Contents

[Related Work 3](#_Toc3401192)

[M, N, K Games 3](#_Toc3401193)

[AI in Modern Games 3](#_Toc3401194)

[Adaptive Behaviour 4](#_Toc3401195)

[Monte Carlo 5](#_Toc3401196)

[System Design 8](#_Toc3401197)

[Methodology 8](#_Toc3401198)

[System Architecture 9](#_Toc3401199)

[Design Patterns 9](#_Toc3401200)

[Observer 10](#_Toc3401201)

[Composite 10](#_Toc3401202)

[Chain of Responsibility 11](#_Toc3401203)

[Dependency Injection 12](#_Toc3401204)

[Design and Implementation 12](#_Toc3401205)

[Model 12](#_Toc3401206)

[Requirements 13](#_Toc3401207)

[Design 13](#_Toc3401208)

[Implementation 14](#_Toc3401209)

[View 14](#_Toc3401210)

[Requirements 15](#_Toc3401211)

[Design 15](#_Toc3401212)

[Libraries 16](#_Toc3401213)

[Bootstrap 16](#_Toc3401214)

[Ngx-toaster 17](#_Toc3401215)

[Modals 17](#_Toc3401216)

[Toast 18](#_Toc3401217)

[Login Screen Design 18](#_Toc3401218)

[Game View Design 18](#_Toc3401219)

[Navigation 19](#_Toc3401220)

[Controllers 19](#_Toc3401221)

[Client 19](#_Toc3401222)

[Requirements 19](#_Toc3401223)

[Design 19](#_Toc3401224)

[Implementation 20](#_Toc3401225)

[Maintenance 20](#_Toc3401226)

[Server 21](#_Toc3401227)

[Requirements 21](#_Toc3401228)

[Design 21](#_Toc3401229)

[Implementation 21](#_Toc3401230)

[Maintenance 22](#_Toc3401231)

[Services 23](#_Toc3401232)

[Client 23](#_Toc3401233)

[Design 23](#_Toc3401234)

[References 24](#_Toc3401235)

# Related Work

In the field of games AI have been a focal point in creating interesting and engaging gaming experiences. This differs from the computer science view of AI in which the goal is to make intelligent systems. In this project providing in engaging game is far more important than the intelligence of the player. This chapter will describe the work related to games; what people find interesting and fun in games. As well as, research into different methods for constructing AI and some of the technical aspects of games related to the one used in the project.

M, N, K Games

An M, N, K Game is one in which players take turns placing coloured stones on a M by N board, the winner is the first to get k stones in a row. Popular examples of m, n, k games include Connect 4 and Tic Tac Toe, a typical tic tac toe game would be a 3, 3, 3 game. Games up to 7, 7, 5 and 8, 8, 5 have been solved, player two can force a draw in both instances (Wei-Yuan Hsu, 2019). The stealing strategy argument shows that m, n, k game in which player two can have a strategy which ensures they win. As these games have finite size the game is convergent, all games finish after some number of moves, the max number of moves available being m x n. They are also zero-sum games as there is only three possible end state a win, loss or draw. Moreover, these games have perfect information and are stochastic, this makes them great candidates for tree searching techniques such as monte carlo, or depth first search.

## AI in Modern Games

Typically, in modern games AI are divided into three categories; Easy, Medium and Hard. The issue with this approach is that it either provides an AI which is too easy for the player, or one which is too difficult which can quickly become frustrating for the player (Chanel, 2008). To combat this issue game developers have turned to techniques known collectively as Dynamic Game Difficulty Balancing (DGDB). This is a process which measures a player’s ability in some form, whether it is accuracy in a First-Person Shooter or number of pieces won or lost in a board game. The data collected is then used to modify parameters in an attempt to keep players entertained. There are many ways this can be implemented is to passively help the player by giving them more resources, such as health packs and ammo, in an FPS setting (Hunicke, 2004). Though this system was found to be effective it did result in situations which were not intended. A solution to this is to use the data collected to modify the behaviour of Non-Playable Characters (NPCs). To take this a step farther in the 2014 game “Shadow of Mordor” featured what it called the “Nemesis System”, this was a system that tracked hostile NPCs and their encounters with the player which the NPC would comment on in their next meeting with the player. This coupled with the NPCs personality gave the player a more in-depth experience with the game as interactions were more dynamic and let the player create stories independent of the main plot line of the game (Donlan, 2014).

## Adaptive Behaviour

As discussed previously having AI which differ drastically from the player in terms of skill level results in games which are not fun to play. Therefore, it is important to have AI which can satisfy players from both sides of the spectrum. Moreover, dynamic difficulty AI in particular has been shown to increase the players enjoyment of games, specifically in a Real Time Strategy game (Mirna Paula Silva, 2015). There are some requirements of dynamic AI which must be met before they would be suitable for use in a game, however. In the case of machine learning dynamically scripted agents it is important that they are quick in their computations, effective throughout the whole game (even while learning), robust to the randomness that is inherent in most games and finally they must be efficient in their learning as games can be short and similar situations will seldom happen in a single game (Spronck, 2006). Besides machine learning there are other methods that can be used to scale the difficulty of an AI opponent to a more appropriate level for the human player. For example “High-Fitness Penalising” is a system which gives higher rewards to mediocre moves instead of on how well the AI system is doing (Spronck, et al., 2004). Games are reported as being more fun when the difference between wins and losses are minimal and draws are uncommon (Tan, et al., 2011). It was shown that this can be formalised into the equation:

Where W, L and D represent the number of wins, losses and draws respectfully and n is the number of games played. The idea of artificial stupidity is also mentioned in Tan et al’s work. This is an AI system which makes plausible mistakes intentionally to help make the game more entertaining (Lidén, 2004).

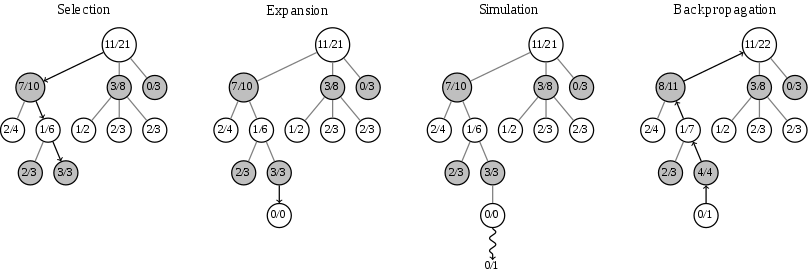
Lidén talks about AI in the context of an FPS however. These include:

* Have Horrible Aim: As the AI can play optimally it is important that it intentionally misses so the player has a chance.
* Move before firing: Give the player a chance to react to the AI actions.

Moreover, it is mentioned that an AI opponent should feed into the players emotions to enhance the gaming experience most notably in the “Pull back at the last minute” scenario. Though Liden does discuss in the previously mentioned section that the AI should let the player win the game, this goes against what Tan et al. say as excessive wins would maximise their equation and would suggest an unfun game. However, in this project the Minefield AI was built to accomplish the goal of raising the tension in the game even if it does not play very well.

## Monte Carlo

Games can be modelled as trees where each node in the tree represents the game state and each connection represents the action taken to move between one game state to another. To decide which move is better than some other move one method is to simply count how often that move results in a win for the player compared to the others. The move which results in a win most often must be the best. The issue is for some games, such as chess or go, the game tree can become quite large quite quickly. As the game of Ultimate Tic Tac Toe (UTTT), the chosen game for this project, has an estimated 1.8 x 10^78 game states (Bredyn McCombs, 2018) it would be infeasible to perform an exhaustive search of the game tree and “Brute Force” a win. Monte Carlo Tree Search (MCTS) aims to make decisions based on the estimated chances of winning the game based on that move. The way it does this is by building up the tree asymmetrically, focusing on more promising sub trees and ignoring less promising ones, and more importantly by doing this it performs better than more classical algorithms in large trees. The algorithm for MCTS follows four basic steps.

* Selection: Starting from the root node, reclusively select the child node which looks most promising (in this project the UCT value will be used to calculate this) until a leaf node is reached.
* Expansion: If the leaf node is not a terminal state of the game, generate all child nodes from the leaf and select one of them. Child nodes are any valid move from the leaf node.
* Simulation: Complete a random playthrough of the game from the selected child node.
* Back Propagation: Use the result of the simulation to update the nodes on the path between the child node and the root.

The UCT formula is a variation on the UCB1 (Upped Confidence Bound) that is used in bandit problems. The formula has been shown to have performance advantages over its competitors in some artificial game domains (Kocsis & Szepesvári, 2006).

Where is the reward for node i, C is some constant, typically it is in a pure monte carlo implementation, N is the number of visits the parent node of i and is the number of visits to the node i. This formula balances out the exploitation of a node, the first term, against the exploration of a node. This ensures that nodes are visited at least once.

# System Design

The system was designed to allow many users to interact with the application while at the same time, while allowing future development work to take modify the behaviour of the application relatively easily. Test driven development was used throughout the creation of the application as it allows more robust and maintainable code to be developed.

Methodology

An Agile development cycle was implemented for the implementation of the application. The Agile development cycle follows a four-step cycle in which feedback about previous development is fed into the next development cycle (see figure 1), so the product better reflects the wants and needs of the client. The cycle that was used in the development of this system is on the simpler side as the project does not have the industrial scope that is normally paired with this type of development. However, the advantage of the Agile development cycle is that, unlike waterfall, it allows for requirements to be updated during the entire development process. Frameworks are normally used in conjunction with Agile to help developers maintain the work load and make sure all tasks are completed. The tool that was used in this project is Azure Devops. Azure devops maintained the work board, this contained all work completed and work to be completed, known as the backlogs, and allowed me to quickly and easily check what was next to be done. The use of these tools is not necessary in agile but by having them a development team can better manage their weekly work load and better assign resources to the tasks that need it. The Build step could be farther broken down into another cyclical development cycle, that being Test Drive Development (TDD). The TDD cycle is a simple one but is designed to make code flexible and robust. It is intended for paired programming but as the process creates a system with high test coverage and gives the developer much higher confidence in the code it was used for this project. Along with its other benefits TDD allows code to be documented while it is developed. This project contains two hundred and forty tests, each one describing a small piece of functionality and each piece of functionality working as the product will not be published if tests do not pass

Figure 0‑1 Agile Development Cycle

## System Architecture

The system will be a web app containing an Angular front end and an Asp.Net Core back end. The choice of a web app was made as this would allow more people to access the application easier as well as have the user experiments take place in the application. The server follows a Layered Architecture with three layers; the controller layer, business layer and the data access layer. The layered architecture allows for each layer to be easily changed without disturbing any other layer. It also clearly separates different layers of abstraction such as retrieving data from a database from interacting with the game logic. The system could easily be changed to support different types of databases for example it currently interacts with a SQL database but changes could be made to the data access layer to interact with a no-SQL database or some other form of storage. The first two layers also behave as the controller and the model in a Model-View-Controller (MVC) architecture. The View is the HTML front end again the MVC architecture allows for parts of the system to be easily changed as long as all concrete classes satisfy the interfaces in the system, this principle is known as “Programming to and Interface” (Head First design). Lastly the front end uses an Event Driven architecture to communicate between components. This was chosen as being a web application connected to the server through the internet responses from API calls would take an unspecified amount of time to return, as such events are used to signal these returns to any component that is interested in the result. Moreover, as the front end needed to interact with the player, eg clicking on the screen (an interaction which is easily modelled with events), an event driven architecture was a good solution to all these problems.

## Design Patterns

A Project of this size must employ established design patterns that allow code to be flexible to change and easily extendable. As such this project features several different design patterns for behaviour and creation.

## Observer

The observer is a common design pattern normally used to communicate updates in a model, returns from asynchronous calls or user events. The pattern was used exclusively in the front end and allows components to maintain a loose coupling on each other while still communicating updates to some part of the system. The participants in a typical Observer pattern are:

* The Subject: The class which is being observed, it provides some mechanism for adding and removing observers
* The Observer: The interface which is called when an event is fired, typically it only has the “Update” method defined
* The concrete Subject: The object which updates observers of change
* The Concrete Observer: Implements the Observer interface.

As Angular was used as the framework the implementation of the Observer is slightly changed in that classes do not implement an Observer interface, instead the subject class has an “Event Emitter” field which can be subscribed to. In effect this is analogues to adding the subscribed object to the subject objects’ observer list, which would be done in a typical observer pattern. On subscription an anonymous function is passed to the subjects’ event emitter which is called when an event is emitted. This implementation of the Observer pattern accomplishes the same goal as the more traditional pattern, but it does not require observer classes to implement an interface, while still maintaining loose coupling between the observer and the subject.

## Composite

The composite pattern is a design pattern that allows tree structures to be easily modelled and allow clients treat parts of the hierarchy uniformly. This pattern was used to model the game of UTTT, as it is a tic tac toe game made of up tic tac toe games this pattern lends itself well to the problem. The pattern only has three participants normally, these are:

* The Component: Declares the interface for objects in the composite pattern
* The Leaf: Represents a leaf object in the tree structure (in the case of UTTT this would be a tile)
* The Composite: Stores child components and defines the behaviour for components which have children

A close up of a logo

Description automatically generatedIn this example of the composite design pattern the Leaf class is the tile and the composite class is the TicTacToe class. The component that obscures this implementation from the client is the BoardGame interface which does not reference itself.

## Chain of Responsibility

A screenshot of a cell phone

Description automatically generatedThis design pattern is flexible and was used for two different reasons in this project, that being for behavioural changes and as a creational pattern. Chain of Responsibility allows a problem to be broken down into component parts and each one linked independently. This allows code to avoid long strings of “if else” blocks which can quickly get cumbersome and unmaintainable. The pattern consists of three classes; the chain interface, which defines a singular method used by the client to interact with the chain, the abstract chain class which defines an if statement and a mechanism for linking parts of the chain and the concrete chain class, this class defines the behaviour for part of the chain. This pattern was used to decouple the system that checked if a game was over from the game itself, while also allowing the win checking to be better tested. The chain components in that example were a VerticalWinChecker, HorizontalWinChecker and a DiagonalWinChecker, which checked for a win condition vertically, horizontally and diagonally respectively. This allowed the problem to be broken down into manageable pieces and avoid multiple if else blocks. As a creational patter it is used when objects arrive to the controllers over the internet. As objects arrive as JSON and functions cannot be transferred objects must be created based on the properties of the JSON that is received. Different players have different properties so a Chain of Responsibility is used to decide which player should be created and to create it. This allowed me to easily add new links to the chain whenever a new player was created with little change to other code.

## Dependency Injection

The final pattern that was used in this project is dependency injection. This is a creational pattern that allows the developer to define what is known as a container, in which mappings between interfaces and concrete classes are established. Then when writing code, the constructor is simply given a reference to the interface and at run time the correct concrete class is provided. This allows the developer to create very flexible code as no class has a reference to any other classes’ constructor. Moreover, no class has access to any concrete class instead only the interface which was defined in the container. However, this pattern does not allow for instances of a class to be swapped during runtime and so is not always applicable.

# Design and Implementation

In this section I will cover in greater detail the design of the application, the models used, and the implementations created for this project. I will go over the changes that the project went through and some testing will be touched on.

## Model

The models are the object representation of the game at any one point, it details where players have made moves, what moves are next available and which player is to move.

## Requirements

The requirements of the model were:

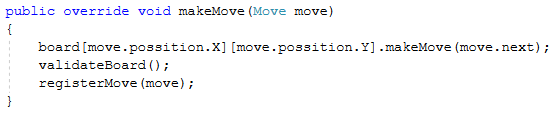
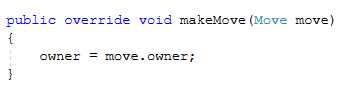
* Store the state of the game
  + The position of moves on the board
  + The types of players in the game
  + The current Player
  + The sub board the player must make their next move in
  + The sub boards that have been won

It must expose methods to the controller that allow moves to be made and to check if the game state is terminal, whether that is a win or a draw. As the View is completely separated from the server and objects are passed in through JSON methods are not transferred. Because of this it is not possible to expose any methods to the view and all interactions with the model must be done through the API.

## Design

The design for the games model was decided early on in development and fit quite well with the behaviour that was needed. The UTTT board would be modelled as a two-dimensional list of objects each of which would have another two-dimensional list of tiles. This model lends itself to the composite design pattern that was used to perform calculations on the board. Moreover, it also made drawing the board on the screen simpler as Angular allows HTML components to be dynamically allocated in the DOM, which means that simply by defining the logic to decide if a tic tac toe board or a tile should be placed in one location would allow Angular to populate the DOM with all the necessary components. AI Players were not implemented on the client side, they ran on the server as to not have the performance of the clients machine impact their performance. To maintain the information of which players where playing a game the players were enumerated and the enumerated value was sent along with the board to the server where the correct player could make its’ move. The classes involved in the model are the TicTacToe class and the Tile class, both of which implement the BoardGame interface but the TicTacToe class also extends the CompositeGame class which is where it derives its’ composite behaviour. The BoardGame interface defines only a handful of methods, the most important of which are the makeMove and the getAvailableMoves. Both these methods take advantage of the composite nature of the TicTacToe class to allow code to be more concise and maintainable. getAvailableMoves in particular takes advantage of the composite design pattern, by treating nested components identically no concession needs to be made for leaf components, the TicTacToe board simply calls to get available moves from which ever sub boards are defined as available and returns that list.

## Implementation

The key methods in the model are the MakeMove and the TicTacToe class. MakeMove takes a Move object, which is also a Composite class, and because of the composite nature of the TicTacToe class, it can simply treat all child classes identically pass the move along for processing. The validateBoard method is used to check the winner of the game as well as perform some simple checks after a move is made, registerMove allows the class to restrict what the next available sub board will be. The Tile class, which is the leaf of the composite pattern, must also implement the makeMove method and this is what allows the TicTacToe class to treat all sub boards identically.

## View

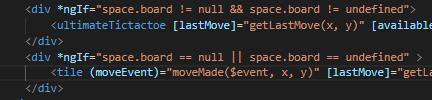
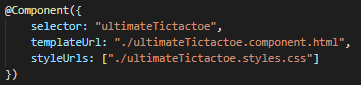
The view is a web app that displays and allows the user to interact with the model. It was made using Angular 5 framework which enables elements to be dynamically allocated to the DOM.

## Requirements

The fundamental objective of the view is to provide the user with a clear and intuitive representation of the underlaying data. The view does not provide the user with any functionality to manipulate the data however as this would break the single responsibility principle. The functionality the view does provide include:

* A Start screen with the functionality to:
  + Ability to selecting players in a game
  + Ability to select the number of games to play
* Login screen which provides functionality to
  + Ability to register as a new user
  + Ability to log in as an existing user
* Game screen with functionality to
  + Allow a player to play a move
  + Clearly display the moves available to the user
  + Display moves previously made
  + Allow the user to differentiate between the two players moves
  + Allow the user to see who one each sub board at a glance

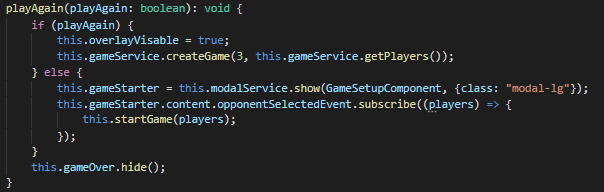
## Design

The view was implemented with Angular Framework. It makes use of the Model-View-ViewModel (MVVM) design pattern. This means that the representation of the model in the view does not necessarily need to completely reflect the model, only the parts which are relevant. It also allows for separation between the logic and the HTML that displays the model to the viewer. Angular allows components to be dynamically be loaded to the dom, this was beneficial as it allowed components to be nested without knowledge of how deep the UTTT game is nested. The “\*ngIf” decorator is part of the angular framework and is the mechanism by which angular dynamically selects components to render in the DOM. Both the “ultimateTictactoe” and the “tile” tags will be resolved to the HTML that is pointed to by the component with the selector decorator that matches the tag. The variables in the square brackets denote what angular define as “Inputs”. These are values that are passed from a parent component to a child component through the DOM. Additionally, the event in the round brackets is what is known as an “Output”. This is the angular implementation of the observable, the function that is made equivalent to the output will be called whenever the emit is called on the event emitter that is tied to this output. The variable “$event” will be whatever variable is emitted. This is part of what makes the MVVM architecture powerful, as the logic that drives the UI can be completely refactored and remade without impacting the way the application looks, as long as the interfaces defined are respected.

## Libraries

In the client a number of libraries were used to additional functionality and save time in development. Besides dependencies the main libraries used in this project were; Bootstrap, angular2-toaster and rxjs.

### Bootstrap

Bootstrap provides an extensive css library which allows developers to quickly and easily create web pages that look clean and professional. The basic version of bootstrap however does not integrate well with angular, as it does not provide a way to interact with certain components such as modals. For this a wrapper library was used, “ngx-bootstrap”, which provides components which can be injected into code and provide extra functionality to the UI. The BsModalService is provided by ngx-bootstrap and is a wrapper for the bootstrap native modal class. This allows the developer to have greater control over how the UI behaves. As seen in (figure) the modal service allows bootstrap components to be interacted with in a concise and intuitive way. The show method simply displays the passed in component as a modal as well as applying the style defined by the user. Moreover, it allows developers to access to components encapsulated in the modal by way of the content variable.

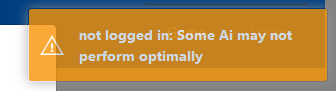
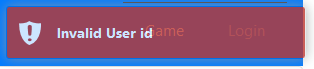
### Ngx-toaster

This library provides services to display toasts. The library was selected over the ngx-bootstrap solution as there were version conflicts with the installed version of bootstrap and an attempt to resolve the conflicts proved to be cumbersome and time consuming. This library does not provide as clean a solution as ngx-bootstrap does as every component that wishes to display a toast must have a “toast-container” tag in its HTML. This in itself is not much of an issue in ways of separation of concerns, where it does become a problem is when errors occur in components that do not have views, such as services. In this case the component that uses that service must recognise the services use of toasts and provide a toast-container for it. This results in high coupling between components in the view.

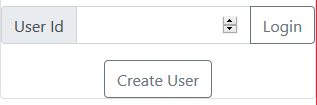
## Modals

Modals are a small component that pop up over the application, these are useful for giving the user a small piece of information or let the user input such as a confirmation box.

## Toast

Toasts are small popups that contain a single piece of information. These do not interrupt the use of the application. These toasts can be different colours to communicate extra information to the user. The major difference between this and a modal is that toasts do not have any interactions with the user. They are used strictly to inform the user of some vital piece of information. The library used for toasts is “angular2-toaster”. This library encapsulates bootstraps native toast classes which allows them to be better integrated with angular, by allowing a toast service to be injected into components that will need to use it.

## Login Screen Design

The loading screen is fairly simple in its design as it does not provide lot of functionality and should very easily afford logging in. It is a simple input with a login button appended and another button which allows users to create new accounts.

## Game View Design

A screenshot of a cell phone

Description automatically generatedIt was intended for the screen to be split into three sections when a game was in play. The outer most sections would be dedicated to each of the players, displaying information on what colour each player was, their name and other stats. Due to time constraints this design of the UI was never realised, and both end components were not implemented.

## Navigation

The navigation uses a simple nav bar places across the top of the screen. There are two pages contained in the app, the login page and the game page. It was intended for the login functionality to be encapsulated in a modal, however it was decided that a better separation of functions between this and other parts of the application made navigation easier.

## Controllers

The controllers are split across the server and the client. They provide the user with a mechanism by which they can interact with and mutate the model.

### Client

On the client side the controllers provide a way for the user to interact with the model through the view.

#### Requirements

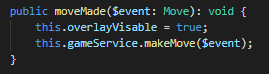
The requirement on the client side are simple and surround the users ability to do two things, log in and play the game. For logging in the controller must:

* Communicate to the server the users actions
  + Login
    - Create new user
    - Login as existing user
  + Game play
    - Communicate with server which players to construct
    - Communicate with server which move to make

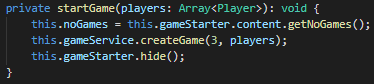
#### Design

The client side has only one controller. It is implemented as an Angular component as though it does not provide any view of its own it wraps the UTTT view and so must have HTML associated with it. It uses a system of modals and events to communicate with the view and underlying services. Moreover, it does not in anyway mutate the model it simply provides the mechanism for the user to communicate the actions they would like to take to the server, which in turn mutates the model.

#### Implementation

The client side controller consists of only two classes, the “GameComponent” class and the “UserComponent” class. The GameComponent class interacts with the UTTT Component, which provides the view for the game, and responds to click events that propagate out from the view. Observers for click events are registered in the HTML, when a click is registered on a tile it emits a move event up to the parent UTTT component. The UTTT component will then recursively emit move events to their parents until it reaches the GameComponent where it is then processed. Processing of moves simply involves sending the relevant information to the GameService, which will then send it to the server in the correct format. Also, the GameComponent is responsible for communicating which players the user wishes to play. Again, to do this the component simply gets the relevant information from the view and passes it to the appropriate service to send it to the server. This separation of concerns between the controller and the server is vital for several reasons, for one the endpoints on the server may change, secondly the server may be replaced with some different system, possibly client side. These changes would be cumbersome if the GameComponent were to directly interface with the server. The other controller on the client side is the UserComponent, it is responsible for communicating login details and requests to the UserService, it does this in the same way that the GameComponent does.

#### Maintenance

In the first implementation of the game players were only given one option when setting up a game, this was who the opponent was going to be. As development progressed new needs arose, those being, two AI should be able to play each other, the human may not always want to play first, for experiments multiple games between the same opponents should be queued up. To accommodate these changes the GameSetupComponent, a component displayed by the GameComponent when the game window is first visited, was extended to allow players to select both players and a new input field was added to allow some number of games to be queued. That is why, in the above figure players is passed as an argument, as an array of players was always constructed by the GameSetupComponent, but the number of games is retrieved in the method call.

## Server

In the server controllers behave slightly differently, they are responsible for retrieving web requests and calling services to do work on the information they receive.

### Requirements

The requirements for a server-side controller are minimal. They must:

* Expose an endpoint of the server
* Pass the appropriate information form the web to the service
* Return the request in the appropriate format

### Design

To abstract some of the basic web interaction from more specialised controllers, all controllers extend a class called BaseController which in turn extends Controller. The Controller class is provided by the Asp.Net framework which provides methods for receiving and returning HTTP requests. There are only two classes which extend the BaseController these are, The GameController and the UserController. Both of these controllers expose various endpoints that allow a client to request actions be made to a model or some value be updated in a database. The Asp.Net framework also provides some decorators which are used to expose methods to the web. Furthermore, there are decorators which tell the framework that certain arguments to a method will be provided in the HTTP request.

### Implementation

The BaseController has only one method, ExecuteApiAction, this method must be called by any controller that wishes to respond to a http request as it is responsible for returning the correct HTTP status code. The ExecuteApiAction method takes an anonymous function as an argument, this function will be called by the BaseController, if no error occur the result of the call will be wrapped in an IActionResult object, which is provided by Asp.Net core. If an error does occur, then the BaseController handles the exception and returns the correct HTTP error code. The decision of an anonymous function was made as it helps the BaseController be more generic. The ExecuteApiAction method can return an IActionResult of any type. It allows all inheriting classes to define any methods they wish as long as they return an ApiResult.

### Maintenance

When a request arrives from the web it is in JSON. In the initial implementation of the GameController the object passed into the rateMove method had a variable called game, which is a representation of the board, and was typed as the interface BoardGame. This became an issue when the client was able to send requests to the server, as the interface is implemented by multiple classes so the JSON deserialiser provided by Asp.Net could not decide which class to use and through an exception. In an attempt to correct this issue, the JSON deserialiser was subclassed to add functionality that would determine the correct subclass to return. This also proved problematic as the new deserialisers could not have classes injected into them which were needed by the concrete classes of BoardGame. Another solution was implemented which saw the game variable’s type being changed from a BoardGame to a JsonObject and a new class was implemented to properly construct the correct BoardGame from the JSON. This solution, though requiring more code, allowed the client to remain unchanged while also allowing classes to be injected into the new BoardGameConstructor class which allowed loose coupling between any service needed by a BoardGame and the constructor of the BoardGame. 

# Services

Services exist on both the client and server side and both intend to make changes to the model. As discussed previously in this chapter client-side services do this by sending requests to the server, the server-side services have access to the actual implementation of the model though and so can call methods to mutate the model.

## Client

On the client-side services exist through the duration of the tab that the app is open in, unlike other components. These services serve as the true version of the model that is passed from the server and all other components use a system of events to communicate with them

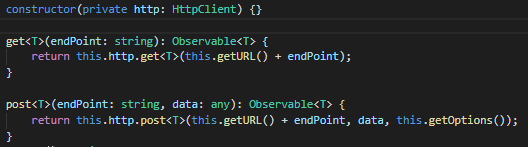
### Requirements

The requirements of clients side services are simple, they must:

* Maintain the model
* Provide methods to mutate the model
* Provide events for when the model is updated

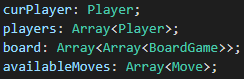
The difference between this and a controller is this does not interact with any event from the view, it does not interpret click events or menu choices. Instead it provides the controllers with a way to communicate with the server, which in turn will mutate the model.

### Design

To decouple the services of the client from the web a lower level of service was created called the ApiService. This service wraps the HttpClient that is provided by Angular.

Doing this isolates any change that Angular might implement on the HttpClient so only this class would need to be updated. Moreover, it allows reusability of this code as multiple services make HTTP requests. The classes which use this service are the GameService and the UserService, both of which have the ApiService injected into them.

### Implementation

The services on the client-side consist of two classes, the GameService and the UserService. As services in Angular exist for the duration of the application and are singletons by design they hold information that be requested for by controllers. The GameService holds all the information that the controllers would need to process user actions correctly, such as the state of the board, the available moves and the current player. The UserService is mostly the same except it stores information regarding the current logged in user. These classes are not meant to be overly large, as they are not meant to have much functionality in them. The UserService for example only has thirty nine lines of code, as it is mainly designed to forward information to the server’s controllers.

## Server

The services on the server are where the vast majority of the games logic is processed. They exist in the business layer of the servers layered model along with the game models.

### Requirements

The services are the where most of the code in this project exists. They provide the functionality that interacts with the model and the logic that formats the HTTP responses. These classes have the biggest requirement set of any in the project. They are:

* Let both players make moves
* Decide who the winner is
* Have the moves made by both AI and human players rated
* Have a mechanism for persisting the move data
* Flatten the response time for AI players
* Create the DTO that will be sent to the client
* Provide a mechanism for creating new users
* Provide a mechanism for retrieving existing users
* Provide a mechanism for updating existing users
* Provide a service for creating random numbers
* Provide a system for ranking moves
* Provide a mechanism for creating Model objects from the JSON received from the web

### Design

The services all exist in a simple inheritance with only an interface and the concrete class. This is because Asp.Net frameworks dependency injection requires injected classes implement an interface and that interface is injected. The concrete class will be created by the runtime. Doing this decouples the implementation of the service from every class that uses it, as nowhere in the code that was developed is a new instance of a service created. This allows future developers to create new services which fulfil the promise of the interface. They can then change the mapping in the Asp.Net container which will then supply the new class. As mentioned previously the services exist in the business layer of the server, this means that they are used by the Api layer to mutate the models passed in from the web, while also having access the data access layer. Because of this these services are the largest classes in this project. There are multiple services in this project which exist only because TDD demands they do. Services such as the RandomService which wraps the Random class. This class is well tested, which would be substantially harder to do if every class which required random numbers simply had an instance of the random class.

### Implementation

The main service class in the server are the GameService and the UserService, mirroring the client side services. These services though are far more substantial as they provide all the mechanisms needed to process the requests that come from the client. They are injected into the Controllers and are called to process all the requests that arrive from the web. The GameService interface only exposes two methods, processMove and rateMove. These methods are used by the controller to have the service process or rate a move that arrived from the web. The UserService is mostly the same providing methods for creating, updating and getting users.

### Maintenance

In the original implementation of the GameService the AI player was given the board which it then made its move on and returned. This system was found to be an issue for two reasons; AI players developed in the future could simply cheat and mutate the board however they wished as it was passed by reference, AI response times depended on how much processing they did before making a move, the RandomAI responded very quickly while the MCTSAI responded after a couple seconds and lastly the amount of time taken for an AI move was effectively doubled as to rate a move the MCTS algorithm was used, so AI which already performed this action would look to the user to take twice as long than AI which did not. To rectify this the MCTS algorithm was called before the AI were. They were then passed the board they were given and the tree built by the MCTS. This flattened the response time as the amount of time AI took to process the nodes they were given is minimal, while also only needing the MCTS algorithm to be ran once on the game. Furthermore, it meant that when rating AI moves the scores that the AI saw are identical to whatever will be stored. This means that the representation of the AI in the database accurately reflects how the AI intends to play.

# References

Bredyn McCombs, C. L., 2018. *Ultimate Tic Tac Toe Search Strategies.* St George, Dixie State Regional Symposium.

Chanel, G. &. R. C. &. B. M. &. P. T., 2008. *Boredom, Engagement and Anxiety as Indicators for Adaptation to Difficulty in Games.* New York: ACM New York, NY, USA ©2008.

Donlan, C., 2014. *eurogamer.* [Online]   
Available at: https://www.eurogamer.net/articles/2014-05-22-middle-earth-shadow-of-mordor-promises-plenty-of-strategising-and-accidental-chaos  
[Accessed 5 March 2019].

Hunicke, R. C. V., 2004. *AI for dynamic difficulty adjustment in games.* 2nd ed. s.l.:s.n.

Kocsis, L. & Szepesvári, C., 2006. Bandit based Monte-Carlo Planning. In: J. Fürnkranz, T. Scheffer & M. Spiliopoulou, eds. *Machine Learning: ECML 2006.* Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 282-293.

Lidén, L., 2004. Artificial Stupidity: The Art ofIntentional Mistakes. In: *AI Game Programming Wisdom 2.* s.l.:Charles River Media, pp. 41-48.

Mirna Paula Silva, V. d. N. S. L. C., 2015. *Dynamic Difficulty Adjustment through an Adaptive AI.* Piaui, SBGames.

Spronck, P. P. M. S.-K. I. e. a., 2006. Adaptive game AI with dynamic scripting. *Machine Learning,* p. 63\*217.

Spronck, P., Sprinkhuizen-kuyper, I. & Postma, E., 2004. *Difficulty scaling of game AI.* Ghent, GAME-ON.

Tan, C. H., Tan, K. C. & Tay, A., 2011. Dynamic Game Difficulty Scaling Using Adaptive Behavior-Based AI. *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES,* 3(4), pp. 289-301.

Wei-Yuan Hsu, C.-L. K. C.-H. H. I.-C. W., 2019. Solving 7,7,5-game and 8,8,5-game. *ICGA Journal,* 40(3), pp. 246-257.