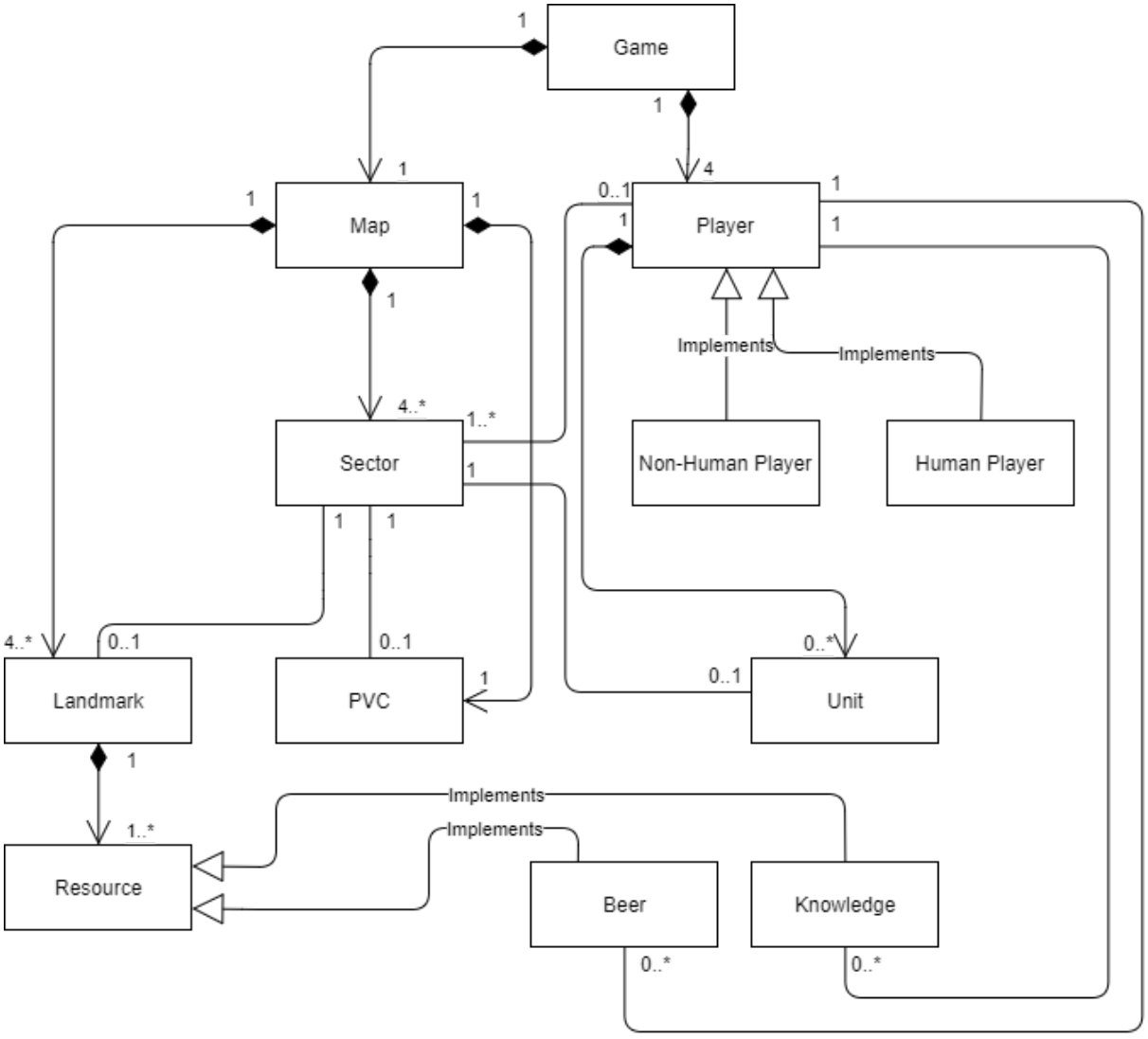
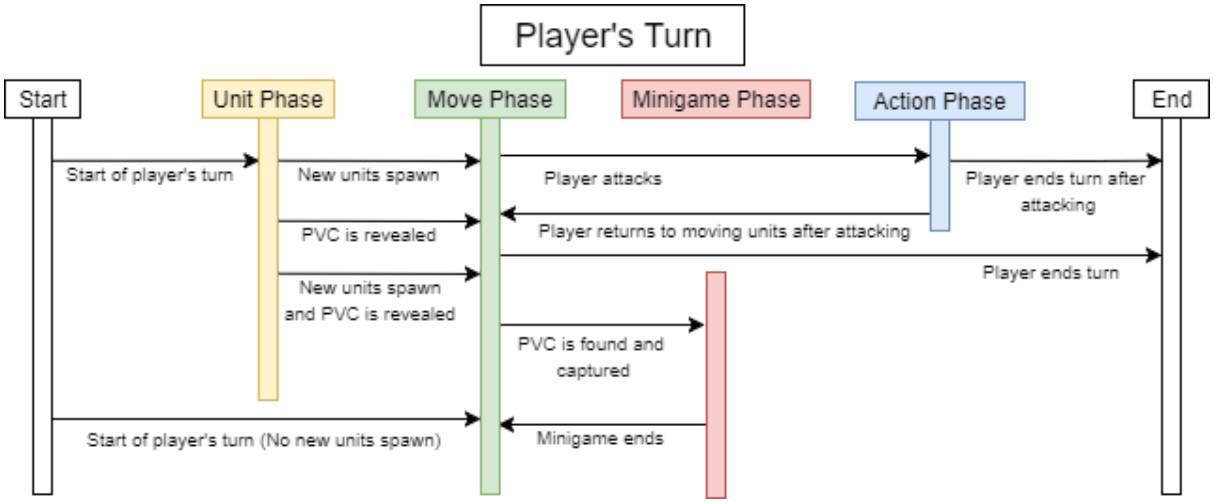
Architecture

This document will cover an abstract representation of the proposed architecture as well as a systematic justification for the architecture, explaining and justifying the individual components within it. The following UML diagram describes the game in a way which does not constrain the architecture to one specific implementation:



As an extension, we decided not to include audio and UI elements, although they are expected to exist in our implementation.

We decided to create our abstract architecture using a UML 2.0 format. UML 2.0 was easy to learn as we had access to useful lectures and documentation [1]. We found that UML 2.0 allowed us to visualise our architecture in a clear yet concise manner. In order to design our UML diagram we required a specialized tool and were pointed in the direction of a few different tools by a lecturer. We first considered StarUML but quickly decided to use Draw.io as it allowed us to save designs straight to Google Drive (our shared team workspace) and edit the UML diagram simultaneously.



The above diagram demonstrates how a player's turn should proceed. The diagram strictly shows what actions can occur from each phase without detailing how the actions are implemented.

A brief description about which actions are linked to each phase below:

- Start - Any action that occurs at the beginning of the player's turn.
- Unit Phase - Any action concerning the production of units or non-player governed actions.
- Move Phase - Any action that concerns the moving of units by the player.
- Minigame Phase - Any action that concerns the minigame.
- Action Phase - Any action that concerns unit combat.
- End - Any action that will lead to the ending of a player's turn.

Justification

When creating our overall structure, we decided that the architecture must accurately describe the main mechanics and features of the game while maintaining a level of ambiguity. Initially when preparing structure and conceptual models we opted for a more concrete model of the game including methods, attributes and implementation specific-classes. However, we quickly realised that this would stifle our ability to adapt to requirement change, and would limit our implementation to a very specific structure. As a result, we produced a new model which maintained a acceptable level of abstraction and represented the game in an implementation-agnostic way.

The following explains the existence of objects and the relations between them in diagram 1. Square brackets contain reference identifiers to the *Requirements* document (Not 'References').

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 constraints upon the composition relations define features of the game in accordance with the requirements:

- *Player*: There are always 4 players for each game [N3].
- *Map*: There is a single map in any one game [N4].

The ***Map*** object is an abstract construct for grouping, and constraining, geographical elements of the game. For instance:

- *Sector:* A map must have a number of sectors greater than 4 [N5].
- *Landmark:* There must exist at least one *Landmark* for each player [N6].
- *PVC*: There is only one single PVC at any time on the map [F2].

The ***Player*** object represents both human players and non-human, computer controlled, players. *Player* has two composition relations:

- *Game:* The *Player* object is 'owned by' *Game* and is constrained to a strict 4 players [N3].
- *Unit*: The *Player* object 'owns' a number of *Unit* objects [F6].

The ***Player*** object also has association relations with a number of other game objects:

- *Sector:* A *Player* object may own of a number of sectors [F5].
- *Knowledge*: A *Player* object may own a number of *Knowledge* resources [F5].
- *Beer*: A *Player* object may own a number of *Beer* resources [F5].
- *Unit*: A *Player* object may own a *Unit* object [F6].

Both the ***Non-Human Player*** object and the ***Human Player*** object are implementations of the *Player* object. The *Player* object cannot exist on its own as it only serves as an interface to its implementations. There exist two types of player due to requirement [F1].

The ***Sector*** object is a geographical element of the game which is 'owned by' the *Map* construct in a composition relation. Each *Sector* object has an association relation to a number of other key in-game objects:

- *Landmark*: There must exist at most one *Landmark* for each *Sector* [N6].
- *PVC*: There must exist at most one *PVC* for each *Sector* [F2].
- *Player*: A *Sector* may be 'owned' by a single player, or not [F5].
- *Unit*: There must exist at most one *Unit* for each *Sector* [F5].

The ***Landmark*** object has a composition with the *Map* object. Each *Landmark* has an association relation with the *Sector* object:

- *Sector*: A *Landmark* must only be associated with one *Sector* [N6].

A ***Landmark*** also has a composition relation to a *Resource*:

- *Resource:* There must exist at least one *Resource* for each *Landmark* [N6].

The ***PVC*** object has a composition relation with the *Map* object:

- *Map:* The *PVC* object is constrained to a single instance per map [F2].

The ***PVC*** object also has an association relation with the *Sector* object:

- *Sector:* The *PVC* object can be associated with at most one sector at any time [F2].

The ***Unit*** object has two relations; the composition previously described with the *Player* object, and an association with the *Sector* object:

- *Sector:* A *Unit* object is associated with a single *Sector* [F5].

The ***Resource*** object has a composition relation with the *Landmark* object:

- *Landmark:* A *Resource* object is 'owned by' a *Landmark* object. There can be an undefined, non-zero, number of resources per landmark [N6].

The ***Resource*** object is implemented by both of *Beer* and *Knowledge* objects:

- *Beer: Beer* implements *Resource* and inherits its constraints [N6].
- *Knowledge: Knowledge* implements *Resource* and inherits its constraints [N6].

Both the ***Beer*** object and the ***Knowledge*** object have an association relation with *Player*:

- *Player*: An implementation of *Resource* may be associated with a *Player* object [F5].

Non-standard notation

In our UML diagram, we use an "implements" arrow when we wish to denote that the parent object cannot exist without extending to one of the child objects. For example, a "*Player*" cannot exist without the object being a "*Human Player*" or a "*Non-Human Player*". Both of the child nodes share the same properties as "*Player*" but have specific additional properties that make them distinct from each other.

References

[1] "UML Class Diagrams: Reference", *Msdn.microsoft.com*, 2015. [Online]. Available: https://msdn.microsoft.com/library/dd409437%28VS.140%29.aspx. [Accessed: 07- Nov- 2017].