1.0 Introduction

While designing our architecture it is important that the team understands both the user requirements and the design/flow of the game. To accomplish this we've ensured that all members working on architecture are familiar with our use cases and understand the team's vision for the design of the game. In order to communicate to the team our understanding of this design we first created a flowchart from the users perspective. This was then used to consider what functionality and objects we would require to implement and converted these findings into a UML class diagram.

In an attempt to keep the flowchart readable and understandable we have omitted some details. We do not discuss all the choices and/or actions a user can possibly perform, for example we may note 'user sells resources on market' instead of having a series of decisions of what the player sells or how much. You may also realise that in phases 4 onwards we have omitted the case for the player cancelling their choice and returning to the market, again this was to ensure that the team focus on the general structure rather than small navigational issues.

Our UML class conforms to UML 2.0, we have chosen this as it is supported by our tools and it allows for the complete expression of our game. As the structure of our architecture is likely to change and evolve we have kept detail to a minimum. In the diagram you'll see class names and their associations, following the diagrams are justifications for our structure.

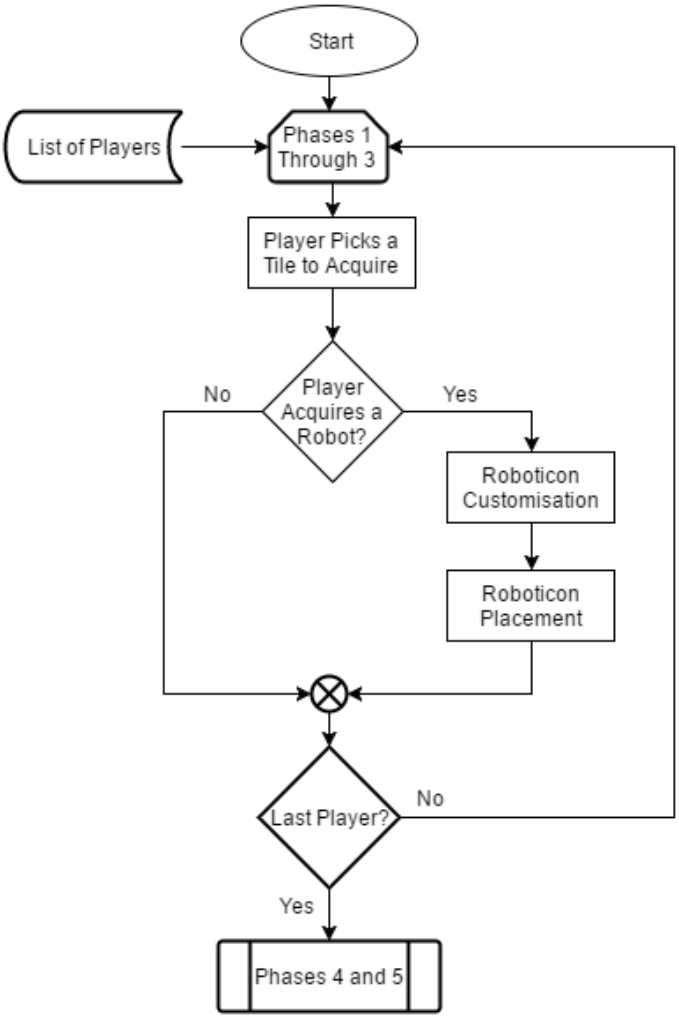
1.1 Tools

Both the UML class diagram and flowchart were developed within draw.io. Draw.io is a diagram creating tool that supports class diagrams, sequence diagrams, flowcharts and entity relation diagrams. We chose this as it is free, simple and allows for integration with Google Drive, this means that multiple members of the team can develop/edit diagrams simultaneously.

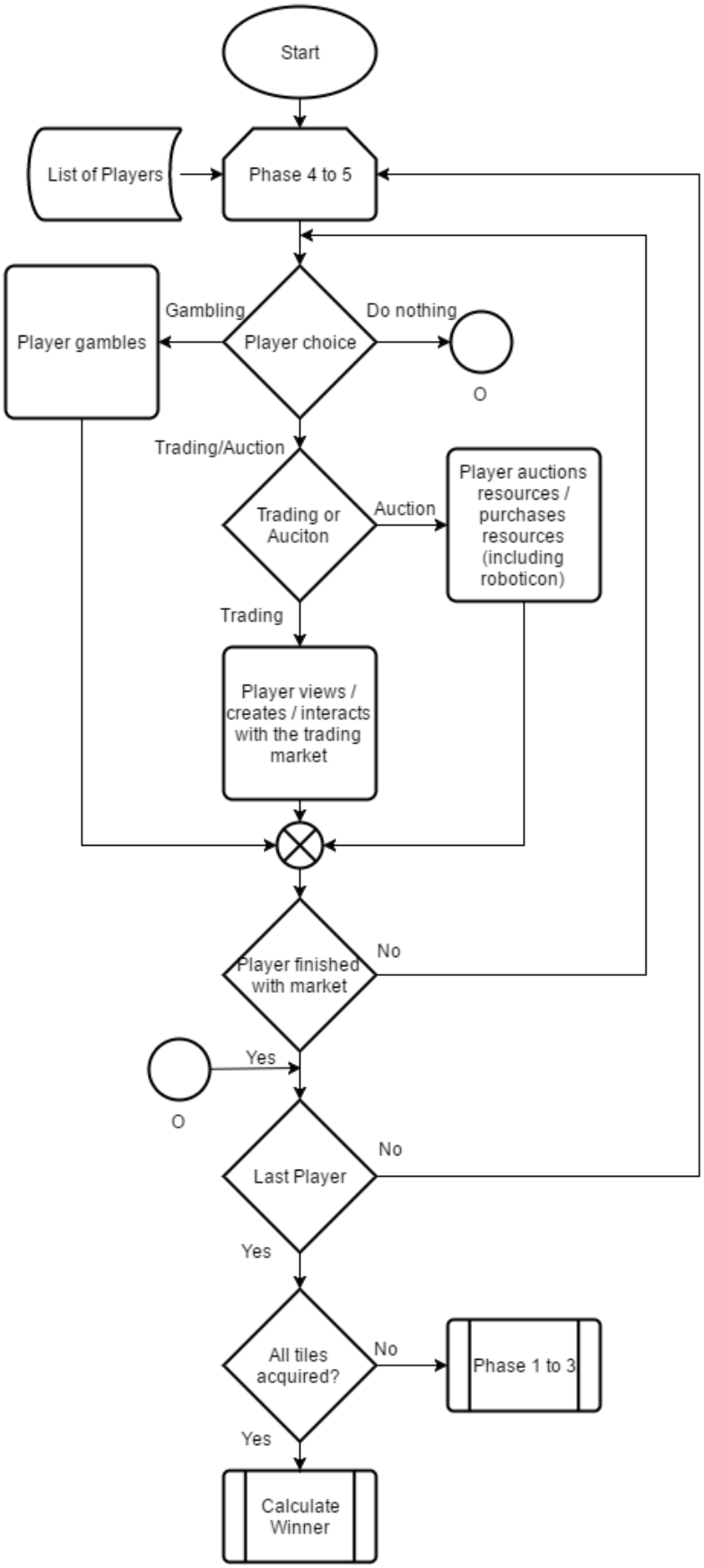
1.2 Structure

1.2.1 Game Phase Flowcharts

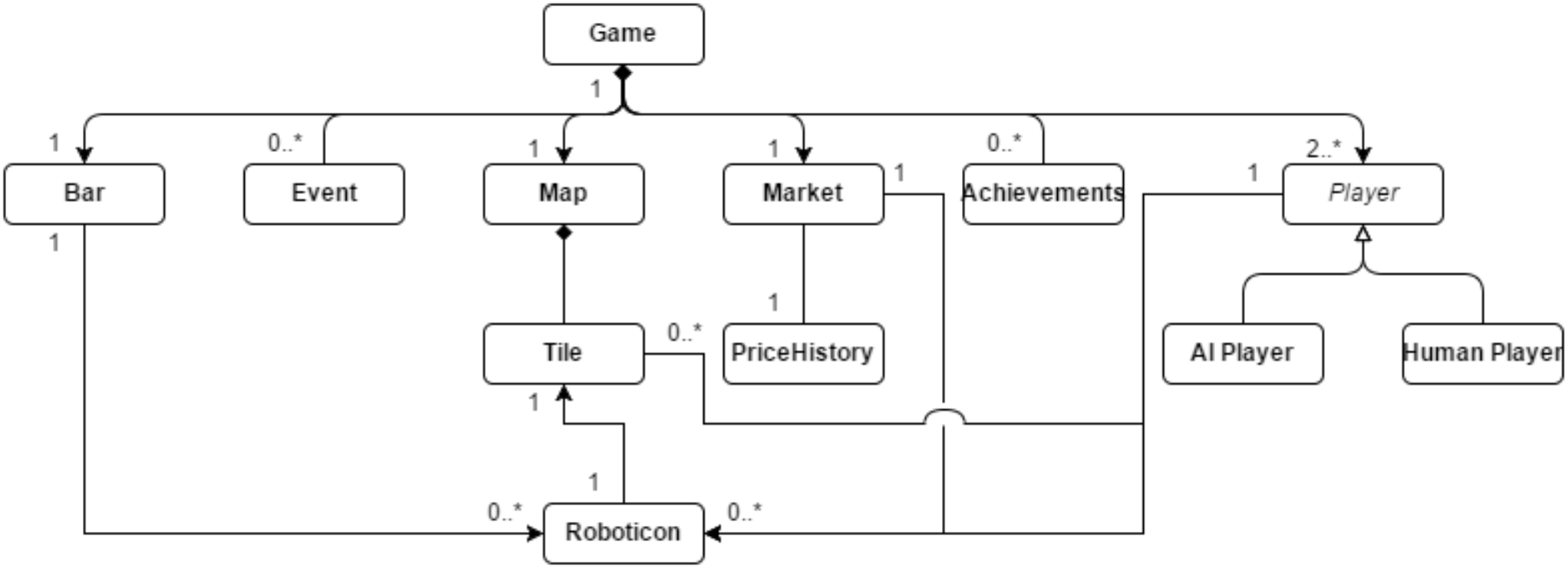
Phases 1 - 3



Phases 4 - 5



1.2.2 UML Class Diagram



2.0 Justification of Architecture Design

During the design of the architecture we have made many decisions in regards to how game entities interact with each other. While for the most part these decisions are intuitive, some required a large amount of discussion to get right. To ensure that all members and interested parties are aware of this reasoning, the following describes any choice we made that may not be intuitive with their justifications.

The first major decision we made was to develop our game in an object oriented (OO) language. We chose this as the project can be intuitively represented by a collection of objects and their interactions. In addition to this, the project structure requires an architecture that can accommodate iterative development and modular code - these are again features of OO programming.

OO programming allows for the use of inheritance and polymorphism, we believe through utilising these methods we can implement both human and AI players with ease. Inheritance allows classes (children) to acquire the attributes of another (parent), as such both human and AI players will inherit a set of methods and variables. Without the presence of polymorphism this could prove a problem as human and AI players, while requiring the same functionality, operate in vastly different ways. Polymorphism will allow method overloading meaning that while both types of player will retain the function names/purpose of their parent they can process the calls in different ways.

We have decided to include a Game class that will manage the sequence of phases while keeping references to most game objects. This allows for the central management of the game from a high level which results in a structure that can easily integrate new phases/objects. Furthermore, such a structure allows for overall game maintenance to be handled more efficiently. The Game class calls object methods in an order and deals with their data, meaning that the structure of the game is easier to understand. This is because the Game class does not need to be filled with game logic, as this can be contained in the objects it has references to. We believe that through adopting this structure, future developers will find our code easier to understand and maintain/upgrade thus improving resale value.

When designing the games trading structure, we decided to make two classes, Market and Bar. Market will represent the market place that is interacted with in phase 5 of the game. Therefore, this class will contain a list of open trades players have proposed, the amounts of each resource the market currently has, and finally the prices associated with each resource. Bar will also be utilised in phase 5, and will contain the logic that allows players to gamble. We decided to split into the two classes described above for numerous reasons. One reason is that in any given game round players do not need to interact with both the market and the bar, they can do just one of those things if they wish. Splitting into two classes highlights this distinction and makes the code easier to maintain and build upon. Another reason for this split is that during gambling it is the player's resources that are being risked, not the resources that belong to Market, and so we can safely separate Market and Bar.

One requirement for the project is that of supply-and-demand economics. The customer also expressed interest in the ability to present historical data to the players so that they can make more strategic decisions. To accommodate this we require a class to store that information. Seeing as the class itself does not need to inherit any functionality from the actual market we've chosen to represent it it's own class. The Market object will contain a reference to an instance of the historical price class and update it when appropriate to include more price data.

When considering how Roboticons interact with the game we realised they are used by several different classes across many game phases. While Roboticons are used by both the Player and the Bar they don't require access to these classes and therefore the association is unidirectional. The decision to have a unidirectional association from the Roboticons to Tile was made to improve performance. During the production phase of the game Tiles are required to produce resources, instead of having this functionality within the Tile class - meaning we'd have to iterate over all Tiles of the map calling this - we assigned this to the Roboticons. This means we can iterate over a player's Roboticons and call the production method, as such we would be iteratively calling the exact number of productions needed.

Although players will now be able to reference a subset of the tiles they own via their list of Roboticons, some Tile objects may not have a Roboticon placed on them. For this reason there must be a bidirectional association between the players and tiles such that a player will be able to reference all of the tiles they own regardless of Roboticon placement. A Tile instance should be able to reference which Player owns it so that the game can determine if a Tile is available for acquisition.

We have a Map class that acts as a middleman between the Game instance and the collection of Tile objects. An alternative to this structure would be to have all of the Tile collection logic in the Game class, however, we felt this could start to clutter the Game class with data and methods that could be encapsulated elsewhere. The Map class therefore is not just a collection of Tile objects, but it also contains logic such as determining whether a Tile is owned by a Player or whether it is available.

We as a team have decided to implement events in such a way that they can affect all aspects of the game. To achieve this we require that the Event class is its own unique class instead of a subclass of some other class such as Tile for instance. In addition to this the Event class must be able to interact with all other classes, therefore the association between Game and Event is bidirectional giving it access to all of Games' associations.

Similarly to the Event class the association between Achievements and Game is bidirectional to allow Achievements to monitor the actions in the Game. The need for the Achievements class arises from our requirements and therefore to allow for this to span all aspects of the game it is important that it has access to all of the Games associations.