Architecture and Modelling

Architectural Tools

We have used UML (Unified Modelling Language) to describe the architectural design of the program, as it is a commonly-used standard model for visualising system design [1]. We have utilised a class diagram to portray the program's structure and state, and collaboration diagrams to describe the behaviour. We considered a number of programs for creating UML diagrams, but they turned out to be rather restrictive and complex to use, and as such were rejected. We have instead made use of Google Drawings [2], allowing us a more freeform approach, requiring less time to learn to use, and also supporting easy collaboration between team members.

Since it is the beginning of the design process of the system, we decided to make the architecture design quite conceptual and abstract. This allows us greater freedom when implementing the design. The team agreed to go with a fairly conceptual static view class diagram where only the main classes were specified by the requirements [3]. These were then modelled along with the main attributes they would need to store and the basic functions that they would have to perform. The class diagram also included the associations and compositions between some of the classes and the game class as to specify that everything only exists for as long as the game exists.

We also included a state machine to define the flow of the data throughout the system and how the game will progress given the requirements that have been specified. This allowed us to create a basic overview of how the system will work, showing the progress throughout each of the phases specified in the requirements. We also give insight as to how it checks to see if the game has ended according to the specification, and present a basic overview of the idea of how the winner will be decided at the end of the game. Such flexibility still allows room for changes during implementation given the fact that the gameplay may affect design decisions, so a concrete state machine is again unrealistic and not possible, just like a concrete class diagram.

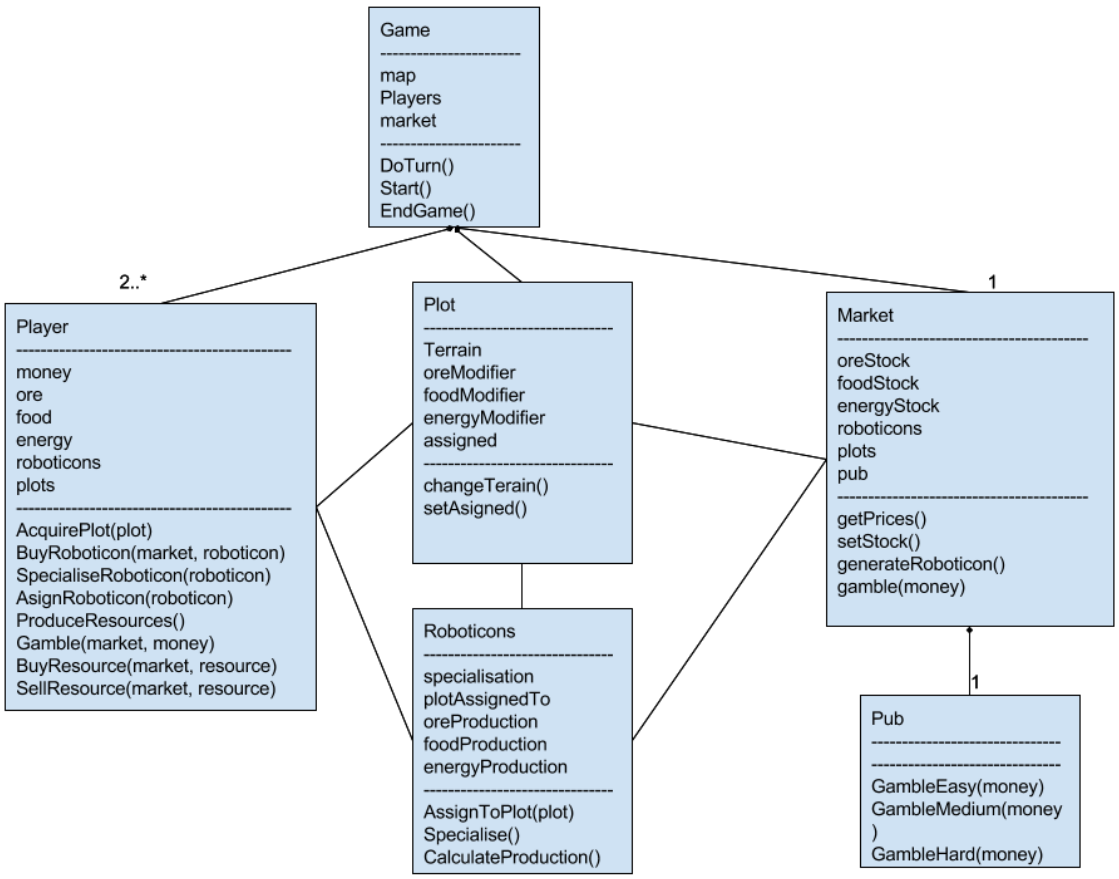
Collaboration diagrams are an effective way to visually represent the actions and interactions a given object will undertake in response to a trigger from an external source, such as a state change or the actions of another object. Other than the core "Game" entity, there are two main object types in this design that have complex action sets and interactions to carry out in response to triggers, as explained later in this document. These descriptions provide abstract overviews of the object's behaviour, but leave enough room for the details to be refined and altered as the development progresses.

---

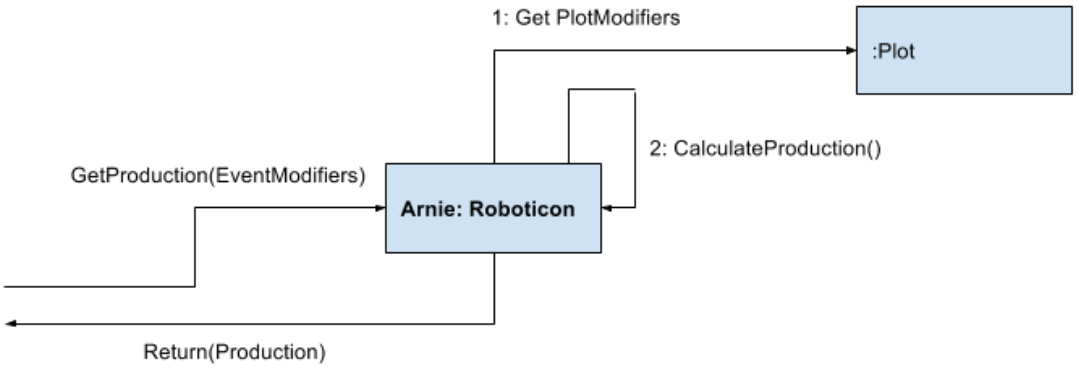
[1] Object Management Group. *"Welcome To UML Web Site!,"* www.uml.org. [Online]. Available: http://www.uml.org/ . [Accessed: Nov. 01, 2016].
[2] Google. *"Google Drawings - create diagrams and charts, for free.,"* drawings.google.com. [Online]. Available: https://drawings.google.com/ . [Accessed: Nov. 11, 2016].
[3] J. Rumbaugh, et al., "UML Waltkhrough" in *The unified modeling language reference manual*, J. Carter Shanklin, et al., Eds. Reading, Massachusetts: Addison-Wesley, 1999, pp. 25, 28-30.

Architectural Diagrams

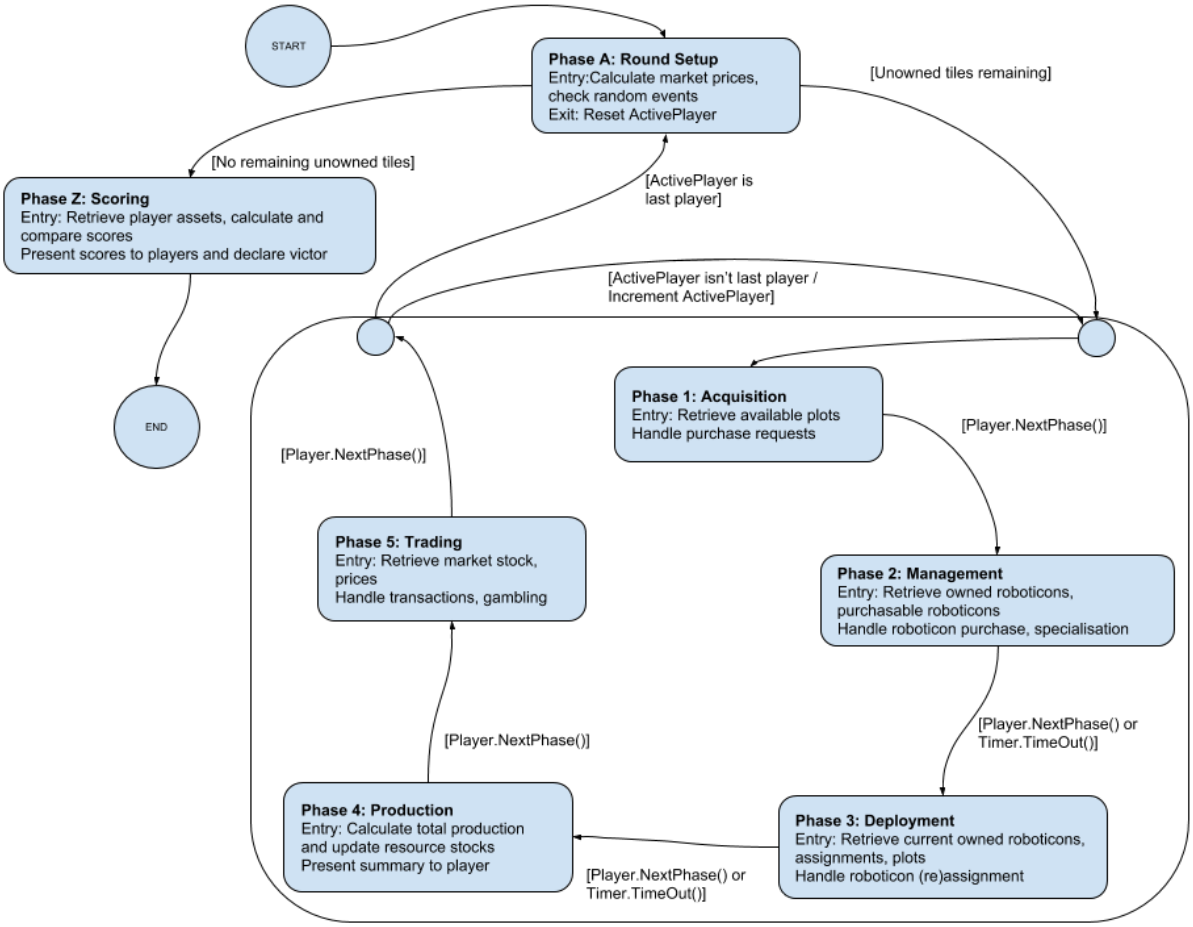
UML Class Model



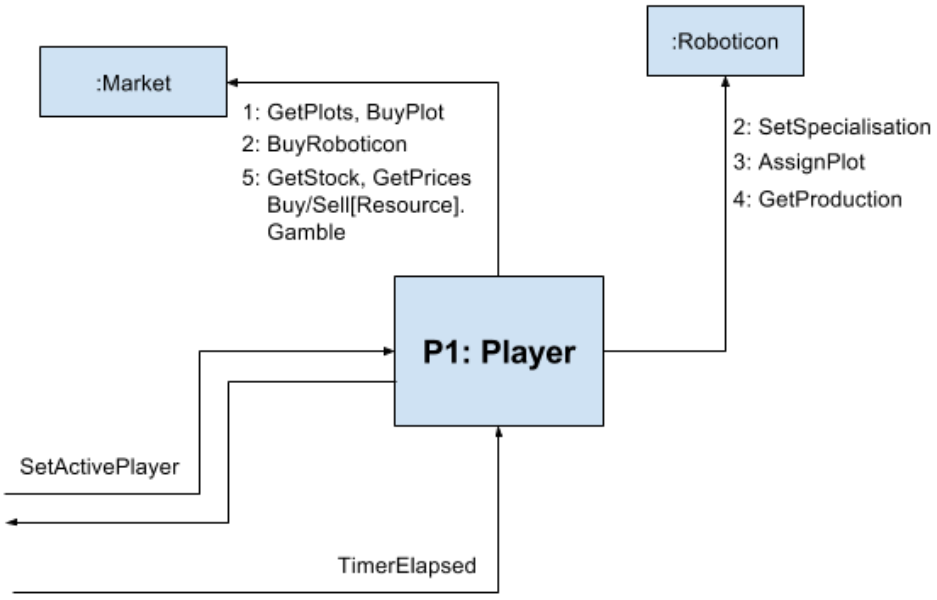
Collaboration Diagram: Roboticon Class



State Diagram



Collaboration Diagram: Player Class



Architecture and Diagram Explanation

Class Diagram

The "Game" class is the class that actually runs most of the game logic, such as changing phases and calculating the winner. Since it does most of the logic, it needs the tiles, the players, and the market in order to do the logic required.

Since the requirements state that there must be at least two players, there is a "Player" class which is specified to have at least two instances which exist in the game for as long as the game exists. The player class stores the money, ore, food, energy, roboticons and plots that the player owns. These attributes should only be directly accessible by the player. Many of the functions that the player performs will need to use a lot of these attributes, which means they are unique to each instance of the player class, for example: the list of plots that player 1 owns is different to the list of plots that player 2 owns. According to the requirements, each player must be able to acquire a plot, buy a roboticon, specialise that roboticon, assign that roboticon, produce resources, gamble, and buy and sell resources. These are actions that are performed by the player; they are functions in the player class, and considering the player class is not able to access the market directly, they must be given for the player to be able to gamble, buy resources and sell resources by the "Game" class that stores them.

Since the game must have a market, there is another composition between the game and the market similar to the game and the player, but this time there should only be one instance of the market. The market should be able to have roboticons to sell, resources to sell, and a list of plots which have not been bought by a player. Since there is only one market, the market should have references to these plots. The market should also have a pub which is where the player goes to gamble. Hence, a function that the market performs is *getPrice()*, which gets the prices of all the resources which are given to the player, since the prices of the resources change for supply and demand. There is also a set stock to reset the stock at each turn to keep with what the requirements require. Also, another function is *generateRoboticons()*, since the market needs to generate roboticons to sell to the players, and finally *gamble()*, as this is how the player will interact with the pub through the market.

The pub is used to store the different gamble functions which vary on probability of win and what the return will be if the player does win. At the moment only three difficulties are used for modelling, and these may change based upon gameplay.

The roboticon must be able to identify which plot it is assigned to so that when the player needs to produce their resources they can go to the roboticon and use the function *CalculateProduction()*, which will use the plot assigned to access the plot which it will be harvesting, in order to get the production modifiers to calculate the total resource gain for that plot.

A plot must be able to store the modifiers to help calculate the resource output as previously described, and a plot must have a terrain for future proofing of random events to occur during turns which is yet to be modelled and will be done so in a more concrete view where certain gameplay aspects have been decided. A plot must be able to identify if it has a roboticon assigned to it and must also be able to change this value with *setAsigned()* whenever a roboticon is assigned or unassigned to the plot.

State Diagram

The initial project brief suggests a number of discrete states and events in the game's structure, including two areas of repetition. This can be expressed as a state diagram implementing a pair of integrated loops with a few condition tests and non-looping states, as follows.

The game states have two main cycles they iterate through: the inner loop handles turn phases with a cycle representing a single player's turn, while the outer loop represents whole rounds and handles per-round events. These events can include the recalculation of market prices, or the generation of random events. The inner loop will run once per player for each iteration of the outer loop, cycling through each player in turns, while the outer loop will run until the condition for the game's end is met.

The inner loop consists of five states, each representing a phase of a turn. These states restrict what actions players can take, retrieving and presenting appropriate information and options based on the current phase. At the end of each cycle, the loop will check if the currently active player is the last in the turn order. If so, the inner loop will terminate and return to the outer, otherwise it will increment the active player and repeat.

The outer loop is somewhat simpler, consisting of a round setup state, that triggers various procedures to prepare for the next round, and the inner loop that handles the actual turns. Upon each iteration of the outer loop, a check is made for unowned tiles. If there are none remaining, the loop terminates, and the game proceeds to the scoring phase and the conclusion of the game.

The scoring phase is the penultimate state of the game, calculating and presenting the scores to the players and declaring the victor, before the game terminates.

Collaboration Diagrams

There are two main types of object in this initial abstract design for the game that actively act and interact, aside from the central "Game" entity, which is somewhat impractical to represent in a collaboration diagram at this time. The "Player" and "Roboticon" objects each have an abstract but clear set of interactions that can be defined, while other objects, such as the plots and the market, largely serve to represent and store data related to certain game entities, and as such are generally acted upon rather than acting themselves.

The player begins acting when it is set as the *ActivePlayer* by the game engine. As the turn phases progresses, different options will become available to the player.

First, the player will retrieve a list of available plots from the market, and purchase one of them. It can then buy roboticons from the market, if there are any available, and assign specializations to those it owns. In the third phase it may then assign owned plots to its roboticons. In the fourth phase, the player will retrieve the production values from each roboticon it owns and add them to the current stocks. Finally, it retrieves the stock and prices from the market and may execute transactions dependant on the retrieved information and its own assets.

During certain phases, it may receive a notification that the phase timer has elapsed, and should conclude any current actions and move on to the next set.

One of the main procedures for a roboticon is calculating its production each round. The roboticon will be passed the event modifiers on the player (these are generally wide-scale, but temporary effects from random events) and will also retrieve the modifiers from its assigned plot. These are then applied as multipliers on the roboticon's base production to generate the final result that is returned to the player. Additionally, the owning player may also assign a new specialisation or plot to them.