Implementation Report

Explanation of how the code implements the architecture and requirements

Overall, when developing the system the team worked closely with the architecture and requirements to ensure that the code was implemented as planned and that high levels of traceability were maintained. However, the previous team to work on this project only provided a concrete architecture that covered the features implemented for assessment 2 and did not include a concrete architecture for the new features.

To begin with, the Game class created represents the Game object in the UML class diagrams. This class is used to generate a new game, initialize a Map and create four Players, as shown by the compositions in both the concrete and abstract architectures. The Game class helps to meet requirements F6 and F7, as it detects when to spawn new units for players and encapsulates information about the game, such as the current player, which is used during saving and loading.

Another section of the code is the menu system and the Menu class that controls its functionality. This, although absent in the architecture, is vital for meeting a number of requirements. For example, the menu contains a slider which allows users to state how many human players should be used in the game (requirement N3). The menu also allows the system to meet requirement F7 because it provides buttons for the user to save and load games, as well as providing pause functionality when the escape key is pressed.

Within the game there is a map that is used to meet requirement N4 as it displays a representation of the University of York campus, which contains 3D models of the key buildings, roads and geographic features (the lakes). The addition of the map is also used to implement requirement N5, which states that the game map must be divided into sectors. This is achieved because in the map, the area it covers is divided into a number of subsections, named sectors, and each sector is represented by a Sector class. These features are consistent with the architecture because a Map contains multiple Sectors, and if the Map is removed, so are the sectors.

Additionally, Sectors can contain Landmarks and can be owned by Players and Units, as shown in the architecture. For all of these, a Sector can only have at most one owner, unit or landmark at any time during the game. The inclusion of landmarks enables requirements N6 and N7 to be met because instances of the Sector class can contain a reference to a Landmark class, which is used to provide Players with resource bonuses when captured. These resources are represented by an Enumerator ResourceType that can be either equal to Beer or Knowledge.

Requirements N3 and F1, which describe the creation of players within the game, are fulfilled by the addition of the Player class, NonHumanPlayer subclass and the CreatePlayers method within Game. When the game is initialized, the CreatePlayers(numberOfPlayers) method is called which creates 2-4 human players (depending on the value of numberOfPlayers) and NonHumanPlayers are created to bring the number of players upto four. Also, the Player class created represents the players within the game and is used to record the sectors and units owned by the player, as well as its colour, beer and knowledge values. The class NonHumanPlayer was created which inherits from Player, as shown in the abstract architecture, and contains two methods MakeMove and FindBestMove that are used to simulate player turns.

Additionally, the Player class contains the method Capture that changes the ownership of a sector and updates the beer/knowledge values if the sector contains a landmark. The inclusion of this feature allows Players to capture sectors, which is described by requirement F5. Also, requirement F6 states that Players must receive new units, and is met by adding a SpawnUnits method within Player that is called by Game at the start of a new game and after every turn.

Within the code a Unit class has been included. This class is present in both the abstract and concrete architectures, and is used to represent the gang member units controlled by each Player. A Unit can only be owned by a single Player, however Players can own any number of Units, as shown by the 1 to 0..* composition in the concrete architecture. Also, Units are placed within a Sector and Sectors can either contain a unit or not, which follows the architecture by having a 1 to 0..1 relation.

The addition of the Unit class helps satisfy requirements N8, N9 and F6. Requirement N8 is met because the Unit class controls the position of a unit GameObject which is shown on the map during the game, providing a clear representation of the unit's position. Also, requirement F6 is met because new instances of Unit are created and initialized in landmark sectors after each turn. Furthermore, the Unit class plays a large role in the fulfilment of requirement N9, which describes the conflict resolution system. This is because within the Sector class the method Conflict is called which takes two Units as parameters. Also, the attack and defence scores for the Units is based on a random function where the upper bound is equal to the unit's level value plus the beer modifier for attack, and knowledge modifier for defence.

The saving and loading of games, as described by requirement F7, is another key part of the system. The code implements such functionality with the use of the SavingNLoading class, which contains classes SavedGame, SavedPlayer and SavedUnit. The class copies data from the current game, players and units into the 'Saved' classes, which are then serialized and written to a save file. The loading is done by reading from the save file and converting the data into an instance of SavedGame, which is then used to load in an active game.

Within the code, a PVC unit is represented as a boolean value 'PVC' in the Sector class. This differs from the architecture, which shows it as its own class, however this implementation still maintains the same relations with the other classes, for example, a Sector can either have 1 or 0 PVCs, which can be represented as either PVC = true, or PVC = false. Additionally, the current implementation meets requirement F2 because the PVC spawns in a random Sector after 10 turns. The code does this by keeping a count of the number of turns, and after each turn the Game class checks to see if the numberOfTurns has reached 10. When this occurs the PVC attribute within a random, unowned Sector is changed to True.

Additionally, requirement F3 is met with the addition of a dropper minigame, where the player controls a person and catches beer, books and bins which fall from the sky. This dropper game is implemented using two classes, Dropper and MovementLR. Dropper attaches to the items that fall down, and it controls the rotation of the item, as well as detecting if the item gets caught or missed by the player. MovementLR is the main interface for the minigame, which contains methods that start and end the game, spawn the items and assign bonuses. Also, the class implements the movement of the catcher using the left/right arrow keys.

Furthermore, requirement F4 is satisfied because within the minigame, if the player catches two books or two beers, their book/beer modifier will increase by one, which increases the player's attack or defence strength.

Finally, the code meets requirements N1 and N2 because when implementing the game the team made sure that the variable names used were descriptive and that comments were added to the code which help describe how the code works.

Description of significant new features

At the start of assessment 3, we identified a list of features that needed to be implemented. Among them, there were 3 features that, in the team's opinion, were of great importance. These include the non-human player (AI), saving and loading functionality, and the minigame.

The Non-human Player

When the new game button is clicked, the first thing that the user sees is a slider for choosing how many players there are going to be in the game. As there always have to be 4 players in a game, if the number of players chosen is less than 4, the game creates non-human players to compensate that number.

Simply speaking, non-human players have the same capability to make moves and capture sectors since they are a subclass, which is implemented in *NonHumanPlayer.cs*, of the Player class. When it is time for a non-human player to make a move, it checks all the units it owns (*NonHumanPlayer.makeMove)* and finds the one which gives the best result (*NonHumanPlayer.findBestMove*).

Saving & Loading

The functionality of saving and loading can be seen in the menu UI before a game and while in game by pressing the escape key on the keyboard.

Saving works by copying all of the necessary data, including certain fields of the Game, Player, Sector and Unit objects to a custom serializable data structure, which can be found in *SavingNLoading.cs*. Although a dictionary could have been used, dictionaries in C# cannot have values of different types (**note**, there are a few solutions to overcome it), hence the custom data structure chosen.

On the other hand, loading does it in reverse. First, it loads the serialized data structure from the path stated in Application.persistentDataPath and deserializes it. After that, it works very much the same as *Game.Initialize()*. The only difference is that it sets the fields of the Game, Player, Sector and Unit objects to the values of the saved game.

It is worth mentioning that after the game is saved, the file that holds the game's data is not deleted until another game is saved. Furthermore, there can only be one saved game at a time.

The Minigame

The minigame is an essential part of the game, because the PVC is spawned at some point in a game and the player who manages to find it in one of the sectors on the map gets a chance to receive beer and knowledge, ultimately giving the player a slight advantage over others. However, the amount of beer and knowledge the player gets depends on how well they play the minigame.

The minigame is initialized just after the game begins (*MovementLR.Start()*) and stays hidden until a player finds the sector the PVC is in. In which case, the camera that corresponds to the minigame is shown and the minigame objects (*beer*, *knowledge* and *bins*) are randomly dropped (*MovementLR.startDropperGame()*). After a total of 20 objects have been dropped, the amount of beer and knowledge the player managed to catch is given to the player and the dropper game is stopped (*MovementLR.stopDropperGame()*).

Report of significant changes

<u>Different Initialization of a Game</u>

Changes:

- The initializer has been removed from the GameManager, making the Initializer class obsolete.
- The *Game.Initialize()* call in the Initializer class has been moved to the Menu class.

Justification

Intuitively, after pressing the start button, a new game should be initializer. Unfortunately, the button couldn't be implemented, because the game was already initialized in another part of the game, i.e. in *Initializer.Start()*.

Requirements No requirements exist for this change.

Architecture

There have not been any significant changes in the structure of the architecture. However, the previous workflow of a game started from the Game class, whereas now it starts from the Menu class, which initializes the Game class in *Menu.StartButtonClicked()*.

<u>A Menu UI</u>

Changes:

- A menu UI game object, which includes a number of children, has been added to the project hierarchy.
- A Menu class has been implemented to accommodate the functionality of various buttons, i.e. *new game, start, continue, save* and *load.*

Justification

The UI is essential to the functionality of game initialization (choosing the number of players), pausing, saving and loading.

Requirements N3 and F7

Architecture

It has the same effect on the architecture as described in the change above.

<u>Saving & Loading Functionality</u>

Changes:

- The saving mechanism records only the necessary values of the fields in the Game, Sector, Player and Unit objects. The data is then converted to JSON and saved locally on the computer.
- The loading mechanism loads the content of the saved file, converts it back into a C# object and loads the values into the current Game class object.

Justification

Although the saving mechanism could have been implemented in two ways, either by serializing the whole Game object or by creating our own data structure, due to certain limitations, the latter has been chosen. In addition, this choice has not had any considerable impact on the game's performance nor the development speed. This is because both of these methods require implementing an object copying mechanism as there is no simple solution for changing one object with another, which is the case in the loading functionality, where we want to change the current Game class object with the saved one.

Requirements F7

Architecture

As both saving and loading mechanisms are used in the Menu class, the effect on the architecture described in the previous two changes holds in this change as well.

<u>The Minigame</u>

Changes

- A minigame has been added. It starts when a player moves into the sector that has the PVC within it.

Justification

This feature was included to add a bit of skill to the PVC bonus. Along with deciding how much of a bonus you get from the Game, it also increase the enjoyability of the game adding variety. A dropper game was chosen as the aim is intuitive, it doesn't take very long to complete, and it fits with the overall theme.

Requirements F2 F3 F4

Architecture

The mini game is linked to the unit class and the sector class. The unit class is what triggers the dropper game when the unit moves into sector with the PVC. the MovementLR script inherits data from the unit class and in turn the Dropper Game inherits data from the MovementLR.

<u>Spawning of Players</u>

Changes

- When the game is initialized, Players are now created dynamically rather than being manually created within the Unity inspector window.

- Within the Game class, the CreatePlayers method creates four GameObjects, each containing either a Player class or NonHumanPlayer class as a component.

Justification
The previous implementation of the player spawning system was very limiting and this meant that all the players were of type ‘Player’. However, when the NonHumanPlayer type was implemented this had to change because now players are a mix of ‘Player’ and ‘NonHumanPlayer’ types, and the number of each is not fixed. This new player generation system improves the flexibility of the code and therefore may be advantageous in the next assessment.
Requirements N3, F1
Architecture
The changes made to the spawning of players remains consistent with the architecture because once the game is initialized there is no difference from the old implementation. However, this system more closely meets the architecture as it allows for NonHumanPlayer to inherit from Player and allows the number of NonHumanPlayers to be varied between games.

<u>NonHumanPlayer</u>
Changes
- A NonHumanPlayer class has been added to the game.
- This class inherits from Player and is used to control computer players in the game. The class contains two methods; makeMove and findBestMove, which are used to decide the best move to make, then simulate it.

Justification
This class has been added to implement non human, AI players. The class inherits from Player because it requires most of the attributes and methods of Player, and other components in the game see NonHumanPlayers as a Player.
Requirements F1
Architecture
The addition of this class is consistent with the abstract architecture because it shows that NonHumanPlayer must inherit from the class Player.

<u>Slider Menu & Gameover Screen</u>
Changes
- A slider was added to the menu to allow users to choose how many human players there are.
- A gameover screen was also added, which is shown when the game finishes. This screen includes an exit button, which allows users to close the game.

Justification
The number of players slider will allow users to choose whether they want to play against AI or not, improving player choice. A slider was used for this because it is very intuitive and easy to validate. The exit button was added because there needs to be a way to exit the game, especially if it in full screen mode.
Requirements F1
Architecture
An equivalent Menu class is not present on either the concrete or abstract architectures, however the requirements state that A game may include 0 to 2 computer-controlled players.

<u>Landmark Position Changes</u>
Changes
- The location of 3 out of the 4 landmarks were changed so that they outline university landmarks.

Justification
This was done because requirement N7 states that Landmarks must correspond with real-life landmarks at the University of York, and in the old version of the game two of the landmarks were placed outside of the university. The third landmark was moved to Biology so that the landmarks remained a similar distance apart.
Requirements N7

<u>Code Completeness</u>
All features required for Assessment 3 have been fully implemented. This is shown by the requirement and unit testing.