1.0 Introduction

As we were unsure how the inclusion of networking would affect the architectural design of a Unity project we decided that we would not design a concrete architecture at once. Instead we met as a group regularly to design each feature as it was about to be implemented, then modeled how the integration of this new feature would interact with our previous designs. We made note of the general methods, variables and concepts that were needed, and went about creating a backlog for completion by our pair programmers. Once we reached a point where our design direction was clear we created our architectural diagram before implementing any further. This allowed us to plan exactly what we needed to implement and increased our efficiency. During the remaining scrums, we announced any deviations from the architecture diagram and updated it where necessary.

The decision to model our architecture through the use of UML class diagrams and activity diagrams was made with care. Whilst UML class diagrams would be able to fully represent a C# program with ease, it has proven difficult to model the behaviour of our Unity project. This difficulty is caused by the presence of prefabs, which are GameObjects that are initialized with components and properties outside of the code. However after producing early UML mock-ups we found that most if not all functionality can be conveyed through the use of a UML class diagram so long as the reader has a very basic understanding of Unity GameObjects. As such we have modeled our game structure conforming to UML 2.0 with slight variations on the appearance of certain notations owing to our use of software, these differences are explained and addressed by a legend present on the class diagram that follows.

In addition to the class diagram, we created an activity diagram for the game controller class that shows the abstract structure of gameplay. We did this to visualise the high level flow of the game, giving names to states the game could be in at any one time. Throughout development we referred to this diagram, which gave us a logical order of what to implement. At the end of the implementation, we updated the diagram to add a few more states that we used in code that were not initially included. Despite adding more states, the diagram (and thus the game) doesn't have a terminating state, meaning it is impossible to 'win' the game. We purposely left winning conditions out of the game as this was not a requirement at this stage of the project. The activity diagram can be found here on the website [1].

1.1 Tools

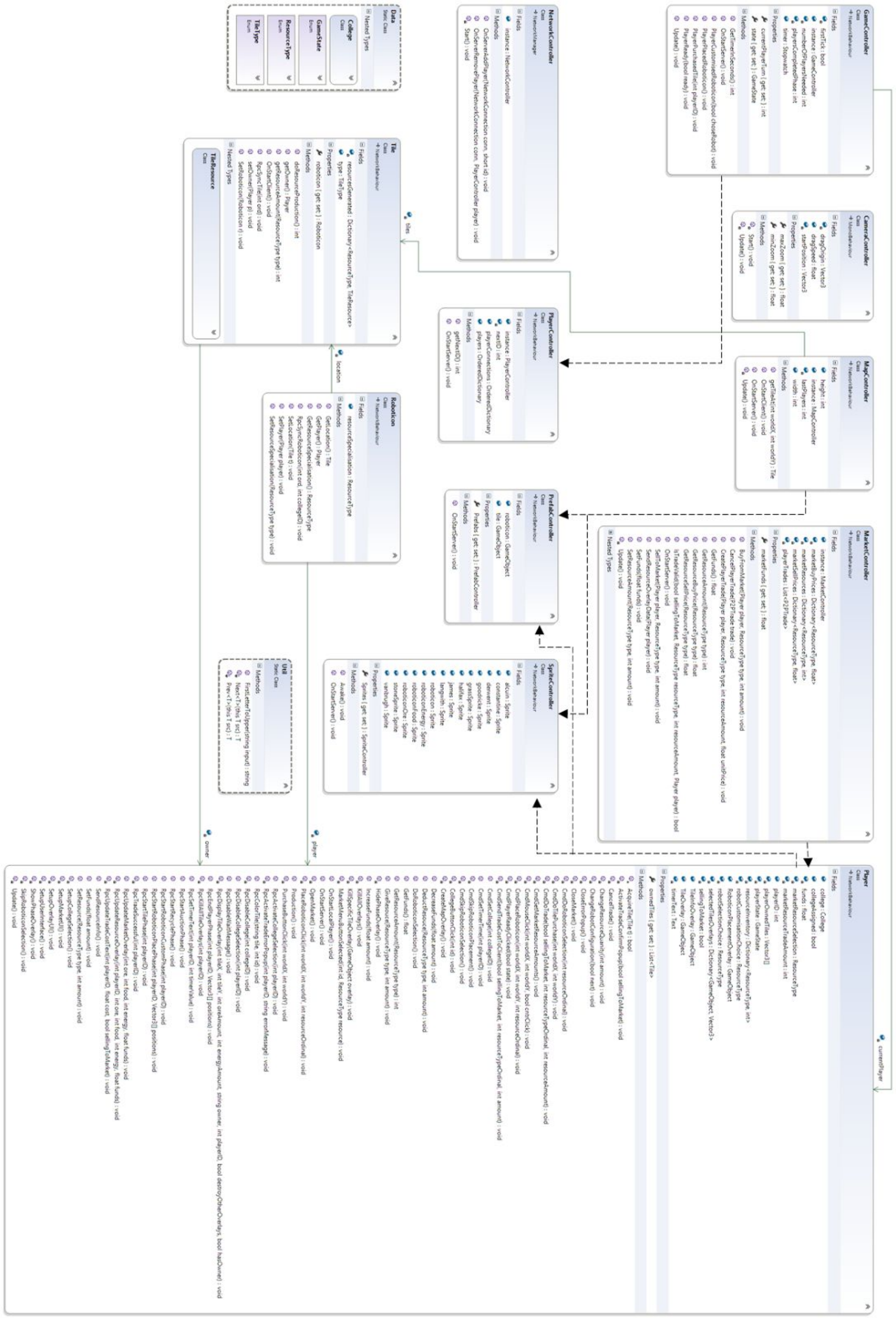
To create the start of the diagram initially we used the same piece of software used for the abstract class diagram, draw.io as we had previous experience using this tool. Another advantage of this tool is its integrates into our google drive, where all of our documentation is kept. In group meetings we constantly added to this UML diagram, reflecting the additions and changes in the architecture that were made.

Once we reached the end of the project, we realised there were small differences between the methods in the diagram and that in our project (redundant methods/deleted methods). To quickly rectify these mistakes we elected to automatically produce a partial class diagram directly from our project code through the use of Visual Studio. This included all the correct variable names and method signatures with their accessibility, ensuring their were no discrepancies between the diagram and the code. However this did not include important relations between classes. At this stage we exported this diagram as an image, and in PaintShop we added the dependency relations between the appropriate classes. We decided to use these software packages as they were readily available and simple.

Visio Studio also provided a means of creating activity diagrams and as such was also used in this capacity.

1.2 Concrete UML Diagram

Padlock represents a private attribute/method, asterisk represents protected while a lack of symbol means public.



2.0 Justification of Architecture Design

One of the main reasons why we did not create a UML diagram initially was because of the networking code we needed to implement. We understood that making our game network enabled meant that a lot of the structure of our project would differ from the original abstract UML diagram. By the time we created the concrete UML diagram we were confident with Unity networking and manipulating it to add new features to the game.

One of the biggest changes to the structure of the code was making the game centralised around the player class. For each client connected to the game server, there is a player object that is created on the network. Therefore the player class acts as a way of gathering user input, setting the appropriate UI for that particular client, and issuing commands to other server objects after appropriate player actions. Unity only allows for one type of player object to be spawned on the network, and also there were no requirements for an AI player at this time (F1) [2], so we decided not to create an abstract player class and have human and AI classes inherit from it. This does not limit our game from further expansion, however, as a player type variable could be stored within the player class, and its value checked at the appropriate times.

Changing the internals of the player class to this extent also changes the relations it has with other classes. We decided to have a collection of controller objects within the game scene, such as game controller, map controller and market controller. Each player object in the game interacts directly with instances of these controllers, and therefore the new architecture is centred about the player class, rather than the game controller class like we had originally planned. There were also some changes we made that were not due to networking changes. Originally we had the player class store a list of roboticons that the player owns, however, we realised this was actually not required at this time, as roboticons could be accessed via the list of tiles instead.

This is a concept observed in both of our architectural designs i.e. the use of controller classes which monitor and dictate the flow of the game. We believe this to be the most efficient way of controlling the game flow, encapsulating such logic within a single class improves the maintainability of our code and ensures an easy means of integrating new phases or features. The game can then be simply represented as a state diagram with calls to the functionality of relevant classes at the correct stages in the game, this integrates nicely into the concept of different game phases (F9). This also makes managing multiple players from a central location much simpler, we simply ensure that the server is the active game controller and therefore there need only be communication between the player and the server for the game to work.

It’s also important to distinguish the difference between the server side and client side of our game. All of the main controllers are server side only, as this means there is only one instance of them and all of our game logic can run server side. All spawned objects in the game have a local and server version simultaneously. This mean that the player and tile classes actually handle server side and client side code. This is why the player class is at the heart of the game. They can gather client side input, render client side graphics, and yet still process server side logic.

With the current architecture, it is possible to have up to eight human players play in the same game (this limitation being due to the number of colleges in the game - F12). This is achievable due to the way game controller and players interact. Game controller has access to a collection of players in the game, and it can dynamically control the game with any number of players. We have setup the game controller to only expect two players, however this can be changed with a single variable. The same goes for the size of the map in the game, a width and a height variable can be changed in the map controller giving the game scalability.

The game map has been implemented as a list of tiles (F2) which is of a very similar design to our abstract architecture. Whereas previously we held a reference of all game tiles within the Map class they are now held within the MapController. The GUI aspects of these tiles i.e. recognisable owners, roboticon placement and tile information (N2), are managed by the clients through the use of the Player class.

The inclusion of enums within the data class has improved our data encapsulation. This allows for easy access to constants and improves the readability and modularity of the code. Through incorporating a data class into our architecture it’s now a simple matter to manage all constant, meaning that our code is now more maintainable. For example, the resource type is access by roboticons, tiles and the market by having a central data class we have eliminated the need to include these definitions within the classes themselves.

The use of nested classes was an intuitive decision by the development team. While nested classes provide no improvement in performance or simplicity of the code, it has made development easier as closely related classes can be managed within a single document. Furthermore, we believe through nesting class we’ve provided a more intuitive architectural design. This is because most nested classes are used to support the functionality of their outer class and are rarely used by any other class. The exception to this of course is our data class which is widely used by a variety of classes.

The decrease in the number of associations between classes and the addition of class dependencies was implemented to ensure that there are no redundant attributes in our classes. While class functionality of player, roboticons and tiles are dependent on their ability to access each other most classes can function with passing instances for use in specific methods. For example, whilst the market requires an instance of player when conducting trades (F6) it does not need access to the player instances at all times therefore it is best passed as parameter rather than an having them as an attribute.

Bibliography

[1] SEPR, "Activity Diagram," [Online]. Available: https://seprated.github.io/assessment2.html. [Accessed 21 January 2017].

[2] SEPR, "Updated Statement of Requirements," [Online]. Available: https://seprated.github.io/Assessment2/Req2.pdf. [Accessed 21 January 2017].