Architecture Report

Concrete Representation of Software Structure

The following section outlines the concrete architecture for the game's software, described in UML 2. The target language for the architecture is C#, which is modified for use in the game engine Unity. Unity has been chosen because it makes the development for the game simpler as the program is tailored for game development, and C# is used due to its similarity with java syntax.
In order to create the UML diagram below, the online tool 'draw.io' has been used, and was chosen because it is easy to access from any device, meaning it is a lot easier to share between group members. Also, it was decided to stop using argoUML, which was initially chosen for the abstract architecture due to it being installed on university machines. The program frequently experienced freezes when saving, causing the changes to be lost, so argoUML was not used this time, when creating the concrete architecture.

Link to png version of diagram:
https://raw.githubusercontent.com/as2378/unlucky/master/docs/files/Assessment2/ConcreteUMLClassDiagram.png

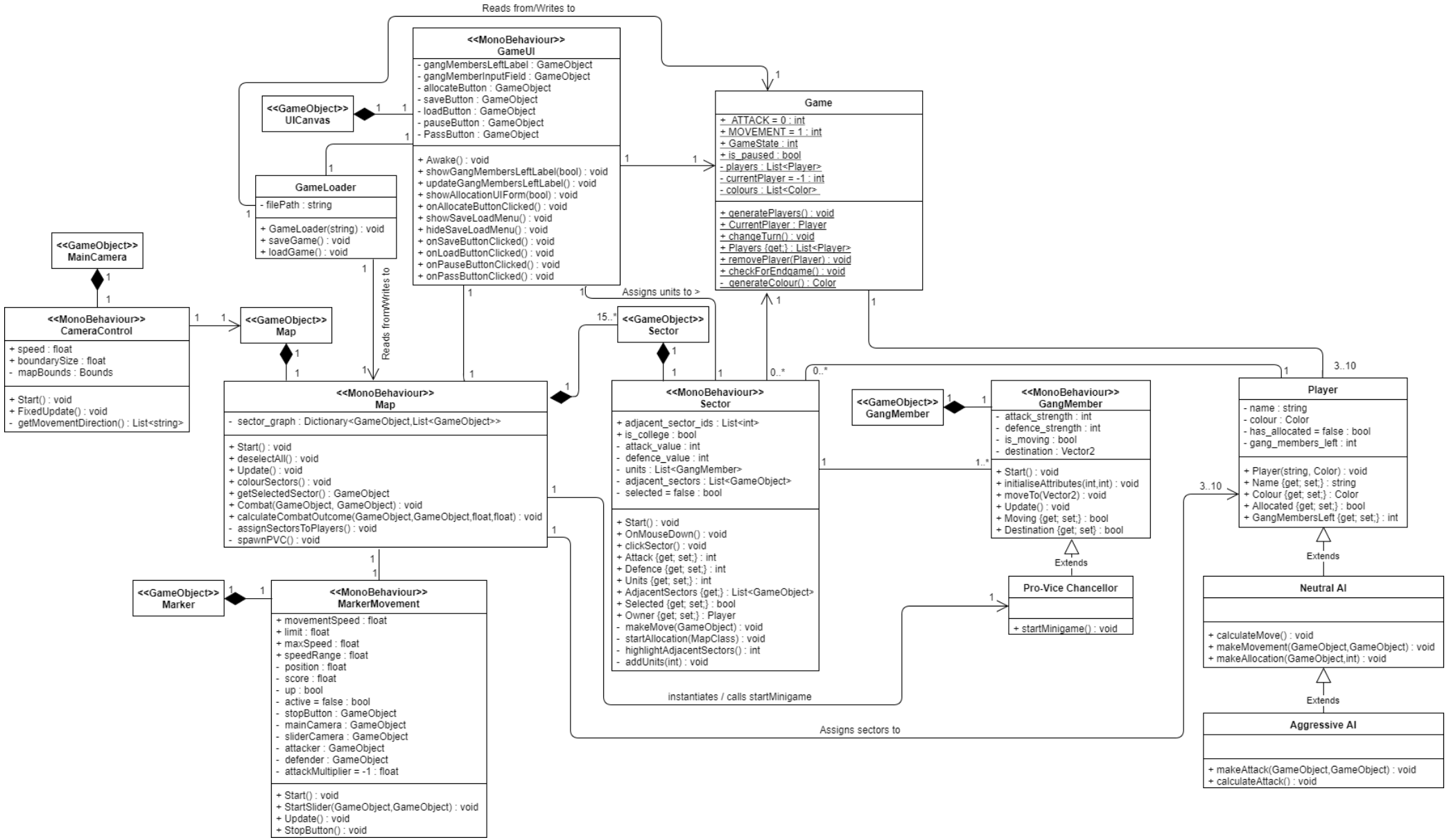

Figure 1. Concrete UML class diagram describing the structure of the game's code.

Architecture Justification:

Additional Classes:

In the concrete UML architecture diagram, a number of classes have been added which were not present in the abstract version. For example, a GameLoader class has been added that associates with the GameUI, Game and Map classes. This class will be instantiated by GameUI and it will handle the saving and loading of the game. The addition of the GameLoader class will satisfy requirement REQ-6.3, which notes that the user interface will allow players to save the current game state, as well as load previous save games.

Additionally, a class named 'MarkerMovement' has been included. This class attaches to a Marker GameObject and will be used to control its movement within the combat slider game. MarkerMovement will also handle the computing of the multiplier value for the slider game, and the addition of this class meets the requirements REQ-3.1 and REQ-3.2.

Also, a CameraControl class been added to the concrete architecture. This class will control camera movements within the game and has been included because it allows the camera to be zoomed in, meaning that the game does not seem too cluttered, and allows the game to work better on smaller displays, improving its usability.

Furthermore, during the conversion from abstract to concrete, it was decided that the architecture should show the Unity GameObjects that contain instances of the monobehaviour classes. Therefore, the UML diagram has been updated to show the Sector, GangMember, Map and GameUI classes inheriting from Unity's monobehaviour class and associating with their GameObjects. This has been done so that it makes the architecture less ambiguous and improves the traceability between the architecture and the implementation.

Sector:

Within the Sector class, various methods and attributes have been added to expand on the small number shown in the abstract architecture. The new attributes include adjacent_sector_ids, a list of integers; units, a list of gang members; adjacent_sectors, a list of adjacent sector GameObjects and selected, a boolean value. Adjacent_sector_ids is used so that developers can change which sectors are adjacent to another by using the Unity inspector window, making it easier to create maps.

Additionally, 'selected' is used in conjunction with the sector's clicking functionality, i.e when a valid sector is clicked and there are no other selected sectors, it becomes selected.

Furthermore, the majority of the Sector class' methods are used to control the clicking of sectors and making moves within a turn. The OnMouseDown() method is overridden from monobehaviour and calls clickSector() when the sector is clicked by the mouse. ClickSector() is separate from the OnMouseDown() method so that the AI can invoke a click event without needing to simulate a mouse click.

The method makeMove is used when a player makes a move during the movement phase of their turn. When called, makeMove moves gang members between two player-owned sectors and is included because REQ-4.2 states that the user should be able to split up their units and send some to separate sectors, and REQ-7.3 states that the system shall allow players to move gang members between their owned sectors.

In addition to this, the method highlightAdjacentSectors is used to show the players which sectors they can attack or move gang members to, meeting requirements REQ-2.1 and REQ-4.1. These requirements describe how the system should show the player which available sectors can be moved to or attacked.

The Sector class also has a few changes to its association with other classes in the UML architecture diagram. For example, Sector now has a directional association from 0..* Sectors to 1 Game. This association has been added because the Sector class needs to read values stored in Game, such as the CurrentPlayer and GameState.

Additionally, there is a one-to-one association between Sector and GameUI. This has been included because one GameUI is used by Sector to show the unit allocation menu, and the same instance of GameUI allocates gang members to a single Sector (REQ-1.2, REQ-1.3).

Player:

There is a number of changes made to the Player Class when compared to the abstract architecture. For example, the subclass 'Human' has been omitted from the concrete architecture because it is an unnecessary class, as there is no functionality that human players have that AI players do not. Therefore, human players are now represented using instances of the Player class, and the AI classes add to the human player functionality (REQ-7.5).

Additionally, Players are no longer directly associated with Gang Members, as they did in the abstract architecture; Players can control Gang Members and access their information through the Sectors they own, making the direct connection redundant.
In relation to attributes and methods, Players now store a colour, which is used to change the colour of the sectors. This has been chosen so that each Player is assigned a unique colour, allowing users to quickly know who owns which sector in the map, improving playability. Also, has_allocated, gang_members_left and the accessors/mutators for each, are used for the spawning of units, which meet REQ-1.1, REQ-1.2 and REQ-1.3.

Map:

Overall, Map has links with Player, GameUI, MarkerMovement and Sector (through the Sector gameobject). It was decided that Map should associate with MarkerMovement and control the Combat between Sectors (REQ-2.2,REQ-2.3) because the purpose of the Map class is to provide methods that function over multiple Sectors, and the combat requires two Sectors. Also, Map has a directional association with Player because the map class is used to assign gang members to players, for example, during combat. However, there is no longer a connection between Map and Game because Sectors now associate with Game, as explained above.
Furthermore, Map contains the attribute sector_graph, which is an adjacency list implemented as a dictionary, where the keys are the sectors within the map and the value is a list of sectors adjacent to the key (REQ-5.1). An adjacency list has been chosen over a matrix representation because each sector will be adjacent to a small percentage of the total sectors.

GangMember:

Within the GangMember class, the Special Unit subclass has been removed because special units will be improved versions of the basic gang members, therefore the only difference will be changes to the attack_strength and defence_strength attributes. Also, the addition of ‘destination’ and ‘moveTo(Vector2)’ will work towards meeting requirements REQ-4.3 & REQ-7.3, which describe the movement of gang members between sectors.
Also, the Pro Vice Chancellor includes a method ‘startMinigame’, which will be used to run the PVC minigame when a player moves into the sector containing the Pro Vice Chancellor, as stated in REQ-8.2.

Game:

All of the Game classes attributes and methods are static, meaning that the class does not need to be instantiated. This was done because there only needs to be a single instance of Game, allowing any of the classes to access the same information from the class.
The ‘turn’ attribute from the abstract architecture has been replaced by its implementation, which includes a list of players, a reference to the current player and the method changeTurn to change whose turn it is (REQ-7.2). The ‘turn’ attribute has been expanded in this way because the list allows any number of Players to be present in the game (REQ-7.1) and changes turn by switching who the current player is, which allows other classes to know the identity of the current Player.

GameUI:

The GameUI expands upon the User Interface class from the abstract architecture. The class’ main role involves implementing the functionality of the various graphical objects within the game, such as the pause and the gang member allocation buttons. The class also stores references to these buttons, so that they can be enabled or disabled at the necessary points in the game.
In addition to this, GameUI will implement the save/load menu, which is why a 1-to-1 association between GameUI and GameLoader has been included in the diagram, meeting requirement REQ-6.3.
Compared to the abstract architecture, GameUI now has a directional association with Game, instead of a 2-way association. This was done because GameUI requires access to methods such as ‘changeTurn’ and Game no longer needs to instantiate GameUI as it is a monobehaviour class that is loaded on startup. GameUI will also implement the ‘pass’ turn button, as shown by the inclusion of method ‘onPassButtonClicked’ and attribute ‘PassButton’ that will help in meeting requirement REQ-7.7.