Project Proposal – Robert Lucas

# Concept

Chess is an ancient game played by millions around the world and has seen many apps made for competitive and casual play however these tend to offer only the default versions of chess or slight spinoffs that still play on the same sized board with the same pieces but with changes to the rules.

I plan to make a chess game with support for adding many game modes as well as multiplayer support, save game support and AI that will work with any new game mode.

While other chess options do exist, the key unique feature of this game is the many available game modes and the ability to save all of them. This will make the game more appealing to a casual audience who want to experience unique variations of chess, be able to play chess with more than one friend or play longer games that they can save and come back to.

The game modes will also include other established board games such as checkers and Viking Chess (Hnefatafl). While some of these can be played online (it’s worth noting that I couldn’t find an online version of Viking Chess), there isn’t a single place where you can play all of them which makes this game more convenient than the alternatives.

# Stakeholders

The stakeholders in this project are myself and the players as all decisions should be made to either make development more feasible for me or to improve the player experience as this will bring in more players and improve player retention.

## Key points from responses to questionnaire (10 people)

### Do you have a Windows device?

* Almost everyone asked (8) had one
* One person had a Mac

Making the game exclusive to Windows shouldn’t reduce the player base by too much however making the game support Mac wouldn’t require much more development due to Unity’s platform independence.

While I have considered supporting Android / IOS, due to smaller screen sizes and different typical aspect ratios, the UI would have to be completely redone for mobile devices. Most mobile devices probably wouldn’t have enough power to run the AI at a reasonable speed.

### Do you know how to play Chess?

* Around 1/3 of people knew how to play chess

This statistic means that the tutorials / help provided must not assume that the player knows how to play chess

The name of the game should not leave the impression that this game is only chess

### What do you think of a game where you could play lots of unique board games against friends?

* Most people (7) liked this idea
* Some (2) mentioned that this would need to be different from the offerings of online sites

This game needs to have lots of game modes that either aren’t available in other places or aren’t available in one place.

### Do you know how to forward ports?

* No-one knew how to forward ports

The game needs to have detailed instructions on how to forward ports

In the future the game could use an external proxy to no longer require port forwarding.

# Computational methods used

### Abstraction

Abstraction is necessary to hide complexity allowing new parts to be developed without needing understanding of how the entire existing code base works.

It also allows unnecessary detail to be removed completely in some places making the game simpler to develop

* The game will be 2D and not realistic simplifying it reducing development time and making the board easier to understand for a user
* The game will use Unity to handle most of the rendering, IO and packaging the game into an executable reducing development time
* After the multiplayer system is developed, the game logic will only call exposed functions on the networking classes without having to worry about handling networking allowing different sections of the solution to remain more independent allowing for easier iteration on different parts of the code
* IP addresses will be encoded to alphabetic strings to make them easier to remember and pass on to friends

### Thinking Ahead

Thinking ahead is necessary to prevent bad decisions that can make the game difficult to develop / work with later. This is especially important as I need to make it very easy to add new game modes.

* As the game is symmetrical, code can be reused for both players decreasing development time
* Instead of writing a client and a client-server hybrid to allow one of the players to host, the client can be reused and a separate server can be made. The local client can then connect to the local server. This will reduce code duplication between a client and a client-server hybrid reducing development time and decreasing the chance of errors occurring due to mismatches between how a client and how a client-server works.
* The board can be serialised to save its state and this needs to be done in a smart way that can work for any game mode and any number of custom pieces with custom data as well as not breaking with future updates

Decomposition

A project must be decomposed into smaller chunks to allow components to be developed one at a time making the process simpler.

* The networking library, game logic, AI and visuals can all be developed separately as they only interact with each other in limited ways allowing the project to be easier to manage
* These subsystems are further divided into smaller chunks such as the networking being divided into client and server and the chess manager being broken down into the game, save and input system

Procedures

Procedures are needed to complete series of steps with each step being dependent on the ones before.

* The game will use procedures such as:
  + The network library going from receiving data to validating it to processing it to passing it to the game logic for it to be applied to the board
  + The client clicking on a square to move a piece to the input manager receiving that to the game manager validating the move to the game manager applying the move to the game manager checking for a check or checkmate and finally the game manager updating the visuals manager

Logic

* Logic will be used very frequently for example evaluating which moves a player can and can’t make or for not letting more players join a lobby once it is full both of which are essential for the game to work

Concurrency

Concurrency is needed to run slow or blocking (such as waiting for a client connection) code that isn’t interdependent. This is especially important when using the Unity Engine as it has a single main thread and any code running on that thread directly affects framerate.

* The networking library will need to run concurrently to be able to receive and process data from other players independently of the game’s framerate
* The AI will need to run on a separate concurrently to the main thread as calculating the best move can take up to 30 seconds and the game shouldn’t freeze during this time as this can ruin the user experience

### Divide and Conquer

This technique is useful for splitting a task into smaller subtasks that can be completed in parallel with greater efficiency

* AI will run on multiple threads at the same time with each thread searching different possible moves

Encapsulation

Encapsulation is used to control communication between classes to carefully select what class data can be modified externally

* Input manager will encapsulate Unity’s input system to make it easier to use and adding features such as support for multiple key presses
* To maintain a comprehensible class hierarchy, the ChessManager will encapsulate many methods from other managers to keep the relation tree more similar to a star than a mesh making it easier to understand and debug.

Backtracking

* The Mini-Max algorithm uses backtracking

Heuristics

Heuristics are useful for optimising algorithms based on existing knowledge

* The AI will use heuristics such as not going further down a path that has a very low score

# Features based on existing solutions



### Features included

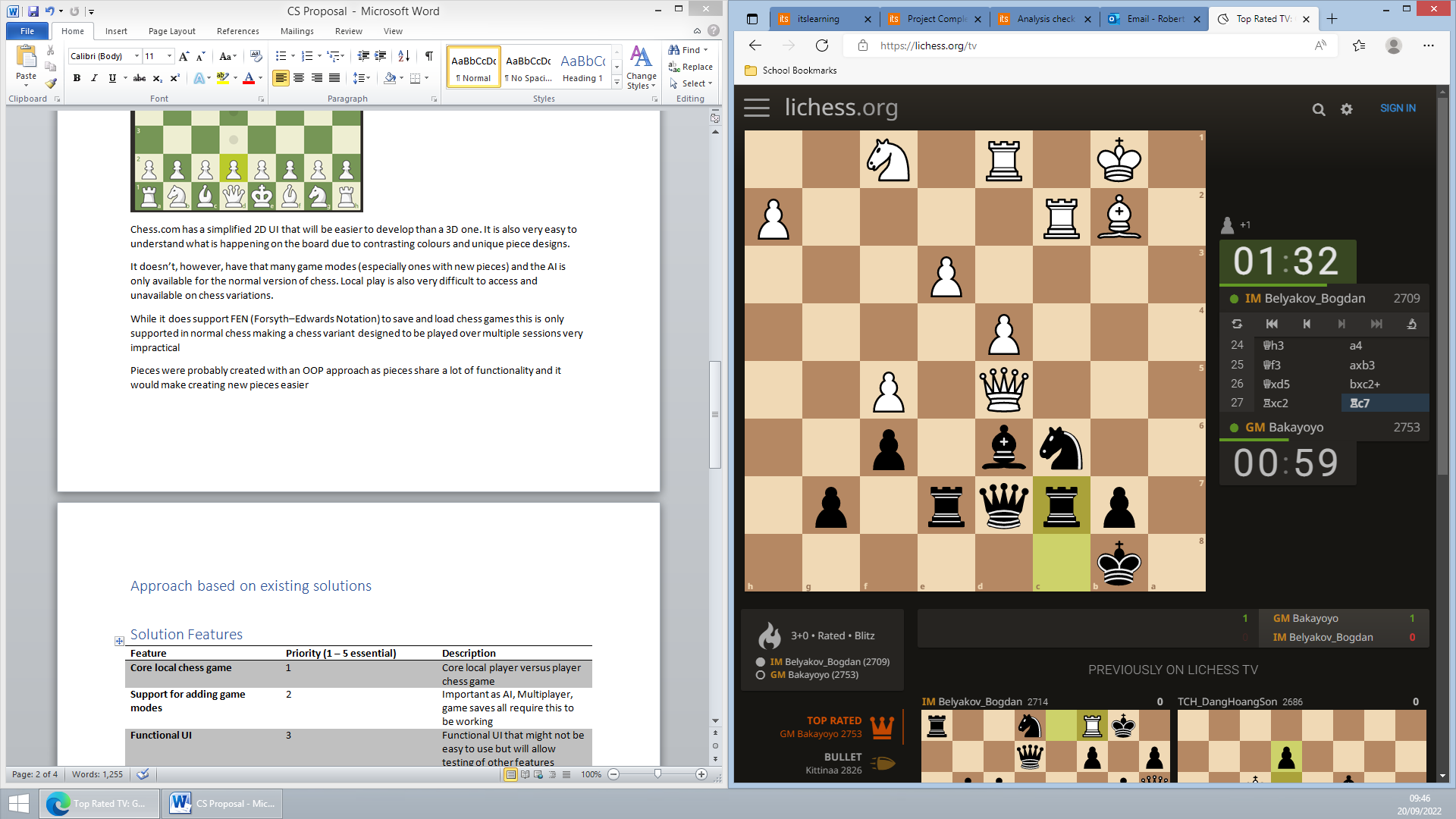
Chess.com has a simplified 2D UI that will be easier to develop than a 3D one. It is also very easy to understand what is happening on the board due to contrasting colours and unique piece designs.

Pieces were probably created with an OOP approach as pieces share a lot of functionality and it would make creating new pieces easier

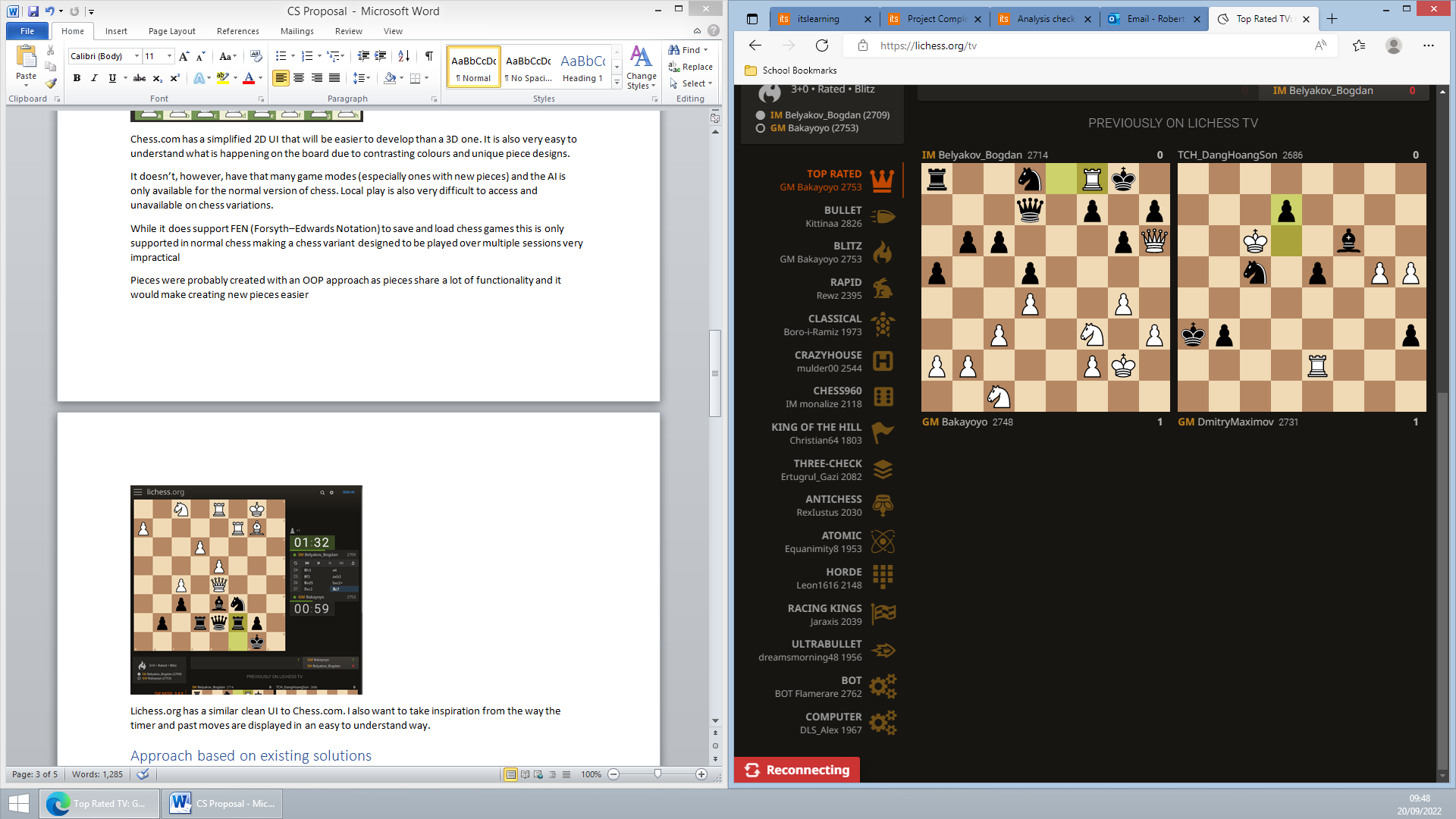
### Features missing

It doesn’t, however, have that many game modes (especially ones with new pieces) and the AI is only available for the normal version of chess. Local play is also very difficult to access and unavailable on chess variations.

While it does support FEN (Forsyth–Edwards Notation) to save and load chess games this is only supported in normal chess making a chess variant designed to be played over multiple sessions very impractical



Lichess.org has a similar clean UI to Chess.com. I also want to take inspiration from the way the timer and past moves are displayed in an easy to understand way. It also displays moves available to each piece and highlights a piece’s last moves.



Lichess does also have quite a few modes however, like Chess.com, Lichess doesn’t stray too far from normal chess and doesn’t support saving games in progress that can’t be represented with FEN



This is a generic app on the Play Store with the board represented in 3D. I find this representation to be more cluttered and more difficult to understand.

A 3D view would also require more developer time as models for custom game modes would have to be created and there are extra complications around camera controls and ensuring that the contrast is good enough on all the pieces to see them clearly

I do, however, like that when you click on a piece it shows you a preview of legal moves and believe that it would greatly improve the user experience

## Summary

### Key positives other solutions have

* 2D simplistic UI
* Being able to load games (at least for normal chess)
* Shows where pieces can move
* Shows where pieces moved last turn
* Simple piece icons

### Unique components not found in most other solutions

* Support for a large number of game modes
* Support for saving every game mode
* Support for online and local play
* AI support for every gamemode

# Approach based on existing solutions

Most chess games will probably use an object-oriented approach to create the pieces as they need to share some functionality which can be provided by a parent class. I will do this and also use a similar approach for implementing multiple game modes as large amount of functionality will be shared between them

A lot of chess AIs use the minimax algorithm with alpha-beta pruning and I will use this as I have some experience with it. The minimax algorithm (excluding some optimisations) will also work with any game mode as it requests all possible moves from a piece which custom pieces can also provide. I might also use Zobrist hashing to create transposition tables for the board but this would be a massive increase in complexity, especially for custom game modes.

Sockets are commonly used for low-level and efficient communications and as I want very fine control over network communications I will be using that.

# Solution Features

|  |  |  |
| --- | --- | --- |
| Feature | Priority (1 – 5 essential) | Description |
| Core local chess game | 1 | Core local player versus player chess game |
| Support for adding game modes | 2 | Important as AI, Multiplayer, game saves all require this to be working |
| Functional UI | 3 | Functional UI that might not be easy to use but will allow testing of other features |
| Multiplayer | 4 | Allow players to play with each other across the internet  Must support working with any future game modes |
| Help System | 4 | Help system that helps players understand how to use the game |
| AI | 4 | An AI that can play any game mode |
| Save games | 5 | Allows player to save the current state of the game and load it later including loading it into a multiplayer game |
| Good UI | 6 | An intuitive UI that is easy for a new player to use, possibly with tutorials |
| Animations | 7 | Animations for pieces moving from square to square |
| Themes | 7 | Allow user to change board themes |

# Limitations

### Art

This game will require art assets for each piece as well as a logo, fonts and UI elements. Because of my limited knowledge in this field and time limitations I will use third-party assets for some components and simplistic elements for most other things

### Money

Third-party asset packs, animations packs, font packs and more cost money which has to be managed carefully as this project is on a small budget so in most cases free alternatives will be used

### Computational power

The AI can’t be too complex as the typical user system won’t have a powerful CPU and this game needs to be accessible to as many people as possible meaning that the AI may be quite limited in its skill

### Time

Development time is limited due to the lack of other developers and the limited time frame therefore each feature must be considered with the time it takes to make it

### Platform

While ideally this game would be made on a website, due to my more limited knowledge of JavaScript and the large extra time requirement creating a website has I’ve decided to use Unity and C# as an alternative. This would reduce player numbers as players as players would need to also have a game distribution platform such as Steam installed

This game also won’t be cross platform as despite Unity supporting IOS and Android, these require extra development time due to UI considerations and would require scaling back the AI even further

# Requirements

* A powerful computer capable of running Unity, Visual Studio 2022 and completing Unity builds in a reasonable amount of time
* Ability to code C# as well as knowing Unity and socket specific parts of C#
* Ability to use Unity
* A second computer capable of running the built game and connecting to a hotspot to test multiplayer support
* HTML knowledge to build the help webpages

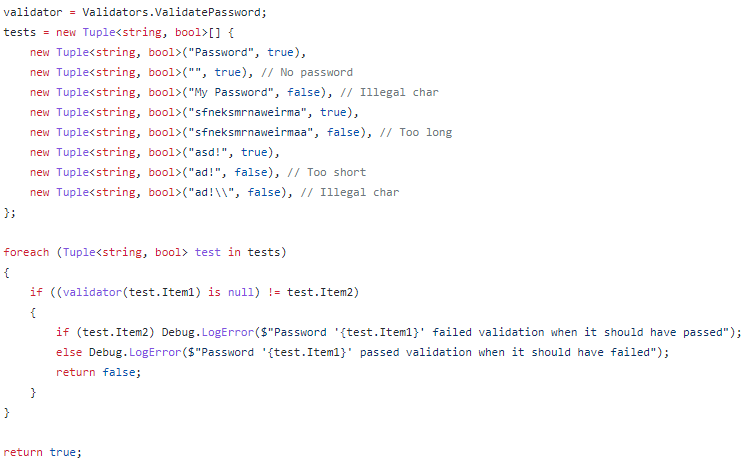
# Success Criteria

|  |  |  |  |
| --- | --- | --- | --- |
| Criteria | Category | Measured by | Justification |
| All essential solution features included | Functionality  Usability | Tick list  Testing | The game must be complete |
| Error-free | Robustness | Stress testing (possibly automated) | Clients should never have to restart the game due to errors to ensure a good UX |
| Secure (optional) | Robustness | Ensure data is encrypted  Use Wireshark to inspect packets sent and ensure they are encrypted | Optional as no sensitive data is transferred |
| User friendliness | Usability | Tests with target audience | Users must be able to use the software without any external assistance |
| AI performance reasonable | Functionality | AI makes moves (on average) in under 1 minute on mid-range hardware | Users should be able to use the AI without requiring very powerful hardware |
| First functional build released by end of November | Functionality | - | Ensures project is completed on time |

# Testing

Excel spreadsheets for testing will be created as the project progresses as the UI, AI and game modes are all subject to change and thus might require different tests.

Testing itself will be carried out through a combination of manual tests and automated tests. These automated tests will be in a class in Unity that is only compiled and ran in the editor where it runs after every code change, not the release build for optimisation purposes.



^ Snippet of the code that tests the below validator



Solution

# Overview

The ChessManager will encapsulate functions of other classes and handle communication and coordination between different systems (listed below).

# Game Mode System

This game will support many different game modes. This will be achieved using OOP techniques, namely polymorphism.

## GameManagerData

Uninitialized form of GameData used before full instance is needed e.g. in main menu when getting a list of all game modes

|  |  |  |
| --- | --- | --- |
| GameManagerData | Description | Type |
| **Data** | None |  |
| **Methods** |  |  |
| GetUID | Returns a unique ID for the game mode | Int |
| GetName | Returns the name of the game mode | String |
| GetTeamSizes | Returns the team sizes for this game | TeamSize[] |
| TeamAliases | Returns names for the teams e.g. { “White”, “Black” } | String[] |
| GetDescription | Returns a short description of the game mode (mainly used as a backup in case the help system doesn’t work) | String |
| Instantiate | Returns an instance of the full GameManager class | AbstractGameManager |

## GameManager

Controls a gamemode. Classes inheriting from GameManager might also implement additional data and methods

|  |  |  |
| --- | --- | --- |
| GameManager | Description | Type |
| **Data** |  |  |
| GameManagerData | A reference to the ‘Data’ version of this class | GameManagerData |
| Board | A reference to the Board class of this game mode – see below | AbstractBoard |
| **Methods** |  |  |
| GetData | Returns a serialised version of the game – ready to be saved | SerialisationData |
| LoadData | Loads serialised data | Void |
| GetMoves | Returns a list of available moves to make | List<Move> |
| OnNoMoves | Handles what happens if no moves are available. Usually just makes the opposing team win | Void |
| OnMove | Applies a selected move. Returns either the next player to play’s ID or the winning team ID in the form –(TeamID + 1). If the returned value is less than 0, a team has won. | Int |
| GetScore | Returns a score for the current position. Used by the AI to determine how good a position is | Float |
| Clone | Returns a clone of the game. Used by the AI to test different moves | AbstractGameManager |

## Board

|  |  |  |
| --- | --- | --- |
| Board | Description | Type |
| **Data** |  |  |
| PieceBoard | 2D piece array representing the positions of pieces currently on the board | PieceBoard[,] |
| **Methods** |  |  |
| GetData | Returns a serialised version of the board | SerialisationData |
| LoadData | Loads serialised data | Void |
| GetMoves | Returns a list of available moves | List<Move> |
| GetPiece | Returns a piece at a position or null if there is none there | AbstractPiece |
| GetBoardRenderInfo | Returns specifications for the board e.g. size, highlighted squares and removed squares | BoardRenderInfo |
| OnMove | Applies a selected move | Void |
| GetScore | Returns a score for the current board. Used by the AI to determine how good a position is | Float |
| Clone | Returns a clone of the board | AbstractBoard |

## Piece

|  |  |  |
| --- | --- | --- |
| Piece | Description | Type |
| **Data** |  |  |
| Position | The piece’s current position | V2 |
| AppearanceID | The piece’s current appearance | Int |
| Team | The team the piece belongs to | Int |
| Board | The board the piece is on | AbstractBoard |
| **Methods** |  |  |
| GetData | Returns a serialised version of the piece | PieceSerialisationData |
| LoadData | Loads serialised data | Void |
| GetMoves | Returns a list of available moves | List<Move> |
| OnMove | Applies a selected move | Void |
| GetValue | Returns the value of the piece. Used by the AI to determine how good a position is | Float |
| Clone | Returns a clone of the piece | AbstractPiece |

# Networking System

Handles all online communications

## Network Manager

|  |  |  |
| --- | --- | --- |
| NetworkManager | Description | Type |
| **Data** |  |  |
| chessManager | Reference to the ChessManager | ChessManager |
| server | Reference to the Server, if there is one | Server |
| client | Reference to the Client, if there is one | Client |
| **Methods** |  |  |
| Awake | Initialises NetworkManager | Void |
| OnApplicationQuit | Shuts down the NetworkManager when Unity exits | Void |
| OnLocalMove | Passes a move to the Client to be sent to other players | Void |
| GetPlayerList | Gets the list of all players from the client | ConcurrentDictionary<int, ClientPlayerData> |
| GetLocalPlayerID | Returns the ID of the local player | Int |
| Host | Hosts a game | Void |
| HostStartGame | Starts a hosted game | Void |
| Join | Joins a game | Void |
| ConnectionFailed | Handles what should happen on a connection failure | Void |
| GetPing | Calls the provided action with the ping as a parameter | Void |
| Stop | Shuts down the server and client if either are running | Void |
| **Client Callbacks** | Methods that directly map to methods of other classes used to maintain an understandable reference tree. | All Void |

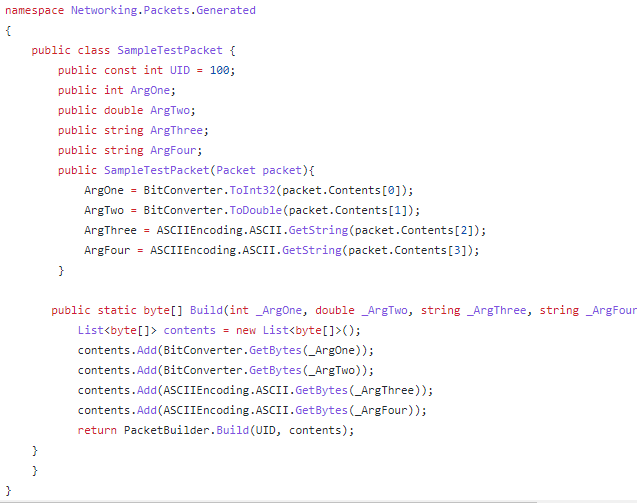
## Client and Server

These bare many similarities and both function by having a receive thread that reads incoming messages one at a time and a send thread that sends messages one at a time (the server also has a thread for accepting new clients). The server primarily acts as a relay passing on messages from one client to all other clients such as a move update. See ‘Pregame Flow’ diagram for a representation of what happens in the lobby and ‘Game Flow’ for a representation of what happens during a game.

## Packet System

Every unit of data sent between the server and client is a ‘packet’ formatted in a very specific way

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Description | Type | Length |
| PacketLen | Length of the entire packet | Int | 4 bytes |
| UID | Unique identifier for this type of packet | Int | 4 bytes |
| Content block 1 length | Length of the first block of content | Int | 4 bytes |
| Content block 1 | Any | Type according to packet type | Content block 1 length bytes |
| [Repeat last two for all content] |  |  |  |



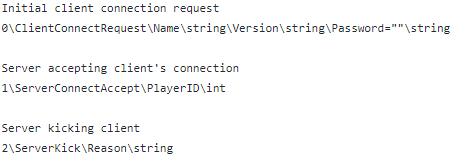
Packets, such as the one above, are automatically generated from a text file of requirements in conjunction with the ‘PacketBuilder’ that converts these to a byte array ready to send and back from received bytes allow the developer to never have to interact with raw bytes for sending and receiving data as usage of the system looks like this:



^ Encoding

# 

^ Decoding



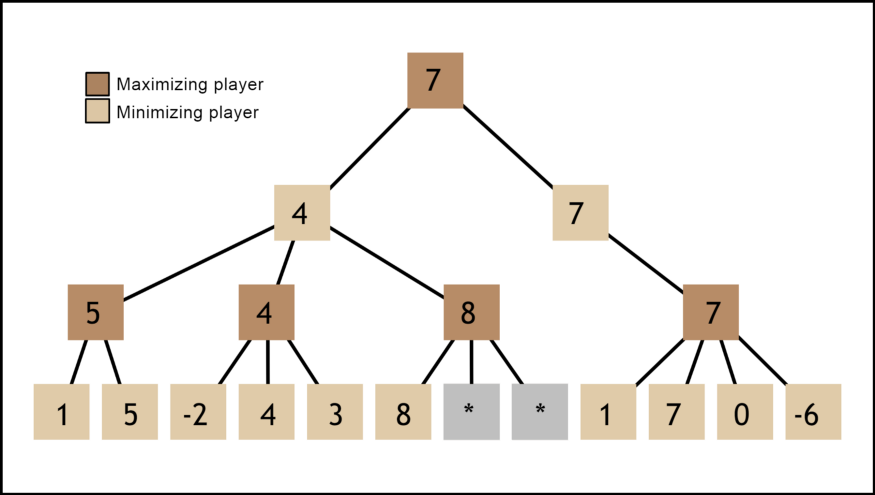
^ Packet generating file (Any line without a ‘\’ is a comment)

# AI System

The AI will use the minimax algorithm to calculate the best move and use Alpha-Beta pruning for optimisation. The AI will be given a time limit and try to look as many moves ahead as it can in that time frame. To allow this to work with all game modes, game modes must implement a cloning method and an evaluation method that returns a score for the current board.

# https://robertlucas.pythonanywhere.com/static/projects/chess/minimax.png

^ Diagram explaining the minimax algorithm. Source (for this and next image): <https://medium.com/@SereneBiologist/the-anatomy-of-a-chess-ai-2087d0d565>



C

B

A

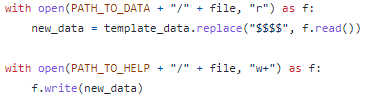
^ Diagram showing AB pruning.

After node A receives the value 4 from node B, it moves on to node C. As C is maximising, once C gets its first value of 8, it doesn’t need to check any more children as we know it will return at least 8 and as A is minimising, A won’t take its value. In some scenarios AB Pruning can reduce end nodes searched by 99.8%

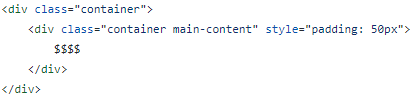
# Help System

The help system will use local static HTML and CSS files to show help information in the default browser. This allows rapid development and iteration on help menus with support for various layouts and images. Developing a help / tutorial system in Unity would be very slow due to the more manual approach to element scaling and positioning.

To create the static HTML pages a python program will take a template.html file and combine it with other HTML files to keep a consistent sidebar and style between pages and allow changes to this style to automatically propagate through all pages.



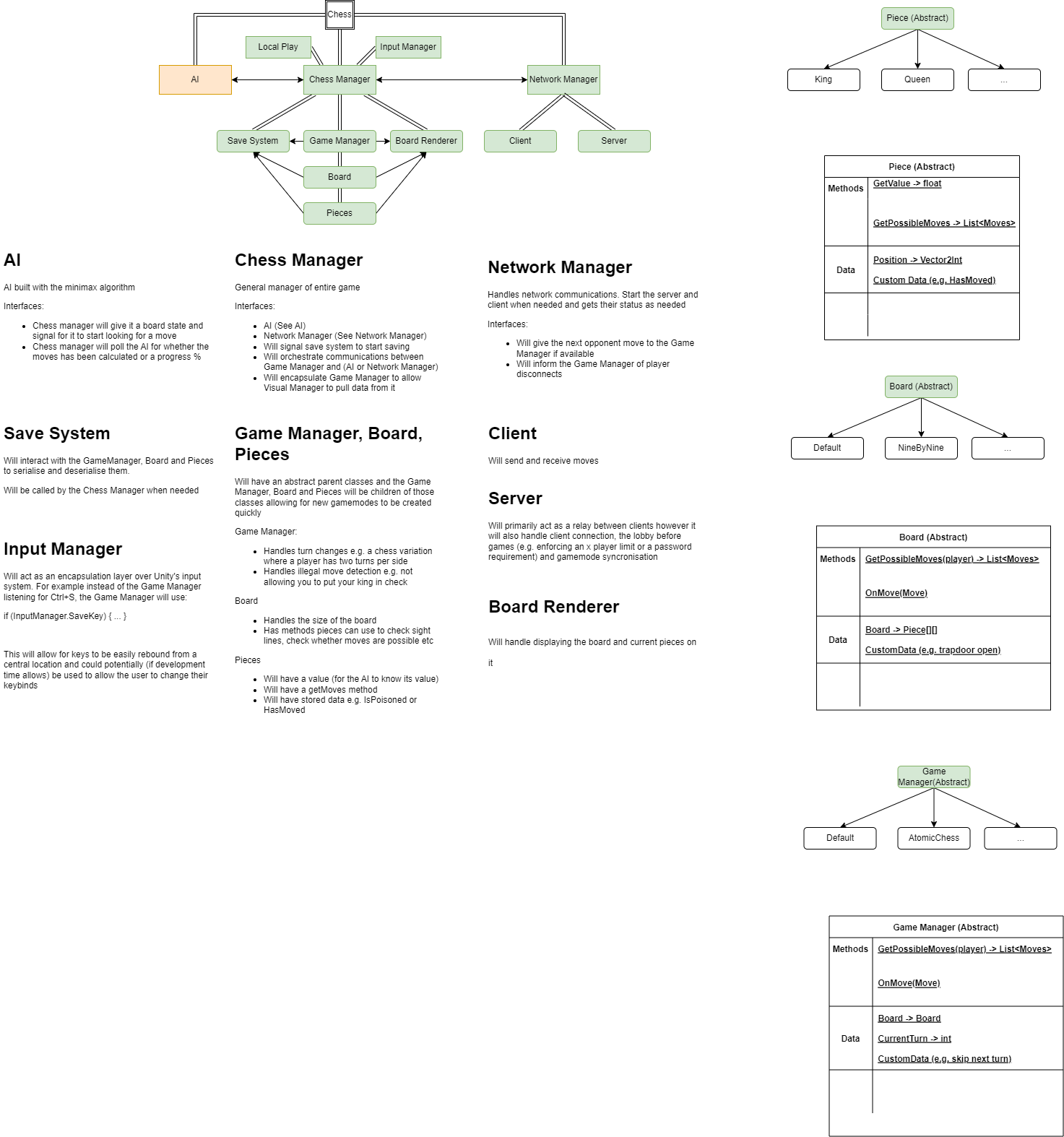
^ Code creating a new file based on the template.



^ Area where file is inserted into template

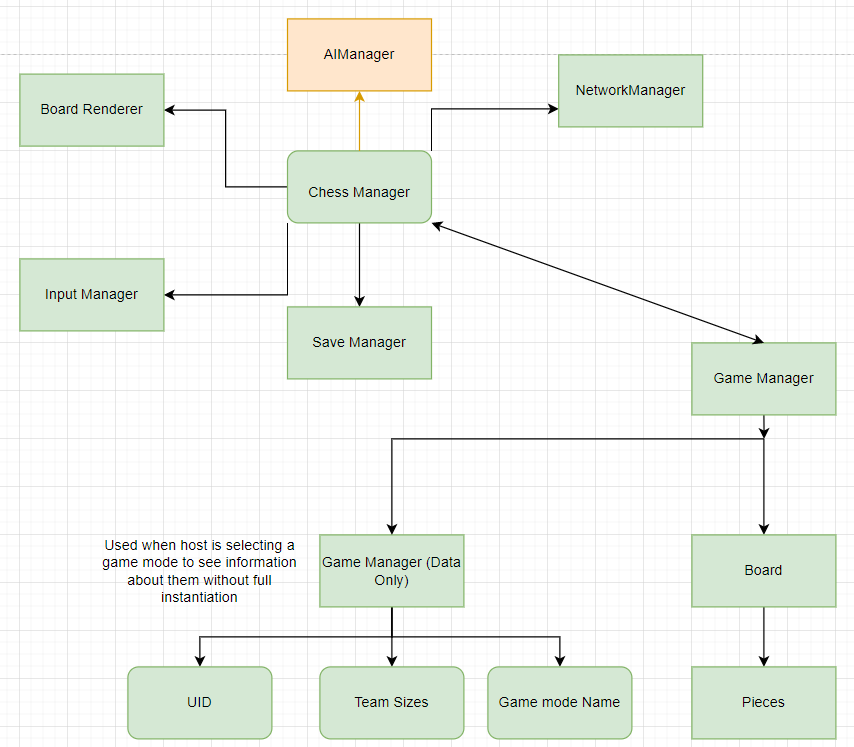
Design Flowcharts

# Overview

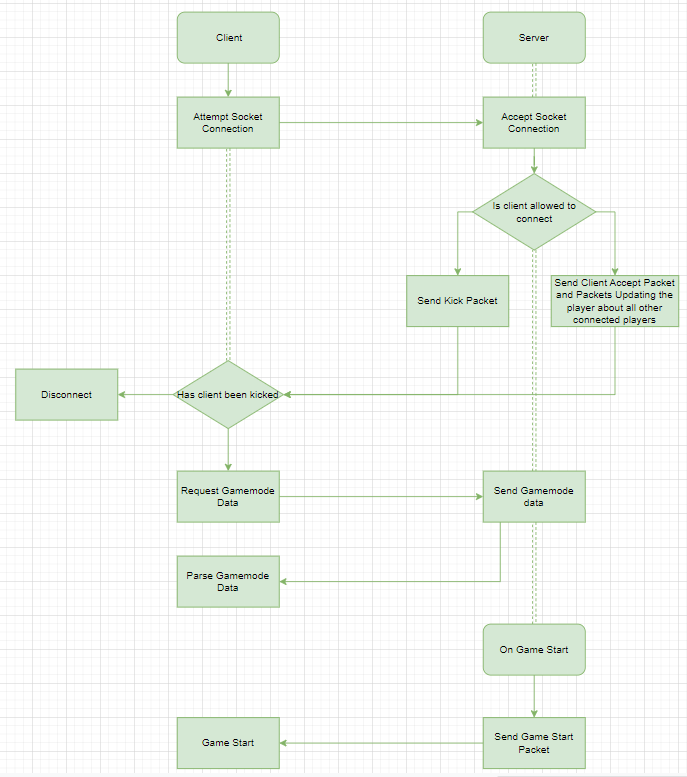


# Chess Manager

# Game Manager



# Pregame Flow



# Game Flow

