Table of contents

[Table of contents 1](#_Toc474039093)

[Analysis 2](#_Toc474039094)

[Description 2](#_Toc474039095)

[Target Audience/Stakeholders 2](#_Toc474039096)

[Why a computer is suitable for the task 2](#_Toc474039097)

[Research 3](#_Toc474039098)

[Features and limitations 4](#_Toc474039099)

[Requirements 4](#_Toc474039100)

[Success Criteria 5](#_Toc474039101)

[Design 6](#_Toc474039102)

[Decomposition of the problem 6](#_Toc474039103)

[Proposed structure of the program 6](#_Toc474039104)

[Algorithms 6](#_Toc474039105)

[Usability 6](#_Toc474039106)

[Key variables and data structures 6](#_Toc474039107)

[Test Data for development 6](#_Toc474039108)

[Test Data for beta testing 6](#_Toc474039109)

[Development 6](#_Toc474039110)

[Iterations of development 6](#_Toc474039111)

[Prototypes 6](#_Toc474039112)

[Evidence of modular code 6](#_Toc474039113)

[Evidence of validation 6](#_Toc474039114)

[Review 6](#_Toc474039115)

[Evaluation 7](#_Toc474039116)

[Testing 7](#_Toc474039117)

[Testing of usability features 7](#_Toc474039118)

[Overall evaluation 7](#_Toc474039119)

[Future Maintenance 7](#_Toc474039120)

Analysis

# Description

The aim of this project is to create a virtual version of the game Tic-Tac-Toe with an AI that learns how to play using data from its previous matches.

Tic-tac-toe is a 2-player game played on a 3x3 grid. One player will play as the ‘X’ piece, while the other player is the ‘O’ piece. Each player takes a turn to place their piece on the grid. The aim of the game is for one of the players to get 3 of their pieces next to each other in a row; which can be done vertically, horizontally, or diagonally.

The AI will be used to substitute one of the required players, and the aim is for this AI to learn from its previous experiences to try to figure out what is a likely way of it being able to win.

The AI should look at its past history of matches, and use this data to figure out the best way to continue during a match. However, the AI won’t always have enough data to use (it learns while it plays games, so if it hasn’t played many games, there’s not a lot of data) so it should also have a way to deal with such a situation, such as doing a move completely at random. The AI could then use this new data in future matches, giving it more variety in how it can play.

# Target Audience/Stakeholders

There is no specific audience due to the simple/casual nature of the game, and the fact that most people seem to know what tic-tac-toe is means it is very accessible to a large number of people. This means that the stakeholders will most likely be anyone who is aged 6 years or older.

The game should provide instructions however, in the case someone does not know how to play tic-tac-toe.

There is no need for technical expertise when playing the game, so it is accessible to most people.

# Why a computer is suitable for the task

Computers are very fast at performing calculations, and the only errors they make are generally due to human errors (coding mistakes, or errors intentionally put in, for example). For a game as simple as tic-tac-toe, a computer is more than capable of calculating what it should do in a reasonable amount of time, and can possibly be almost impossible to beat.

As an example, when two human players are playing against each other, Player 1 may be one move away from winning the match with Player 2 not noticing they can stop them; however, when a human is against an AI, the AI can be made so it will always block the human from winning, if there’s a chance to. This is due to the point made earlier, the only errors a computer can make are usually due to human errors, so if a human tells the computer to always (or to never/only sometimes) block the other player, then it will do so without fail (assuming there’s no errors in the code).

As another example, for a simple game like tic-tac-toe, the computer may be able to plan ahead of time and think of the most optimal route to take, similar to a human. The difference is that a computer can analyse the possible paths it can take significantly faster than a human, and a computer will be able to ‘remember’ them all perfectly, whereas a human might forget something or make mistakes in their logic.

Computers are also capable of storing tremendous amounts of data. For this project’s use case, this is good as it allows the game to be able to store data of hundreds or thousands of unique tic-tac-toe matches for use with the AI.

# Research

While researching on what algorithms I might use when writing the AI for the game, I came upon the Minimax [1] algorithm.

The Minimax algorithm as defined on Wikipedia is…

A decision rule used in decision theory, game theory, statistics, and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario.

After further research, I came upon a website [2] where a programmer describes how they used the Minimax algorithm with tic-tac-toe. The general idea is, they calculated every possible route the AI could take, and awarded points to each route which signifies how much of a loss (-10 points) the AI would suffer if it went down this path, and how much of a gain (+10 points) it would get. The path with the highest amount of points would be chosen. (The website also talks about other tweaks needed to make it work well with tic-tac-toe).

The issue with this algorithm is, it creates an unbeatable AI, which is not fun for the human to fight against (nor does it seem terribly interesting to code). The upside is, this algorithm is a perfect example of how a computer is suitable for playing tic-tac-toe, and can be better at it than humans.

The idea of weighing which path is most likely to win/lose was interesting to me, and during a session with my computer science tutor, he was discussing about possibly using machine learning, where the computer stores data of past games and then uses that data to determine which moves have led it to a win in the past.

The advantage of the AI using past data, is that instead of calculating the best moves to make on the spot, is that it can attempt to ‘learn’ the best way to win which I see as an acceptable compromise between ‘impossible to beat’ and ‘impossible to lose against’. At the start, when the AI lacks data, it should be pretty easy to beat; but as time goes on the AI will ‘harden’ and gradually get more data meaning it will be able to perform better than when it started.

Similar to how the minimax algorithm would create a tree of moves to analyse, my AI could store the data of its past games in a tree. For example, it may be formatted like:

-> “X is placed in the top-middle slot” -> “O is placed in the bottom-right slot”

“empty grid”  
 -> “X is placed in the top-left slot” -> “O is placed in the bottom-middle slot” etc.

[1] <https://en.wikipedia.org/wiki/Minimax>

[2] <http://neverstopbuilding.com/minimax>

# Features and limitations

The game must provide a GUI. This GUI must display the 3x3 grid which shows the current up-to-date state of the match. The GUI must at the very least allow the player to play multiple matches without having to restart the game. Finally, The GUI must allow the player to interact with the 3x3 grid, following the rules of how you’re allowed to play pieces in tic-tac-toe.

The game should provide a message box that details how to play tic-tac-toe. Ideally this should be shown when the game is opened for the first time, and whenever the user presses some sort of “help” button.

The game must not allow the user to perform an invalid move, and should simply wait for the user to input another move if this happens.

The game will require having to store data on previous games, and being able to load this data when it is opened. The game should use a binary format, as it allows for more compact file sizes, but there is a trade-off of a human (me in particular) being able to easily read and debug the data as would be possible using a text format.

Multiplayer, while a desirable feature, is not the focus point of the project; that would be the AI. Therefore, multiplayer capabilities won’t be added to the game until sometime in the far future, if at all.

The game will require an AI for a human to fight against. This AI should make use of its past matches with humans to aid it with choosing what moves to make during a match. The AI should not be unbeatable, as it would be unfun to fight against.

An animated GUI that comes with sound effects is quite a bit of effort with very little worth considering how simple a game tic-tac-toe is, so I have decided to go with a very simple, soundless GUI.

Due to the reason that the AI learns as it plays, it will start off being incredibly easy to beat, but over time it will become more challenging. Theoretically, it should only end up either winning or tying after a while (something I wish to avoid, due to it being unfun); however, this will require over two-hundred thousand unique games to have been played [1]. Because of this, it may take some time for the AI to actual be a considerable threat, and is likely to be very easy to beat for longer than I’d like it to be.

[1] <https://www.jesperjuul.net/ludologist/255168-ways-of-playing-tic-tac-toe>

# Requirements

OS: Windows Vista SP2 (with .Net 4.5 installed) or later (Any Windows OS that can run WPF [1])

CPU: 2GHz or faster.

GPU: Integrated graphics card, or better.

The project will be coded in C#. It will use the WPF framework for the GUI. This means that a Windows OS must be used, as WPF (and C#, for the most part) is only supported on Windows.

The project will be built and tested against .Net 4.5, so .Net 4.5 must be installed on the computer. The project *might* work with older versions of .Net, but it is not guaranteed. .Net 4.5 comes preinstalled with Windows 8 and later versions of the Windows operating system.

There is no reason in particular as to why .Net 4.5 was chosen, so it is most likely possible to compile the project for an older version.

There are no special graphical requirements for the project, so any GPU that can comfortably run Windows is all that is needed, graphics wise. While any CPU will also work fine as well, a CPU that is *too* slow could make the AI begin to take seconds before it performs the move, possibly making the game feel bad to play.

[1] <https://en.wikipedia.org/wiki/Windows_Presentation_Foundation>

# Success Criteria

To be deemed a success, the game must provide:

* A user-friendly, responsive GUI that provides: the 3x3 grid with an up-to-date view of the game board’s state; text informing the player which piece they’re playing as; text that displays whether it is the player’s or AI’s turn, and it must allow the player to place their piece via the 3x3 grid.
* An AI that is not impossible to win against, and is capable of analysing the data from its past matches to determine which move it should take.
* The game must not crash unexpectedly, and in the event something goes wrong, it must simply show the user an error box saying something’s gone wrong.
* The game must be stable and free of any major bugs (for example, if the GUI suddenly stopped functioning, this is a major bug and should not happen). Certain features of the game, and small parts of the code can and should be tested. The preferred method of testing is unit testing, where a small piece of code is written to test a very specific part of the code. Features of the game that are tricky to test via code (such as how the GUI functions) should be manually tested and documented.

Design

# Decomposition of the problem

The problem as a whole for the AI can be described as ‘An AI using data from past matches to determine which moves will most likely result in a win.’

## Problem 1.1 – How is the ‘data from past matches’ stored?

The AI requires a way to store and use data of multiple tic-tac-toe matches.

A tree would be a suitable data structure to use due to the nature of tic-tac-toe. For example, a node in the tree might describe the following state of a tic-tac-toe board, after the AI (for example) sets their piece at the top-left slot:

|  |  |  |
| --- | --- | --- |
| **X** |  |  |
|  |  |  |
|  |  |  |

The player (again, as an example) could then place their piece in any of the other empty slots:

|  |  |  |
| --- | --- | --- |
| **X** |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| X |  |  |
|  | **O** |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| X | **O** |  |
|  |  |  |
|  |  |  |

As shown in the diagram, the bottom-left node represents when the player places their piece in the middle slot, **after** the AI places its piece at the top-left slot; while the bottom-right node represents when the player places their piece in the top-middle slot, also **after** the AI places its piece at the top-left slot.

As the diagram demonstrates, a tree would be a very natural data structure to represent data from numerous tic-tac-toe matches, as it would support being able to store data about every possible match of tic-tac-toe, and stores it in a logical manner.

“AI places X in the top-left slot” -> “Player places O in the top-middle slot” -> etc. is very easy to represent in a tree (as long as the nodes can have any number of children).

Here is an example of a slightly more fleshed out tree to further demonstrate the viability of using a tree for this data. The root of the tree is simply an empty board. The move that each node in the tree represents is bolded:

*[Figure: Example tree]*

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |  |
|  | |  |  |  | | --- | --- | --- | | **X** |  |  | |  |  |  | |  |  |  | |  | |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  | **X** |  | |
| |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **O** | |  |  |  | |  | |  |  |  | | --- | --- | --- | | X |  |  | |  |  |  | |  | **O** |  | | |  |  |  | | --- | --- | --- | |  |  |  | |  | **O** |  | |  | X |  | |
|  |  | |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **X** | |  | O |  | | |  |  |  | | --- | --- | --- | |  |  |  | |  | O |  | | **X** | X |  | |

## Problem 1.2 – What data is actually stored?

Each node in the tree needs to store data that allows the AI to determine whether it is likely to win or not.

My proposed solution would be for a node to store the following:

* The state of the game board, representing a single move (See Problem 1.1).
* How many times the move the node represents has led to the AI winning.
* How many times the move has led to the AI losing.
* The index of which slot (range of [0..8] inclusive) on the game board the move was performed on. For example, the index of 0 means that the move the node represents placed its piece in the top-left slot, whereas an index of 8 means that the piece was placed in the bottom-right slot.

A notable thing to point out is that, while it **is** possible to determine the index of where a piece was placed by looking at the previous node’s board state, and comparing it with the current node’s board state, it will be more simple (and more importantly, less buggy/more stable) if each node simply stored the index.

Storing how many times a move has caused the AI to win/lose allows the AI to calculate the win percentage of a node – what percentage of games it has led to the AI winning. The formula for calculating this is where ‘w’ is the number of wins, and ‘l’ is the number of losses. This percentage can be used by the AI’s algorithm to determine what move to perform.

Storing the index of where the piece was placed is done so the AI can replicate the move during a match. For example, if it’s picks a node where the index is 2 (the top-right corner), then the AI will know that it should place its piece at slot 2 to replicate the move that the node represents.

## Problem 1.3 – How is the state of the board stored?

Now that I have specified how the data will be stored (in a tree), and what data the nodes store, I now need to determine what kind of ‘format’ the data in a node is stored in.

For the win counter, loss counter, and slot index stored in a node, it is quite clear that they are numbers. However, nodes must also store the state of the game board which so far, have not been provided a ‘computer-friendly’ way to be represented.

The solution is pretty simple, a 9-character string is stored with the node, where the 0th character represents the 0th slot of the game board, the 1st character represents the 1st slot, etc. I refer to this as a *hash* of a game board.

For example, take the following game board:

|  |  |  |
| --- | --- | --- |
| X |  |  |
|  | O |  |
|  | O | X |

The hash of it would be “X…O..OX”, where an ‘X’ represents the X piece, a ‘.’ represents an empty space, and an ‘O’ represents the O piece.

## Problem 1.4 – A hash of a board is only valid if the AI only plays the same piece.

This solution has a slight flaw however, if the AI was **always** the X piece, and the player was **always** the O piece, then this solution would be fine, but if the AI were to suddenly become the O piece, and the player suddenly became the X piece, then the data would no longer be valid because, while the AI and player have changed which piece they’ve used, the hash itself doesn’t reflect these changes.

This problem has an easy fix; instead of storing ‘X’ to represent the X piece, and ‘O’ to represent the O piece in a hash, we instead store ‘M’ to represent the AI’s piece (‘M’ stands for ‘Mine’), and ‘O’ to represent the player’s piece (‘O’ stands for ‘Other player’).

As an example of this new idea, take the previous game board; If the AI is X and the player is O, then the hash would now become “M…O..OM”. Now, if the AI was O, and the player was X, then the hash would become “O…M..MO”.

The two tables below demonstrate what the hash “O…M..MO” would look like if the AI was playing as X (represented by the left table), and if the AI was playing as O (represented by the right table).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  | | --- | --- | --- | | O |  |  | |  | X |  | |  | X | O | | |  |  |  | | --- | --- | --- | | X |  |  | |  | O |  | |  | O | X | |  |

This means that a node with a board hash of “O…M..MO” would be useable regardless of if the AI was playing as the X piece or the O piece, whereas the old idea (‘X’ for X, ‘O’ for O) would make the nodes incompatible depending on which piece the AI plays as.

When the hash needs to be converted back into a board state, then ‘M’ can simply be replaced with what piece the AI is using, and ‘O’ is replaced with whatever piece the player is using.

It’s worth noting that, even if it’s unlikely for the AI to be able to change which piece it uses in my project, I still find it important that the data is reusable (there is no difference between which piece the AI uses, so the data should be the same for whether it’s playing as X or O), which is why I came to this solution. It will likely require little effort to implement while providing a rather large bonus, making it worthwhile.

## Problem 2.1 – How does the AI use this data to decide a move?

At a high-level, all the AI needs to do is go over every path in a tree of moves it has created from its past matches, and select the path that has the highest win rate.

Here is the example tree displayed in Problem 1.1, but with the extra data explained in Problem 1.2 (Excluding the hash, for readability). ‘W’ means ‘wins’, ‘L’ means ‘losses’, and ‘I’ mean ‘index’. The AI is ‘X’, the player is ‘O’. This tree will be used in later examples.

The root node has the value ‘0’ for the wins, losses, and index. This is because the root does not represent an actual move, so should not have these values modified.

*[Figure: Detailed Example Tree]*

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | [Root]  W:0 L:0 I:0   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |  |
|  | W:8 L:4 I:1   |  |  |  | | --- | --- | --- | | **X** |  |  | |  |  |  | |  |  |  | |  | W:3 L:7 I:7   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  | **X** |  | |
| W:1 L:1 I:5   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **O** | |  |  |  | |  | W:7 L:3 I:7   |  |  |  | | --- | --- | --- | | X |  |  | |  |  |  | |  | **O** |  | | W:3 L:7 I:4   |  |  |  | | --- | --- | --- | |  |  |  | |  | **O** |  | |  | X |  | |
|  |  | W:7 L:3 I:5   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **X** | |  | O |  | | W:3 L:7 I:6   |  |  |  | | --- | --- | --- | |  |  |  | |  | O |  | | **X** | X |  | |

## Problem 2.2 – How does the AI determine a ‘path’ (also, how is the win percent calculated)?

A path is list of nodes that make up a tic-tac-toe match. I will provide a high-level example of the paths that exist in the example tree shown in Problem 2.1, then describe the algorithm that can be used to determine all the paths.

The diagrams below show all of the different possible paths the AI could take, where the nodes coloured in blue are the nodes making up the path. The win percentage (depicted as ‘W%’) of each path is calculated (using the formula in Problem 1.2), and the average of the win percentages is used to determine the overall likelihood of the path winning. The win percentages are calculated in this section so later sections of the document can reference to it.

To find the average win percent, add up all of the win percentages of the nodes that make up the path, then divide the number by how many nodes are in the path. If the percentages are used in decimal form (60% being 0.6, 25% being 0.25, etc.) then multiply the result by 100 to get a more ‘natural’ percentage (e.g. 0.25 x 100 = 25%).

*[Figure: Path #1]*

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | [Root]  W:0 L:0 I:0 W%:0   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |  |
|  | W:8 L:4 I:1   |  |  |  | | --- | --- | --- | | **X** |  |  | |  |  |  | |  |  |  | |  | W:3 L:7 I:7  W%: 30% (0.3)   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  | **X** |  | |
| W:1 L:1 I:5   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **O** | |  |  |  | |  | W:7 L:3 I:7   |  |  |  | | --- | --- | --- | | X |  |  | |  |  |  | |  | **O** |  | | W:3 L:7 I:4  W%: 30% (0.3)   |  |  |  | | --- | --- | --- | |  |  |  | |  | **O** |  | |  | X |  | |
|  |  | W:7 L:3 I:5   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **X** | |  | O |  | | W:3 L:7 I:6  W%: 30% (0.3)   |  |  |  | | --- | --- | --- | |  |  |  | |  | O |  | | **X** | X |  |   Overall W%:  0.3 x 100 = 30% |

*[Figure: Path #2]*

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | [Root]  W:0 L:0 I:0   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |  |
|  | W:8 L:4 I:0  W%: 67% (0.67)   |  |  |  | | --- | --- | --- | | **X** |  |  | |  |  |  | |  |  |  | |  | W:3 L:7 I:7   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  | **X** |  | |
| W:1 L:1 I:5   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **O** | |  |  |  | |  | W:7 L:3 I:7  W%: 70% (0.7)   |  |  |  | | --- | --- | --- | | X |  |  | |  |  |  | |  | **O** |  | | W:3 L:7 I:4   |  |  |  | | --- | --- | --- | |  |  |  | |  | **O** |  | |  | X |  | |
|  |  | W:7 L:3 I:5  W%: 70% (0.7)   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **X** | |  | O |  |   = 69% | W:3 L:7 I:6   |  |  |  | | --- | --- | --- | |  |  |  | |  | O |  | | **X** | X |  | |

*[Figure: Path #3]*

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | [Root]  W:0 L:0 I:0   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |  |
|  | W:8 L:4 I:0  W%: 67% (0.67)   |  |  |  | | --- | --- | --- | | **X** |  |  | |  |  |  | |  |  |  | |  | W:3 L:7 I:7   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  | **X** |  | |
| W:1 L:1 I:5  W%: 50% (0.5)   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **O** | |  |  |  |   = 58.5% |  | W:7 L:3 I:7   |  |  |  | | --- | --- | --- | | X |  |  | |  |  |  | |  | **O** |  | | W:3 L:7 I:4   |  |  |  | | --- | --- | --- | |  |  |  | |  | **O** |  | |  | X |  | |
|  |  | W:7 L:3 I:5   |  |  |  | | --- | --- | --- | | X |  |  | |  |  | **X** | |  | O |  | | W:3 L:7 I:6   |  |  |  | | --- | --- | --- | |  |  |  | |  | O |  | | **X** | X |  | |

As promised earlier, I will provide the algorithm to find the paths in a tree. This algorithm can be used to simplify other algorithms, and will be referred to as the ‘WalkPaths’ algorithm in this document. It may also be worth noting that WalkPaths is recursive, so the algorithm uses itself in a ‘divide-and-conquer’ fashion. This is an O(n) algorithm, as the amount of time it takes is related to how many nodes there are in the tree.

The algorithm at a high level:

1. Starting with the root, get the next child node, and add it to a list.
2. If the child node also has children, perform step 1 on it.
3. It the child node does not have children, then it is the end of a path, so perform an *Action*(defined below) on it.

Variables:

* *Node* = The node passed to the first step in the algorithm. The first node given is defined as the ‘Root’ node.
* *Path* = An array of nodes.
* *Action* = Another algorithm that is performed on *Path*.

WalkPaths is defined as:

1. If *Node* isn’t the root node, add it to the end of *Path*. \*
2. If *Node* doesn’t contain any children nodes, then it’s the end of a path, so,
   1. Perform *Action* on *Path*.
   2. Go to Step 4.
3. Otherwise, get the *Node*’s next child.
   1. Perform WalkPaths, where *Node* is now the child, *Path* is the same *Path* currently being used, and *Action* is the same (come back to this step afterwards).
   2. Remove the last node in the *Path* (this is the child node we just got, and would’ve been added by step 1).
   3. Perform Step 3 again if the *Node* still has children to go over.
4. If *Node* is currently the root node, then the algorithm has gone through all paths, so end the algorithm.
5. Otherwise, set *Node* back to its previous value.

\* *The Root node doesn’t represent a move, so shouldn’t be included in the Path.*

I recommend using the WalkPaths algorithm on the example tree (*Action* can simply be defined as ‘do nothing’ in this case) if it is unclear how it works.

The *Action* is task-specific, so should default to ‘do nothing’ if no specific action is defined. Later sections will specify an *Action* to demonstrate how generic and reusable this algorithm is.

## Problem 2.3 – How does the AI decide which path has the highest win percentage?

Now that we have defined the ‘WalkPaths’ algorithm, and it is made in such a way that we can attach an ‘Action’ for it to perform on every path, it makes other algorithms very simple - such as the algorithm the AI will use to determine which path has the highest win percentage.

This is an O(n) algorithm, as the amount of time it takes is in proportion to how many nodes are in the *Path*s given to it. When used together with the WalkPaths algorithm, they become an O(2n) algorithm, as they go over every node in the tree twice.

The algorithm at a high level is simply to:

1. Go over every path in the tree.
2. Pick the path with the highest average win percentage.

To do this, the AI will perform the WalkPaths algorithm, where ‘Action’ is defined as:

Variables:

* *BestPath* = The path (array of nodes) that has the highest chance of winning.
* *BestPercentage =* The win percentage of *BestPath*.
* *Path* = The path given to this algorithm by the WalkPaths algorithm.
* *Percentage* = The overall percentage of *Path*. This value is the percentage as a decimal (25% being 0.25 as a decimal, for example).

Steps:

1. Get the next node in *Path*.
2. Calculate the win percentage (described earlier in the document) and add it to *Percentage.*
3. If there are still nodes in *Path*, go to step 1.
4. If *Percentage* is greater than *BestPercentage.*
   1. Set *BestPercentage* to *Percentage*.
   2. Set *BestPath* as a copy of *Path*.

At the end of the WalkPaths algorithm, the *BestPath* variable will hold the path with the highest average win percentage. As before, I recommend to try it out on the example tree (as the example diagrams show, the result should be the same as ‘Path #2’).

This algorithm also demonstrates the reusability of the WalkPaths algorithm, as WalkPaths handles finding every path in a tree, while the ‘Action’ algorithm can focus purely on the task its designed for.

I generally refer to this algorithm as the “StatisticallyBest” algorithm, and will use this name to reference it later on.

## Problem 3.1 – What does the AI do during a match?

Now that we know how the data is stored, and have some algorithms to use on this data, we now need to figure out what the AI should be doing during a match.

The AI first of all, needs to keep a tree full of nodes from past matches; this is called the ‘Global’ tree. This Global tree, because of it holding all of the AI’s past matches, is the prime candidate for the data the AI will be using to decide it’s move.

The AI should also keep another tree which represents the current match it is playing, and is called the ‘Local’ tree. This tree is needed so the AI can ‘mirror’ itself from the Local tree into the Global tree.

So overall:

* After every move (whether it’s by the player or the AI) create a node for the move, and add it into the local tree.
* During the AI’s turn, it should use the data in its Global tree to figure out what move to make.
* At the end of a match, the AI should bump the ‘win’ and ‘loss’ counter of each node in its Local tree, and then merge it into the Global tree.
* Either after a match, or when the game is closed, the AI should save its Global tree into a file.
* At the start of a match, the AI should load its Global tree from a file (if one exists), and empty its local tree.

## Problem 3.2 – How does the AI figure out what move to make?

It’s simple to say to myself ‘Just use the StatisticallyBest algorithm and that’s that’ but unfortunately there are some problems that must be solved.

Imagine the AI is in a match, and its Global tree is the example tree shown earlier (the ‘Detailed Example Tree’ in Problem 2.1). Now, if the WalkPaths algorithm is used, where its *Action* is the StatisticallyBest algorithm, then the AI would choose to go down ‘Path #2’ (from Problem 2.2).

So, the AI has chosen Path #2, and starts off the match putting its piece in slot 0 (since the first node in the path is ‘AI puts piece in slot 0’, meaning the AI will mimic it). Now, there are a few things that can happen depending on what the player decides to do.

If the player places their piece in slot 7 as the second move (the second node in Path #2 represents this move) then the AI can keep using the path it selected, so it will put its piece in slot 5 (the last node in Path #2). This scenario is the easiest to handle, since the AI will be able to walk down the path it chose at the start of the match. While not a terribly clear name, this is the ‘Matches selected path’ scenario.

If the player places their piece in slot 5 as the second move however (take a look at Path #3, the second node in that path represents this move) then the match has gone off track from the path the AI selected (Path #2), but the Global tree has a node for the move the player has chosen (in Path #3), so it could re-do the WalkPaths algorithm where the node representing the move the player did (the one that caused it to go off-track from the previous path) is used as the root *Node* parameter. In short, if the player performs a move that goes off track from the selected path, and if there’s a node in the Global tree for this move, then recalculate the StatisticallyBest path using the node for the player’s move as the root node. This is the ‘Off path with data’ scenario. This method only works well if the Global tree has enough data, because otherwise...

If the player places their piece in any slot that isn’t 5 or 7 as their second move, then the example Global tree doesn’t have any nodes representing this meaning the AI can’t use WalkPaths with StatisticallyBest to figure out the best path to take. In this case, the AI should fall back to performing completely random moves during a match. This allows it to continue playing, while still gathering data for its Global tree. This is the ‘Off path without data’ scenario.

## Problem 3.3 – What algorithms does the AI use to handle these problems?

First, this is the algorithm ‘DoRandom’, which the AI uses to perform a random move.

Steps:

1. Generate a number between 0 and 8 (inclusive). This number is referred to as *index*.
2. If the slot at *index* is not empty, go to Step 1.
3. Otherwise, place the AI’s piece at this slot.

Now, for the algorithm that the AI uses to determine its move. Steps that are encased in square brackets (‘[‘ and ‘]’) are comments.

This algorithm handles the 3 previously described scenarios. I will refer to this algorithm as the “Find Move” algorithm

Variables:

* *Parent* = The node that will be given as the root node for the WalkPaths (with StatisticallyBest) algorithm.
* *LastNode* = A single node, used when trying to mirror the local tree with the global tree.

Steps:

1. [If the Local tree has nodes in it, then ‘mirror’ the local tree with the Global tree, so the AI gets the Global tree’s version of the nodes] If the local tree has at least 1 node in it:
   1. Set *LastNode* to the Global tree’s root node.
   2. Get the next node from the Local tree (going from the start).
   3. Compare the hash of the node from the Local tree with the hashes of the *LastNode*’s children.
      1. If a matching hash is found, set *LastNode* to the matching node in the Global tree, and go to Step 1.b
      2. Otherwise, fall back to using the DoRandom algorithm. (This is the ‘Off path without data’ scenario)
   4. Set *Parent* to *LastNode*
2. If the Local tree doesn’t have any children in it, set *Parent* to the Global tree’s root.
3. Perform WalkPaths, where *Action* is StatisticallyBest, and *Root* is *Parent*.
4. Get the first node from the path that the StatisticallyBest algorithm chose, and perform the move that the node represents. (Because this algorithm expects the direct children of the root node to only represent the AI’s moves, the first node in the path **should** represent the AI’s move. If it doesn’t, then something has gone wrong.) [This is both of the other scenarios, since the algorithm technically handles both.]

There is a flaw with this algorithm though; it expects that the first children of the root node **only** represent the AI’s moves. If they represent the player’s or both the AI’s and the player’s, the algorithm will fail.

This means the *root* must either represent no move at all (the true root node of a tree), or the move of the player since a node representing the player can only have children representing the AI, and vice-versa. This means that the AI must always make the first move, because if the AI always goes first, then the true root node of a tree will only have children representing the player, keeping the tree in the format that this algorithm expects.

# Proposed structure of the program

## My reasons for proposing a synchronous (multi-threaded) approach

Because the game will be making use of a window (referred to as ‘GUI’ for the rest of the document), the game will require the use of either asynchronous/synchronous technology. I have chosen to take a synchronous (multi-threaded) approach, as I am more familiar with it than asynchronous code.

There will be 2 main threads in the program; the ‘Game thread’, which runs the actual logic for the game (AI taking its move, checking if someone has won, etc.), and the other thread is the ‘GUI thread’ which is the thread the GUI will run on.

The reason for this approach is, imagine the AI gets to the point it has to spend several seconds to calculate a move (due to too much data to go through quickly), if the AI was running on the same thread as the GUI, then the GUI would be shown as ‘Not responding’ – a scenario I would like to avoid, as it makes the game feel slow and unresponsive (and seeing a game produce the “XXX.exe is not responding” message isn’t a terribly great sign of a well-made program).

The solution is to run the game logic in a separate thread (The ‘Game Thread’) from the GUI (The ‘GUI thread’), so even if the AI is taking up to a minute to calculate a move (a scenario my algorithm can’t avoid, but would require **many** matches to have been played), the GUI still remains responsive, and a simple message such as “The AI is thinking” can be displayed to the user so it’s more obvious what the game is doing.

## Communication between threads

One of the most annoying/biggest issues with multi-threaded software is communication between threads.

C# (or rather, the .Net framework) provides a container called ‘ConcurrentQueue’ [1] which is a thread-safe (safe to use between multiple threads at the same time) implementation of a queue.

My solution for the GUI thread talking to the Game thread (‘Game thread to GUI thread’ will be covered later) is to provide a ConcurrentQueue that they can both access. This ConcurrentQueue should store messages, so a basic communication may look like:

*GUI Thread:* Queue Message ‘Start Match’

*Game Thread:* Get a Message from the queue.

*Game Thread:* Sees that it is the ‘Start Match’ message, so executes the code to start a match.

More examples of how this messaging will work will be explained later.

Now, we have a basic solution for how the GUI thread can talk to the Game thread, so how does the Game thread talk to the GUI thread? The windows in a WPF program provide something called a Dispatcher [2][3] which to put it simply, allows threads to queue up ‘tasks’ to be ran in the thread that the Window itself is ran in. (slightly ironically, I’m pretty sure it uses asynchronous technology).

So, with the dispatcher, a basic communication may look like:

*Game Thread:* Queue up task ‘Update On Screen Grid’, passing data on what the grid looks like

*GUI Thread:* [At some point] Execute task ‘Update On Screen Grid’

The reason that the Game Thread can’t directly execute these tasks on the GUI, is because WPF does not allow anything displayed in a window to be modified from any thread other than the thread the window is being ran on. This means that the Dispatcher is used as a ‘proxy’ for the Game thread to update the GUI.

Also, in WPF, I have not been able to find a reliable way to run a piece of code every ‘tick’ (update of the window), so this makes it difficult/slow to make the GUI thread also use the ConcurrentQueue mentioned earlier (otherwise, I’d just make both threads share the queue, and if either thread got a message they shouldn’t have, they just requeue it. Or at worst, supply one queue for ‘gui->game’, and one queue for ‘game->gui’).

[1] <https://msdn.microsoft.com/en-us/library/dd267265(v=vs.110).aspx>

[2] <https://msdn.microsoft.com/en-us/library/system.windows.threading.dispatcher(v=vs.110).aspx>

[3] <https://msdn.microsoft.com/en-us/library/system.windows.window(v=vs.110).aspx>

## The Game loop

My idea of how the game logic should work, is that there is a ‘Board’, and that there are ‘Controllers’.

The ‘Board’, as the name suggest, provides an interface to modify a tic-tac-toe game board. The ‘Board’ will also contain the logic for executing a match.

The ‘Controllers’ can be viewed as the ‘Players’. There is a controller for the ‘X’ piece, and a controller for the ‘O’ piece. There will be a controller that handles the input of the player (explained later), and a controller which provides the AI.

My reasoning for having ‘Controllers’, is that it allows my code to be more reusable and modular. For example, I could create a match between two player controllers to represent a ‘Player versus player’ match; I could put up two AI controllers against each other to have the AI fight itself, and of course, I could put a player controller up against an AI controller. It also lends way to further additions, as an example if I ever wanted multiplayer in the game, I could just create an ‘OnlinePlayer’ controller that handles getting input from over a connection and put it up against another player controller.

The use of a hash to represent the board state (explained earlier in the document) will allow controllers to be written in a ‘Piece-independent’ way, meaning the code can be written in a way that doesn’t matter whether the controller is ‘X’, or ‘O’. Although, as explained previously alongside the AI’s algorithm, the AI controller will always have to be the piece that gets the first move (this is an issue with the algorithm the AI uses, not an issue with the hash/program structure/whatever else).

Controllers should be given information about the state of the match at the following steps, where each step is named with text inside of square brackets:

* [OnStart] When a match is first started, the piece that the controller is assigned to should be passed to it. This gives controllers a chance to setup whatever they need.
* [OnDoMove] When the controller needs to perform its move, a hash of the board (hashed from the point of view of the controller), as well as the index of where the last piece was placed should be passed. Incidentally, the index passed represents the index of where the enemy controller placed its piece since the controllers take it in turns to place a piece.
* [OnAfterMove] After the controller has performed its move, the hash of the board and the index of where the controller placed its piece should be passed. This allows controllers to separate their logic to update their internal state from the logic to perform a move.
* [OnEnd] When the match ends, the hash of the board, the index of the last piece placed, and the result of the match (win, loss, tie) should be passed to the controller. This gives controllers a chance to, for example, update the GUI to say “You have won”, or for controllers such as the AI controller to save data for future matches.

The Board will contain the logic of performing a match, and the algorithm for performing a match is defined as:

Variables:

* XCon = The controller that represents the X piece.
* OCon = The controller that represents the O piece.
* TurnPiece = If ‘X’, then it’s XCon’s turn. If ‘O’, then it’s OCon’s turn.
* LastIndex = The index of where the last piece was placed on the board.

Steps:

1. Perform the OnStart step for both controllers, passing ‘X’ to the XCon, and ‘O’ to the OCon.
2. Default TurnPiece to either ‘X’ or ‘O’.
3. Default LastIndex to some value, this value represents ‘No pieces have been placed yet’.
4. Perform the OnDoMove step for the controller who’s turn it currently is (based on TurnPiece). A hash of the current state of the board (where ‘M’ represents the controller’s piece, and ‘O’ represents the enemy controller’s piece) is given, as well as the index of where the last piece was placed.
5. Perform the OnAfterMove step for the controller who’s turn it currently is, where a new hash of the board is given, and the index of where the controller placed its piece is given (so it doesn’t have to keep track of it itself. In code, this will create less bugs).
6. Check to see if someone has won, or if there is a tie. If no one has won yet, and there isn’t a tie, then:
   1. Set TurnPiece to ‘X’ if its currently ‘O’, or set it to ‘O’ if its currently ‘X’.
   2. Go to Step 4.
7. Perform the OnEnd step for both controllers, where a new hash of the board is made for both controllers (from their own points of view), and passing whether they won, lost, or tied.

One final benefit of going with the concept of a controller, is that the Board **only** focuses on the ‘rules’ of the game, whereas the controllers are what provide the AI/interaction for the player, creating a separation of responsibilities, leading for less places for bugs to pop up (e.g. if the AI has something wrong with it, then there is a very high chance that the issue is with the AI’s controller, not the Board, since the Board doesn’t even really do anything specifically for the AI).

## The Game Thread

All that’s really left to talk about the Game thread itself (including the Board, but not any specific controller) is how it starts a match.

It’s simple enough to be explained in an algorithm:

Variables:

* *Queue* = The ConcurrentQueue shared between the GUI and Game thread.

Steps:

1. Get a message from the *Queue*.
2. If this message is ‘Start Match’, then start a match between the two controllers that the message provides. (This starts the algorithm for the Board’s game loop).
   1. Once the match is over go to Step 1.
3. If this message isn’t any of the above, requeue it into the *Queue*.
4. Sleep the thread for 50 milliseconds.
5. Go to Step 1.

In the actual project, I may need to create more messages for the above algorithm to handle, but for a simple ‘Start Match’ message it works well.

As a note, Step 4 makes the thread sleep for a small amount of time, because when turned into code the algorithm is a ‘while(true)’ loop and can eat up a lot of CPU time just constantly checking for a message, so a sleep is inserted to make it only check once every 50ms, instead of checking non-stop.

The final thing to talk about it is how the game thread gets terminated. In C# (or .Net, to be precise) there is an option to ‘abort’ a thread [1] which will cause a special exception to be thrown in the thread. I plan on using this to make the GUI abort the Game thread when the GUI is closed. If the Game thread didn’t close, then the game’s process would still run in the background, even after the GUI is closed. I admit I could design this better, so there is some “doGameThread” flag, but there isn’t much to gain from it in this case.

[1] <https://msdn.microsoft.com/en-us/library/5b50fdsz(v=vs.110).aspx>

## The GUI Thread

The GUI itself should at the very least provide a 3x3 grid (representing a tic-tac-toe board), which contains an up-to-date view of a match, and should allow the player to interact with it. More information about what the GUI displays is described in the ‘Success Criteria’ at the start of the document.

WPF is an event-based framework, meaning doing things such as “Click button”, “Move mouse over text”, etc. can fire an event in the code. So, the 3x3 grid in the GUI should be made up of buttons (or just plain text) where their “On click” event sends a message through the ConcurrentQueue.

For example, if the user presses the top-right slot of the grid, then a message that says ‘Player Piece at slot 2’ should be sent. This is then handled by the player’s controller on the game thread, which will read in the message, and if the move is valid (it’s the player’s turn, and the slot is empty) then it is performed, but if the move is not valid then it simply does nothing.

The GUI itself should provide an interface for the Game thread, which allows the Game thread to modify the GUI (update the grid, change some text on screen, signal whether a game is playing or not, etc.) via the use of the GUI’s dispatcher.

The GUI is also responsible for setting up the Game thread, because in a WPF program, the main window’s (the GUI) constructor is generally the entry point of the program.

The GUI should provide a ‘Start Match’ button, which has the “On Click” event of sending the ‘Start Match’ message described earlier. The two controllers for this message are the AI controller, and the other being the Player controller.

## The AI Controller

The AI Controller is the controller that will provide the logic for the AI.

### [OnStart]

When a match is started, the AI should make sure it has loaded the Global tree from a file (if it hasn’t done it already).

### [OnDoMove]

When it is the AI’s turn to perform a move, it needs to do two things.

1. The *Hash* and *index* given to it can be used to figure out which move the enemy controller last made. This move should be added into the local tree.
2. The AI should then perform its ‘FindMove’ algorithm, to perform its move.

### [OnAfterMove]

When the AI has completed its move, it should add its move to the local tree.

### [OnEnd]

When the match has ended, the AI should bump either the ‘win’ or ‘loss’ counter of each node in the local tree, then merge the local tree into the global tree.

The AI may also save its Global tree into a file at this stage.

## The Player Controller

The Player Controller is the controller that allows the user to interact with the game board, as well as being responsible for displaying the current state of the match to the user.

During a match, the player controller will make use of the GUI’s dispatcher (explained earlier), as well as an interface provided by the GUI so that the controller can update it.

### [OnStart]

When a match is started, the controller should pass which piece the player is playing as to the GUI, so the GUI can then update itself to display something such as “You are playing as X”.

### [OnDoMove]

During the player’s turn, it should first pass the current *Hash* of the board to the GUI, so it can display to the user the current state of the board. It should also tell the GUI that it’s the player’s turn, so it can display “It is your turn” on screen, as well as allow the user to interact with the board.

The controller should then keep checking for a ‘PlayerPlaceMessage’ in the ConcurrentQueue (also explained earlier). This message will contain the index of where the player wants to place their piece. If the move is invalid (the slot is non-empty), then the controller will ignore it, and wait for another message. Otherwise, it will perform the move.

### [OnAfterMove]

After making its move, the controller should send the new *Hash* of the board to the GUI, to keep it up to date. It should then tell the GUI that it is the AI’s turn, so it can stop the player from creating ‘PlayerPlaceMessages’, as well as display “It is the AI’s turn”.

If the player could create messages when it wasn’t their turn, then the messages would ‘buffer’, allowing the user to queue up any number of moves to automatically be performed. The issue is, most users won’t find a use for this, and instead if they click one too many times they’ll find that their next turn is basically ‘skipped’.

### [OnEnd]

As with the other stages, the AI should send the final *Hash* of the game board to the GUI, and then also inform the GUI that the match is over, alongside with the match’s *Result* so it can display “You have won/lost/tied” as appropriate.

# Usability

The usability features of the game involve how easy/accessible the game’s GUI is to use.

The GUI itself will contain a 3x3 grid, which mimics the kind of 3x3 grid that would be used in a real-life game of tic-tac-toe. This means that anyone familiar with the game will be able to easily understand how to interact with the GUI.

The only action that needs to be performed on the GUI, is clicking. The user will have to click a ‘Start Match’ button to begin a match, they also have to click on the 3x3 grid to place their piece on the screen. This means that there is virtually no barrier of entry to play the game, and even a child who is too young to read would be able to play it, in contrast to if I went with a command-line interface for the game, where the user would have to type in commands to play.

The GUI should display which piece the user is playing as (to avoid confusion), as well as whether it’s the player’s turn, or the AI’s turn. This is mainly to prevent confusion to the player.

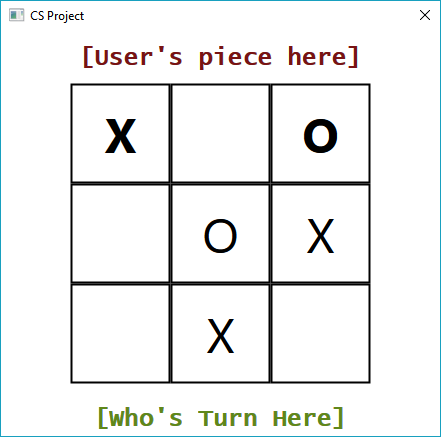
Because the GUI and the game logic run on separate threads, in the event that the game logic is taking a while to process something (such as the AI’s logic), then the GUI will still be fully interact able to the user. This prevents the GUI from freezing, which would make it feel slow, unresponsive, and buggy.

Each cell in the 3x3 grid should be large enough to see which piece is in it, and to be easy to click. The font used to display the pieces in a grid should be clean and simple, where the ‘X’ and ‘O’ characters are easy to read. The font may also be **bold** to make it easier to see the pieces.

The image below is a small mock-up GUI I have put together. It uses WPF, so this mock-up is representative of what the final GUI could look like (it may even become the final GUI itself). Some things to note are:

* The font used for the 3x3 grid is Segoe UI, as it is a very clean and clear font.
* Each cell in the grid is 100x100 pixels. There is no particular reason for this, except that I found this size to be the nicest to look at. Other sizes tried were 50x50, 125x125, 150x150, and 200x200.
* The top row of the grid uses bolded text, while the rest use non-bolded text (to see which is easier to look at).
* The “[User’s piece here]” and “[Who’s turn here]” text are coloured differently so they stand out.
* The font used for the two pieces of text mentioned above is ‘Lucida Console’. It is different from the 3x3 grid so it stands out.
* The window itself is 455x460, so should be able to display properly on most monitors.

*[Figure: Mock-up GUI]*



To aid users who may not know about what tic-tac-toe is, the game should display a message box detailing how to play tic-tact-toe when it is loaded for the first time. There should also be a ‘Help’ button somewhere so the user can bring the message box back up.

The cells of the grid should change colour whenever the player hovers their mouse over them (and change back to their original colour afterwards). This creates a sense of interactivity to the user.

Touch screen monitors should work fine with the game, as the cells are big enough to comfortably tap, and since touch screen monitors basically turn your finger into a mouse, the same code can be will be used for both cases (meaning less places in the code for bugs).

# Test Data for development

Test Data for Development will be a set of automated unit tests that test specific parts of the code to ensure they are working correctly.

Some tests I may create during the development of my project include:

* Performing a dummy match between two specially created Controllers, and checking that the correct data is being passed to the Controllers by the Board. (Seeing that the hashes are correct, seeing that the indices passed are correct, etc.)
* Performing 3 dummy matches, one match to make sure one controller wins, one match to make sure one controller loses, and another match to see if the controllers can tie. (basically, checking the win condition logic).
* Testing to see if a hand-crafted tree of nodes can be successfully written to and then loaded from a file, with all the data properly intact.
* Testing the WalksPath algorithm, where Action is StatisticallyBest, on a hand-crafted tree of nodes to ensure that they are working as designed.
* Testing the WalksPath algorithm to see if it does go over every possible path in a hand-crafted tree.

# Test Data for beta testing

Test Data for beta testing will be a set of tests that must be performed manually, usually because writing a unit test for it would be impractical.

Some tests I may create include:

* Making sure the game does nothing if the user clicks on the grid, and there’s no match running.
* Making sure opening and closing the game does not result in random crashes.
* Making sure that clicking the ‘Start Match’ button actually starts a match.
* Making sure that clicking on a non-empty slot on the grid does nothing, and that the user can try again at placing their piece.
* Attempting to manically spam click things to try and cause the game to crash.
* Making sure that the text displaying who’s turn it is, and the player’s piece, are correct.
* Making sure that the message box describing what tic-tac-toe is, is only displayed a single time when the game is launched, and then never again (unless the ‘Help’ button is pressed, of course).

Development

# Iterations of development

[TODO] (Self note, remember to use Git to ‘go back in time’ if I need something like a screenshot from an earlier version)

# Prototypes

[TODO]

# Evidence of modular code

[TODO]

# Evidence of validation

[TODO]

# Review

[TODO]

Evaluation

# Testing

[TODO]

# Testing of usability features

[TODO]

# Overall evaluation

[TODO]

# Future Maintenance

[TODO]