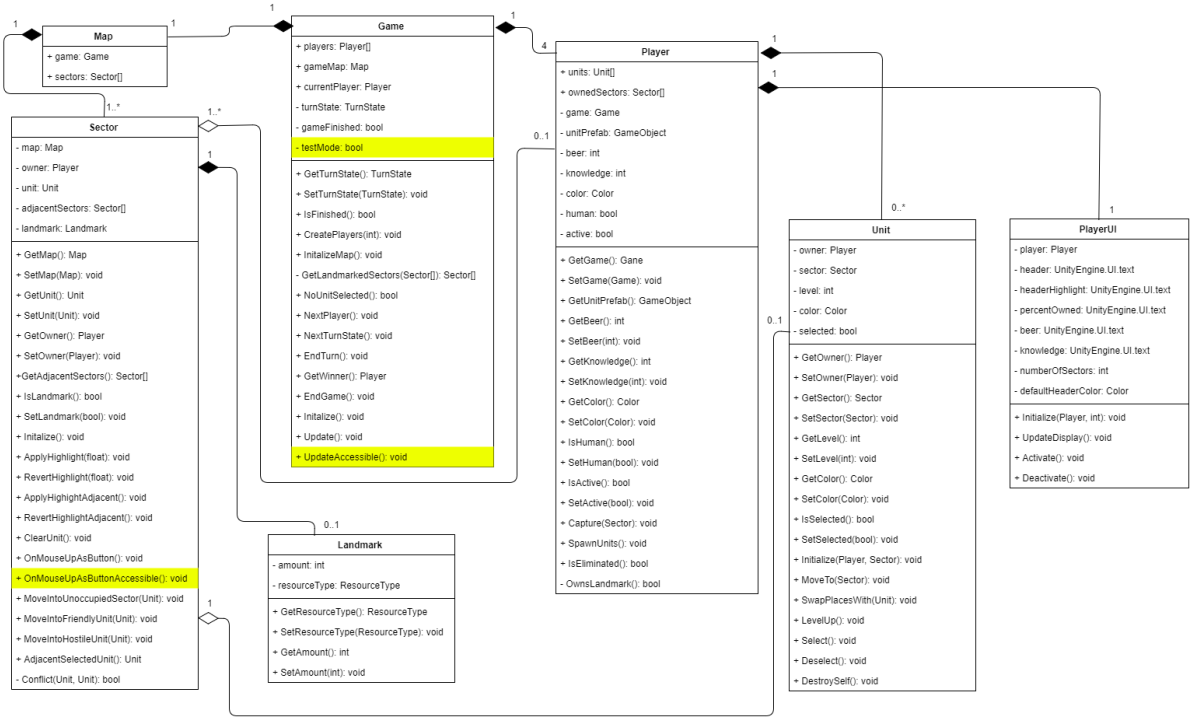
Architecture Report

This document contains the concrete architecture of our product, including an explanation and justification of each of the key subcomponents which make up each class.

To model the concrete architecture for our program, we opted for a UML 2.0 class diagram as they can accurately and concisely describe the specific structure of our product. The methods highlighted in yellow are used for unit tests and therefore doesn't contribute to the normal execution of game.

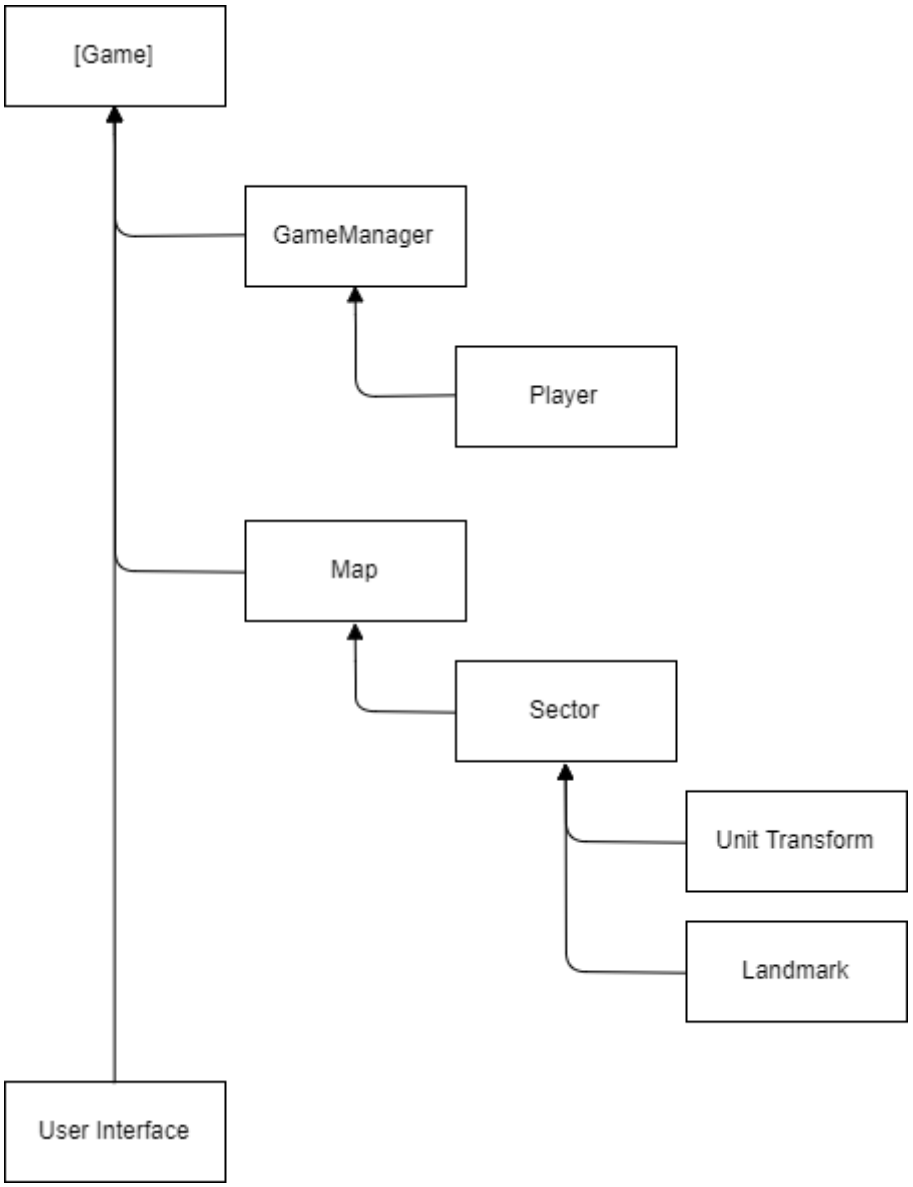
As in our Architecture 1 document we used draw.io to produce the diagrams; with all data collected from our code and design documents and entered in manually.

The following diagram shows the classes, method and attributes of all of the classes written to implement our product, including the relationships between each.



For a high-resolution version of this diagram visit:
(https://sepr-team-margaret.github.io/content/UMLC2.png)

The following UML 2.0 Hierarchy diagram shows the inheritance and groupings of game objects in our implementation. The meta-object '[Scene]' represents the game scene as a whole and doesn't represent a specific, single game object. This diagram gives us a representation of how our code is implemented.



Justification

Concrete Architecture Diagram

The concrete architecture diagram takes the form of a UML 2.0 Class diagram which includes attributes and methods. In contrast to the abstract architecture diagram, our concrete architecture diagram mirrors our implementation absolutely (attributes, methods, names, types, etc.) and leaves no room for interpretation or ambiguity.

Building on the abstract architecture, our implementation follows the same relations and binding relationships detailed in the Architecture Document.

The following explains the existence of objects and the relations between them in our concrete architecture. Square brackets contain reference identifiers to the *Requirements* document (Not 'References'). (Format is *MethodName(Parameters) : Purpose - Justification.)*

Most classes follow a standard implementation in which certain attributes are made private and can only be accessed via "getter and setter" methods; to reduce document clutter, such methods will not be mentioned. In addition Unity required methods such as 'Update' and 'Initialize' will be omitted from the following:

The ***Game*** object provides a root for all other objects. All other major objects have a composition relation with *Game*; this means that upon the termination of game, all objects cease to exist as well. The *Game* object has methods:

- **IsFinished()** : Returns True if the game is over - Assumed that the game must have an end.
- **CreatePlayers(int)** : Initializes up to 4 human players - There exist human players, there must be 4 players (either human or AI) [F1, N3]
- **InitializeMap()** : Initializes the map including initial player owned landmarks - Assumed player starts at a landmark.
- **NoUnitSelected()** : Checks if any unit has been selected
- **NextPlayer()** : Allow the next player to take their turn - Follows Player Turn Sequence Diagram.
- **NextTurnState()** : Moves player turn state to next turn state - Follows Player Turn Sequence Diagram.
- **EndTurn()** : Sets turn state to end of turn - Assumed that players take consecutive turns.
- **GetWinner()** : Checks if any player has won - Assumed that the game must have an end.
- **EndGame()** : Ends instance of the game - Assumed that the game must have an end.

The ***Player*** object handles sector capture and unit spawning methods. It also contains a method that assists in checking if the player has been eliminated. The *Player* object has methods:

- **Capture(Sector)** : Sets the sector as being owned by the player - Capture of sectors [F5]
- **SpawnUnits()** : Spawns a new unit for the player at all available landmarks - Unit spawning [F6]
- **OwnsLandmark()** : Checks if the player owns any landmarks - Player can capture landmarks, Sectors can have landmarks [F5, N6]

The ***Unit*** object handles unit movement, leveling and deletion of units. The *Unit* object has methods:

- **MoveTo(Sector)** : Moves the unit to the given sector - Necessary for sector capture [F5]
- **SwapPlacesWith(Unit)** : Swaps the sectors of two units
- **LevelUp()** : Increments a unit's level by 1

- **DestroySelf()** : Deletes this instance of the unit

The ***Map*** object has no methods directly associated with it.

The ***Sector*** object handles unit movement availability and conflict methods. The *Sector* object has methods:

- **ApplyHighlight(float)** : Highlights this instance of sector
- **RevertHighlight(float)** : Reverts highlight on this instance of sector
- **ApplyHighlightAdjacent()** : All adjacent sectors to this instance of sector is applied a highlight
- **RevertHighlightAdjacent()** : Reverts highlight from all adjacent sectors from this instance of sector
- **ClearUnit() :** Disassociates a unit with the sector - Involved with conflict, unit movement and capture [F5]
- **OnMouseUpAsButton()** : Allows for selection of sectors
- **MoveIntoUnoccupiedSector(Unit)** : Handles unit movement into unoccupied sector - Involved with capturing of sectors [F5]
- **MoveIntoFriendlyUnit(Unit)** : Handles initiation of unit swap - Involved with capturing of sectors [F5]
- **MoveIntoHostile(Unit)** : Handles initiation of combat - Capture of sectors [F5]
- **AdjacentSelectedUnit()** : Helper to detect if movement is valid - Involved with capturing of sectors [F5]

The ***Landmark*** object has attributes for resources associated with it but does not specifically have any complex methods directly tied to it.

The ***PlayerUI*** object handles the display of the player status cards GUI element. The *PlayerUI* object has methods:

- **UpdateDisplay()** : Recalculates the player's current statistics
- **Activate()** : Signify the player's turn - Assumed that players take consecutive turns.
- **Deactivate()** : Remove signification of the player's turn

Updated Abstract Architecture

Our initial abstract architecture was built upon heavily when constructing our concrete architecture with one only one minor change. Originally, landmarks were dependant on the existence of a game map at the highest level but in our implementation we decided that landmarks should be reliant on a sector's existence because of the way landmarks are setup in our game. Also, in line with the requirements of Assessment 2 we limited our scope to the map, conflict resolution and the allocation of gang members. Therefore the game elements relating to the PVC are omitted in our diagrams. See the updated diagram on our website.

Although our concrete architecture diagram is not directly affected by the change, our player turn diagram was also updated to mirror implemented changes in the way the game plays. An updated version of this diagram can be seen on our website. We decided to use this type of diagram because it can very easily be implemented into the game and is very strict in defining what actions the player should and should not be allowed to perform.