. Sub-requirement 6.3 mentions storing general information about the saved games, such as time elapsed. While the information provided as an example in the sub-requirement is not stored, the saved games do store the number of players in the game and display a small thumbnail image showing the saved game state. Sub-requirements 6.4 and 6.5 specify an auto-save system and a quick-save hotkey. Neither of these features were implemented, due to their complex nature and the time constraints on the assessment.

<u>Neutral AI</u> (8)

Our implementation of a 'Neutral AI' as specified by requirement 8, is a non-playable entity which owns all of the remaining sectors which are not occupied by players (as per 8.2 and 8.3). After all human players have made their moves the AI takes it turn. When deciding a move it takes a number of factors into account such as number of enemy sections on a owned section's border, magnitude of enemy forces on the border, whether it could move units to mitigate a threat, whether such a move would put another section at risk. There exist two types of move; 'reinforce' and 'balance'. Reinforce moves will move units from friendly sections which have a surplus of units to reinforce a border which it deems at risk. Balance moves attempt to spread the units evenly across all sections. The AI may also choose to skip a turn if it doesn't see a valid move which would benefit itself. The AI will never attack a player as per sub-requirement 8.1.

The Neutral AI uses a non-primitive data structure called '*OrderedList'* to allow for the selection of important sections. It provides public methods which, return the most vulnerable section to enemy attack, and return the best candidate for a balance move.

<u>3-Player Mode</u> (8)

Requirement 8 states that there should be at "2 or more" human players for a game to exist. While no maximum number of players was specified, we still wanted to meet this requirement and include a game mode that contained more than 2 players. To this end we added a "3-player mode" where there are 3 players and 1 AI player.

<u>Chance Cards</u> (9)

Requirement 9 specified that the game must include a bonus mechanism, to be implemented using chance cards. The previous team had also intended to implement a bonus afforded by the possession of landmarks, but after a conversation with our customer we concluded that the requirement for a bonus mechanism was already entirely met by the chance card system, and as such we have not implemented any landmark buffs.

When we inherited the project, there was no implementation of the chance card system other than the GUI object that showed a card and displayed the text that would indicate the number of cards in the current player's possession. Now, each time a player captures a sector they are awarded an additional chance card (as per sub-requirement 9.3), or two if they successfully capture the PVC. The number of cards held by each player is simply stored as an integer; the effect of each card is not determined until the card is used, at which point one of three functions is randomly selected (satisfying sub-requirements 9.1 and 9.2). One adds gang members to friendly sectors, one takes gang members away from enemy sectors and one adds to all sectors. The original requirement stated that all chance cards would either give good effects or be tradeable for more gang members. This has been relaxed in our implementation: instead, we gave each chance card a 10% chance of having a reversed effect in order to increase the unpredictability, and hence replayability, of the game.

<u>PVC</u> (11)

Requirement 11 outlined the need for a PVC (Pro Vice Chancellor) to be implemented into the game with an accompanying minigame and bonus. The functional requirements surrounding the PVC have all been implemented. The PVC 'spawns' at the start of the game when all players sectors are allocated on the map. A sector is chosen at random (following sub-requirement 11.3) and a boolean for the PVC is flipped to true for the sector. Only one sector ever has this boolean set to true, thus adhering to sub-requirement to 11.2. Each time a sector is taken by a player, the game checks if the sector taken has the PVC. If the sector does, the mini game is triggered for the player who took the sector (as per sub-requirement 11.4). At the end of the mini-game, the boolean for the PVC is flipped to false and the random allocation happens again, picking a new sector for the PVC to be in, meaning there is always a PVC on the map, implementing sub-requirement 11.1.

<u>Minigame</u> (11)

As specified in sub-requirement 11.4, taking the sector that contains the PVC triggers a minigame. The object of the minigame is to click on a box representing the PVC as it bounces around the screen. Successfully clicking on the box in one try awards the player with a bonus (as specified in sub-requirement 11.5) in the form of an extra chance card. Following the lack of strict requirements specifying the minigame, a simple click game was created and implemented as a UI layer, following the previous method

<u>Sudden Death Mode</u> (1)

The sub-requirements of requirement 1 specified various failsafes to ensure that games did not end too quickly or drag on for too long. Sub-requirement 1.1 asserts that the players must not be able to lose until 10 minutes into the game; this feature was cut due to time constraints and an inability to find a way to implement it that would compromise some user's experience with the game. Likewise, sub-requirement 1.2 imposes a maximum time limit of 45 minutes to the game. This was not implemented either, mainly because it was determined through playtesting that a game was very unlikely to last long enough to necessitate such a time limit. The "sudden death mode" mentioned in sub-requirement 1.3 was implemented, though it is activated based on the number of turns that have been played, rather than the amount of time elapsed. In sudden death mode, all players begin to lose units in sectors along their borders, speeding up the game. A time limit of 30 seconds was also imposed on each turn, as specified in sub-requirement 1.4.

<u>Changes Made to Inherited Code</u>

To facilitate modification and extension of the inherited code, one of the first tasks was to refactor the code to better reflect the conventions for variable names and general code structure that the team was already familiar with. In every script and method where one of the variables/parameters was refactored in some way, there is a commented section in the header listing all refactored variables/parameters along with how they previously appeared in the code. The header also lists any variables whose types were changed, any variables that were found to be obsolete and removed, and any variables that were added to support new functionality. In terms of code structure, most of the changes made were the insertion of whitespace and the reordering of variable declarations to make the code more reader-friendly.

While none of the refactoring altered the functionality of the code, some changes that did affect the operation of the code were made. The scope of these changes were limited so that the core functionality of the code was not affected too much. They are listed below by script:

In the script AssignUnits.cs:

- The neutral AI was originally allocated a random number of units between 1 and 11 in each sector. The random range was changed to be between 1 and 25 to slow down the pace of the game and provide more balance against human players.
- The methods P1TurnUnits() and P2TurnUnits() had nearly identical bodies; this similarity was abstracted into the AllocateNewUnits(int player) method. The original methods were replaced by the methods AllocatePlayer1NewUnits() and AllocatePlayer2NewUnits(), respectively.
- The SpawnPVC() method was added to support the new PVC feature.

In the script ConflictResolution.cs:

- In the ResolveConflict() method, the variable conflictOccurs : bool was added to keep track of whether or not a conflict actually occurred by moving units, allowing players to move units between their own sectors without the turn passing to the next player.

In the script Section.cs:

- The variable PVCHere : bool was added to support the new PVC feature.
- The variable numberOfNewUnitsPerAdjacentSector : int was added to abstract the number of units awarded at the end of the turn to a variable instead of a hard-coded value.
- In the OnMouseDown() method, a guard was added to the code to ensure that only sectors owned by the current player in the turn order could be selected. Some code was also added to reset the timeout timer if the game is in demo mode.
- The spawnPVC() method was added to support the new PVC feature.