Draft 2: Design Part 2 Course Work:

(New success criteria)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number** | **Feature name** | **Importance** | **Description** | **Justification** |
| 1 | Difficulty settings | essential | The user is able to select from at least 2 different difficulty settings before the game | Interview, this allows a greater range of casual players to play. |
| 2 | Validation client side | Essential | The program must continue to await the users move until they use the devised system to input a valid to and from square | is needed for chess program to function at all |
| 3 | Validation client side | Essential | When it isn’t the user’s turn, appropriate validation should be able to decide to ignore any of the user’s square clicks | is needed for chess program to function at all |
| 4 | Validation client side | Essential | When in check the user must not be able to make a move that results in them still being in check. This is a difficult part of the chess rules but essential to replicating a proper game. | is needed for chess program to function at all |
| 5 | Intuitive Move Input with dot markings | High priority | When the user clicks on a square that contains a piece they own, this should be registered in software and in the visuals of the GUI they see. They have completed the first part of the move by selecting the form square. Invalid from squares will be ignored. Once the user selects a piece to move, dots should indicate the legal moves that piece can make. Once a marked square is clicked the piece should move into it | Interview and research |
| 6 | Graphically show the move | Essential | Once the user has successfully inputted a from and to square, they should see the piece they tried to move complete the desired move. This includes the piece moving and any taken piece being removed form the board. In essence, the board must be visually updated after both the computer’s and user’s move |  |
| 7 | Computer Move output | Essential | Following each move by the user the computer must make a move shown by their piece moving | is needed for chess program to function at all |
| 8 | Highlighting | High priority | After the computer makes a move the squares that this piece moved form and to should be highlighted for a short time. This will help show the user what is happening by catching their eye. This should ensure that they don’t miss the computer’s move by | Research of chess programs |
| 9 | Chess engine mid game | Essential | To at least some extent, the chess engine should be able to select moves in the mid and early game that take the opponents pieces and provide a utility advantage. This means that even though none of the board states n moves ahead (where n is search depth) involve the game being over, it should still produce a good move. | is needed for chess program to function at all |
| 10 | Chess engine late game | Essential | In the late game, when given the opportunity, the chess engine must at least aim to win by checkmate in order to ensure it can close out the game | is needed for chess program to function at all |
| 11 | Alert after each move for check | Essential | When the user or the computer are in check, some kind of alert should be produced to show this. This will explain to the user why their legal moves are reduced (it is important information to convey) | Needed to create an adequate user interface (it is a necessary output) |
| 12 | Proper recognition of game over and its properties | Essential | The chess game that the user plays should end when the game is over. Additionally, it should be displayed to the user why the game is over. By this I mean the main outcomes such as “stalemate” and “checkmate”. Additionally, the program should be able to identify 2 different kinds of stalemate: No legal moves and not in check, and the game state has been repeated 3 times.  It must be recognised when the game is won and, the game must end (no more moves), it must be outputted who has won the game. | Needed to properly handle when a game is over (essential part of the user experience with my chess game) |
| 13 | Chess engine aware of different game over outcomes | Essential | The chess engine should be able to see when games are over as it looks ahead, it should be able to recognise wins and losses due to checkmate as well as both kinds of stalemate. It should give each one a score to appropriate guide its behaviour (e.g. aim for checkmate, avoid stalemate unless loosing etc) | This is needed to ensure that the move engine models a human opponent in order to avoid irrational moves and to create the best chess games for the user |
| 14 | Undo button for last user move | Low priority | The user should be able to undo their last move and make a different one, then continue playing the game | Interview |
| 15 | Log of history of moves in game | Ideal / desirable | When a move is made it should be added to a list of the moves in the game for later reference by the user: e.g. “white bishop from F4 to G5” | Interview and research of online chess programs |
| 16 | Step back through log | Low priority | The ability to see the board state before a certain move was made in order to see what has happened in the last few turns | Interview and research of online chess programs |
| 17 | Fast and reliable connection between front and backend | Essential | The API interface between the frontend and backend must be robust and it must be efficient in order to not compromise the websites performance or cause functionality to fail | Essential in order for a web interface to work |
| 18 | Allowing users to login & sign up | Low priority | Allow users to create an account to allow access to other features | Research into other chess sites |
| 19 | Reloading unfinished game | Ideal / Desirable | Allow a logged in user to continue a game from where they left of it they login again | Interview |
| 20 | Puzzles | Low priority | Allow users to engage in puzzles to try to see the best moves in some preloaded game state | Interview |
| 21 | Draughts | Last add, very low priority | Allow users to play draughts vs the computer | N/A low importance |
| 22 | Allow users to start with a custom game layout | Low priority | For example users could start a game with a second queen or deprive the ai opponent of a queen | interview |
| 23 | Leader board and trophy system | **After interview I no longer want to implement this (lowest possible priority)** | Logged in users can earn trophies by winning games and completing challenges. Players are on a public leader board by trophies.  The justification for not doing this is that is has a high cost due to its complexity (I must have a user management and database system as well as a multipage website). Additionally this feature is less likely to be appreciated by casual chess players as they are less competitive. This means that it is not the most suitable feature considering my users. |  |
| 24 | Accessibility through website | Essential | This will make the end product more user friendly to older users.  I would define this a factor which leads to decisions to maximise usability for those who are not “tech savvy” or are differently abled. This includes making that all the text and widgets are readable (appropriate colours and sizes) | Thinking about stakeholders needs |
| 25 | Mobile access | Desirable | I could make the website work on the aspect ratio of a phone. This makes the program more accessible for those on the go | Thinking about stakeholders needs (e.g. the use of an IPAD) |
| 26 | Efficient adversarial AI | Highly desirable | The AI which the user plays against should use efficient algorithms so that a reasonable quality of move can be produced in a reasonable response time (minimise he user waiting) | Needed to create a positive user experience when playing against the AI and to differentiate the different difficulty levels. |
| 27 | The AI adversary feels as organic and real as possible | Highly desirable | It should change or learn over time or make purposeful mistakes in order to remain unpredictable and keep games fresh and challenging | Needed to create a positive user experience when playing against the AI |
| 28 | Validation of users move | Essential | The user should not be able to input an invalid move (an illegal move) | Essential to have a proper chess game |
| 29 | Valid computer move | Essential | The computers move should not be able to make an invalid move | Essential to have a proper chess game |
| 30 | Pieces Taken output | Desirable | The use should be able to see at a glance which pieces have been taken form both black and white to allow them to see who it winning. | My research of similar sites showed that this was standard practice |
| 31 | Restart game button | Essential | A button should allow the user to reset the chess game in order to play a new game form scratch or play again after the game has ended | My research of similar sites showed that this was standard practice |
| 32 | Concede game button | Essential | There should be a button for conceding the game. This will allow the user to still see all the information relating to the game (piece layout, pieces taken, move history) without the game continuing | My research of similar sites showed that this was standard practice |

Optional Further Problem Analysis:

Here I intend to explore the problems that must be tackles in a more technical sense to flesh out the specific nature of the problems to be tackled:

Minimax:

* British Museum not possible
* Use heuristic to make the problem tractable

I thought it would be prudent to investigate the technical aspects and details of the problem of making a computer that can play chess. Without doing this I will struggle to decide the appropriate algorithms to use as well as the limitations of my solution.

Chess is a game with complete information as there are no hidden zones (e.g. hands in poker). This is important as it means that both players can use game theory and assume that the opponent is rational in order to try to predict how the opponent will react to each of the moves that they are considering. This thinking ahead is what human players do to identify the best move.

Chess is a game that uses 6 types of pieces:

* Pawn
* Rook,
* Knight
* Bishop
* King
* Queen

Each of these pieces come in 2 varieties: black and white. I will need to incorporate this into my program, both to mark which player owns and controls each piece as well as to properly display the chess board to the end user.

A game of chess is played on a board that is 8\*8 and starts with each player having 16 pieces.

Most chess moves obey a set of rules that can be effectively modelled with vectors. Each chess piece can be attributed a position vector and then a series of movement vector if the chess board is treaded as a vector space. This is a useful abstraction as it provides a mathematical and easy to program way of describing how a piece can move from wherever they are.

Each piece has a set of movement vectors by which it can move. Ignoring the effect of check on legal the legal moves of a given piece for now, each piece can make usually perform each of these moves so long as the square that the piece would move to is empty or contains an enemy piece and so long as this vector doesn’t cause the piece to jump over an allied piece. ­­­­

This is not always true as there are exceptions. For instance the knight can jump over pieces, in addition the pawn if able to move forward 2 places on its first move only and takes pieces in a different direction to how it moves.

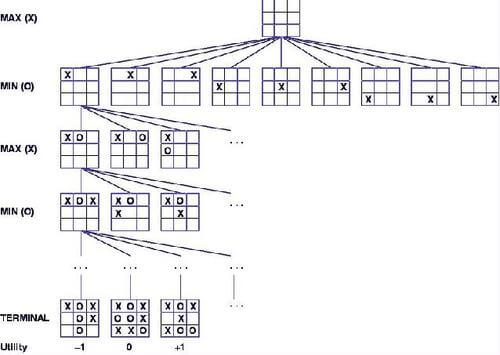
It is a useful generalisation to say that a chess move causes the square where the moving piece was to be empty and the square which it is moving to, to now contain the new piece. Assuming that every move has this simple structure and that no other squares are affected is an oversimplification but makes the problem easier to tackle

As such I don’t aim to add some more complex moves which violate this general idea of a move. These include:

* En passant
* Castling
* Ascension (a pawn becoming a queen)

In addition to analysing the problem of representing a chess game, I must try to investigate how a computer could play chess. In 1913 a man named Ernst Zermelo said: “As long as we are able to describe a game as a finite tree, we can find the optimal play”. Describing a game as a tree is a useful abstraction for an algorithm as the current game state can be

A game like tic tac toe can easily be described as a finite tree, if the user goes first then then this narrows down the number of possible games form 9! (360,000) to 8! (40,000).

****

**(diagram source:** [**https://www.hackerearth.com/blog/developers/minimax-algorithm-alpha-beta-pruning/**](https://www.hackerearth.com/blog/developers/minimax-algorithm-alpha-beta-pruning/) **)**

This means that all the possible games that could result from a tic tac toe game can be described as a small finite tree that can easily be searched in its entirety by the computer. This process is a recursive and involves exploring the decision tree down to the terminal nodes (games that are won, lost or drawn). These nodes are then given a utility and the algorithm propagates back up the tree. By assuming that the user will act rationally it can take the move that maximises the best outcome that it can guarantee even if the user plays optimally. This brute force process which explores all the paths a game could take in its entirety is called the British Museum Algorithm

This will not work for chess. There are around 10123 possible games of chess. This number is far to large for any super computer to apply the British Museum Algorithm. For context there are only 1080 atoms in the universe. This means that if every atom in the universe acted as a chess computer and processes chess board states at a rate of 1 per millisecond since the Big Bang, only 1 100 trillionth of the work (10-14) would have been completed today. This highlights the issue with the exponential nature of possible chess games. Due to this combinatorial explosion, the problem is in intractable.

As a result of this, I will discuss a more feasible algorithm that can tackle this problem later in this document called the minimax algorithm.

Design Objectives:

These should be achievable and measurable goals and technical requirements of the solution you devise in order to meet your success criteria

Better to have many singular and specific

Here are my design objectives. These are based on my success criteria and should serve as a series of specific technical requirements for my solution in order to meet the success criteria.

As the size of the problem is significant, I have focused on the aspects of the success criteria that are of a high priority or are essential. This should ensure that a high quality product is produced at the end of the iterative development process that is realistic considering time constraints.

|  |  |  |  |
| --- | --- | --- | --- |
| Number: | Name | Description | Success Criteria Link |
| 1 | GUI: Chess board | I need a graphical user interface that includes a visual representation of a chess board | 2-8 inclusive |
| 2 | GUI: Chess Board | This chess board should be able to register click events and run a handler function in order to allow the user input their move | 2-8 inclusive |
| 3 | GUI: Chess Board | The square click handler function should be able to keep track of the state of the chess board with some key variables in order to decide how to action the users click input | 2-8 inclusive |
| 4 | GUI: chess board | The program should do nothing when an invalid square is clicked. Relevant validation should decide to disregard the click event rather than raise an error. | 2-8 inclusive |
| 5 | GUI: Chess Board | The chess board should be able to respond to inputs as appropriate by visually changing how it appears to the user to add highlighting and move pieces. I will aim to ensure that ever click causes a reaction (visual feedback) | 2-8 inclusive |
| 6 | GUI: Chess Board | The chess board should be able to display any layout of pieces overlayed onto the squares using either characters or images to show the pieces | 2-8 inclusive |
| 7 | GUI: Chess Board | The chess board and the pieces must be sufficiently different in color so that white pieces can be seen on a black background | 2-8 inclusive |
| 8 | GUI: Chess Board | The chess board should be able to show the computers move but adding highlighting and then moving the pieces. | 2-8 inclusive |
| 9 | GUI: Difficulty Radio Buttons | Radio buttons should be used for the difficulty with the medium button already selected. This will ensure that only valid inputs for the difficulty are given. It will also ensure that at all times, at least one input is given. | 1 |
| 10 | Difficulty Of AI | The move engine (AI adversary) should be able to be limited in how well it can explore the board (measured in either depth or time) to create different difficulty levels. | 1 |
| 11 | Move Engine AI | The move engine should be able to perform a type of tree search to determine scores of child board states and use these scores to decide and return both a move and a score | 9, 10 |
| 12 | Move Engine AI | The move engine should feature some optimisations to allow it produce the same quality of move faster or produce a better move in the same time. The new optimisations should not change the output of a given search (this is deterministic) but should just improve speed | 26 |
| 13 | Move Engine AI | The move engine should be able to learn from its past games in order to improve its decisions and select new moves. | 27 |
| 14 | Move Engine AI | Persistent data storage should be used to enable the AI to retain calculations and cache results from previous games in order to improve and learn. | 27 |
| 15 | Move Engine AI | The move engine should be able to make a good move decision even if none of the terminal nodes are game which are over by using heuristics. | 9 |
| 16 | Move Engine AI | The move engine should be able to understand when a given node represents a board state where the game is over and value this node accordingly. In the late game it should aim to try to steer towards outcomes where it wins. | 10 |
| 17 | GUI: Main Title | Above the chess board there should be a main title which displays a message which changes over time. This should be in large font. It will indicate whose go it is and whether or not a given player is in check. This should also act as the prompt for users to input another move, to make moving a clearer process.  This title will also be used to explain when the game is over and who has won. | 5, 11, 12 |
| 18 | Board State Analysis | The chess engine should be able to determine is a given player is in check. This will affect legal moves and therefore the move engine as well as validation of the users move | 11, 28, 29 |
| 19 | Board State Analysis | The chess engine should be able to determine if the game is over, and the nature of the end to the game (e.g. is it a stalemate or who the winner is) | 12, 28, 29 |
| 20 | Board State Analysis | The chess engine should also be able to accurately determine all the legal moves available to a player for a given board state. This functionality will be relied on to validate the user’s move input and for the move engine’s search of the decision tree. | 28, 29 |
| 21 | Move engine can access board state analysis | The move engine (adversarial AI) should be able to able to access functions to recognise the legal moves of a given board state or whether or not the game is over in order to properly construct and traverse it’s the decision tree. | 9, 10, 26, 29 |
| 22 | Chess game: A data structure should be used to display previous moves | An array should be used to hold as vectors all the previous moves made in the game. | 15 |
| 23 | GUI: A widget to output previous moves | A table should be used to display the previous moves in plain English an using chess squares (not vectors). | 15 |
| 24 | GUI: buttons | I will need buttons on my user interface to concede and restart the game as well as functions to facilitate this | 31, 32 |
| 25 | GUI: Pieces taken function and widget | A function should be able to use the piece layout for a chess board state in order to determine which piece are missing (have been taken). There should be a widget in the user interface to provide this as a graphical output to the user. | 30 |
| 26 | API connection | I should have a fast and efficient way to convey data between the server and the client to allow for real time inputs and outputs and to prevent lags. The connection structure should also allow all the relevant data to be exchanged while not being too complex. | 17 |
| 27 | GUI: Reloading unfinished chess games | I have decided that this feature can be completed without a login system. When loading up, the website should make a request to the server for the up to data chess game rather than load data for a starting chess game. This should allow the user to continue a chess game where they left off if they close and then reopen the tab | 19 |
| 28 | Reloading unfinished chess games | The server should be able to read the cookies within a browser when client server session starts. If a unfinished game exists in the cookie (or the cookie contains an id to lookup in a database) then this game will be loaded into memory for the duration of the client and servers session.  When the session ends, a cookie should be created in the users browser that contains or links to the unfinished game for future use by the server. | 19 |
| 29 | GUI: Mobile device access | The GUI should be designed so that the core components (main title and chess board) can be correctly displayed in a variety of aspect ratios and orientations to allow for the use of tablets and phones (maximise accessibility and convenience) | 25 |
| 30 | GUI Accessibility | The website will aim to practise clarity and accessibility of the content over visual appeal when deciding colour schemes and layouts.  This means that there should be sufficient color contrast to identify individual widgets and text from the background as well as highlighting and pieces on the board. This includes ensuring that the colours used for highlight are not a common color pair for color blindness. | ­­24 |
| 31 | Move Engine Heuristics | The move engine (adversarial AI) should be able to perform a fast heuristic analysis of the board. This will provide a score that one player aims to minimise and the other maximise. It should be based on pieces remaining and piece layout. It will be used by the move engine to play chess in the early and mid-game. | 9 |
|  |  |  |  |

Proposed Solution Description:

Here I will describe the proposed solution by use of a very macro decomposition of design objectives. I will describe what general components are needed and how they will interact.

I will describe the benefits of this macro decomposition and I will designate how each macro component will solve a subset of the design specifications.

I will also describe the responsibilities of each of these components in terms of INPUTS, PROCESSES and OUTPUTS.

My proposed solution will be a full stack web application. It will have this overall structure:

­­Diagram

Description automatically generated

As shown in the above diagram, the solution is a full stack web application. This means that it includes code and logic at both the client and server side.

Front End:

The front end or client side component of the solution will be a combination of HTML, CSS and JavaScript. It will be responsible taking in user inputs and providing outputs. This should be done through a purely graphical user interface (GUI). The client side JavaScript should also perform client side validation of the users inputs in order to ensure that only legal moves are made. The client side JavaScript should also use asynchronous programming by using JavaScript promises to connect to the server and exchange data. The client side JavaScript code is intended to be “dumb”. By this I mean that it won’t have any algorithms that can help it understand what is happening in this game of chess (e.g. if the game is over or what the legal moves are). Instead it will request act only as a wrapper around the logic already created in the chess computer, providing a User Interface on top of this.

Server Side API Connection and Webserver:

The next layer in my application is the server’s webserver and API gateway. This server side code will setup the HTTP webserver that will send the relevant HTML, CSS and JS files to the browser. In addition, appropriate WebSocket routes will be created to allow the server and the client side chess GUI to communicate. Diagram

Description automatically generated

The webserver and API gateway should act as an intermediary between the client side JavaScript and the functions of the chess engine. It should include server side validation to ensure that the data sent to the server by the client is valid (so for instance the user move must be valid). As part of defensive design to minimise unexpected behaviour and validation needed I will aim to minimise he amount of data that the client and server exchange.

Chess Computer:

The chess engine will take the form of a server side python program that is able to keep track of a chess game, provide the legal moves, and recognise check and game over as well as determine the adversarial AI’s move. This will be completed by dividing the chess computer into various sub programs.

Benefit of Decomposition:

Graphical user interface, text, application, chat or text message

Description automatically generated

The benefit of this decomposition of these main parts is that each part can be tackled separately and using different means.

* The chess computer can be tackled with pure python
* The client side user interface can be tackled with HTML and JavaScript
* The API connection and webserver can be tackled using the Flask and WebSocket frameworks

Dividing up the program into these parts allows for specialist languages and frameworks to be used to tackle the sub-problem. Additionally different people could work on different sub problems independently if this was a professional project. Because of this, the above decomposition makes the final product easier to implement and should improve its quality in terms of how well it meets the design goals.

Diagrams of propose solution:

* Data flow diagram
* Menu flow diagram
* Limitations of solution proposed (hardware and software)

Here is a flowchart to describe how the user will input a move and play chess against the computer.

This will all take place on the same page as the application is only a single web page (reduces complex menus that the elderly can struggle to navigate):

Diagram

Description automatically generated

And here is a data flow diagram to describe the flow of data within the program when making a move. It shows the layers of the application that sit between the GUI and the database. It shows how a user move input is processed and then a computer move output is produced.

Diagram

Description automatically generated

Decomposition of Chess Computer (not algorithms just functions, classes and data structures)

Link back to design requirements

The chess computer is a broad term that I am using to describe the backend logic that runs on the server to allow it to manage a game of chess, including generating the computer’s moves.

The chess engine is a large program that can be further decomposed into multiple modules.

Graphical user interface, application

Description automatically generated

The Chess Functions Library or Chess Engine us used to analyse a given chess board state. It will contain functions to use a given board state to determine the possible legal moves, or if the game is over.

The Move Engine component will be responsible for performing a minimax search of a given board state in order to determine a suitable move for the computer to make. As part of this it may interact with the database in order to make use of caching to improve efficiency.

The Game Manager module will act as a wrapper around all of the logic and functionality of the other 2 modules. It will be able to keep track of a whole chess game where as the chess engine can only keep track of a single board state which is a snapshot of a whole game. It will be able to keep track of which players go it is as well as implement moves. It will include validation to ensure that only the right player is able to move on there turn and that they can only make a legal move.

It will make use of the Chess Engine module to keep track of the board state throughout the game. It will use the Move Engine module to decide the move of the adversarial AI that the user will play against. It may also make use of a database in order to save and restore games for the user.

This means that:

* The Chess Engine module has no dependency on the other modules
* The Move Engine module is dependant on only the chess engine module
* The Game Manager module is dependant on both of these modules. It is the way in which the functionality of the other modules is accessed and choreographed together to produce useful functionality.

A screenshot of a computer

Description automatically generated with low confidence

This decomposition should allow me to more easily achieve my design objectives as a big problem is being tackled with multiple separate sub components with their own responsibilities.

Additionally, the use of a Game Manager module should provide abstraction as it allows other components that facilitate a chess game to access shared logic rather than each interacting with board states and keeping track of turns individually. This should make other components such as the Server Side API Handlers easier to implement as less logic will need to be repeated. In addition the Game Manager module can be tested to show that it is reliable whereas lots of different instances of similar code to facilitate a chess game cannot be easily tested.

Each section should be able to achieve the following design criteria (by numbers):

* Chess Engine: 18, 19, 20
* Move Engine: 10, 11, 12, 13, 14, 15, 16, 21, 31
* Game Manager: 22, 18. Also ties in with requirements around game: (part of objectives 4, 5, 6, 7, 80

Decomposition of GUI logic and API client server connection:

This component will include some logic on both the server and client side. Logic on the sever side will facilitate the API connection between the client and the server. Relevant handler function will call the appropriate functions of the Game Manager module when events are received from the client.

The majority of the code will be client side. It will include HTML and CSS to define the relevant components (tags) and there layout / style on the webpage.

JavaScript will also be used to provide interactivity, logic and connect to the server:

* It will include logic to affect the DOM object. The Document Object Model is an object in JavaScript that allows for the HML content that the browser renders to be interacted with. This will allow inputs to be received and outputs to be graphically displayed.
* JavaScript will also be used to create a class to represent all the data and logic from a given board state. This will include all the logic to decide how user inputs will be processed, when server requests will be made and how the returned value will be outputted.
* JavaScript will also be used to request data from the server. For example this could allow for a user move to be implemented and the new board data to be used to update the board object and the front end. This will involve the use of a framework like WebSocket that will provide an API: Application Program Interface with which the client and server can exchange data.

Graphical user interface, text, application

Description automatically generated

Each component should be integrated together to achieve the specified design objectives.

For instance, when a user clicks a piece, the DOM interaction components should register the square click and then send he data about which square is clicked to the Board State Class. This will decide what output are appropriate.

For example it could decide to change the highlighting in response. The DOM interaction components would then use a corresponding function to update the visual highlighting colours that the user sees.

Alternatively that Board State class could decide that the user has inputted a valid move. If so both client and server side API handlers would be used to communicate this input to the Game Manager module in the Chess Computer. Then the new board state and all its relevant data would be sent back to the client side Board State class vie both client and server side API handlers. The new data contained within the Board State class (e.g. a moved piece) would the be provided as a visual output to the user using the DOM interaction component to affect the HTML.

HTML and CSS will be used to dictate the content of the webpage (the tags that JavaScript will interact with) and its layout. I will keep styling to a minimum and focus instead on the interfaces usability. I will also use CSS to ensure that all the text is big enough and clear to read. The layout of the webpage, as defined by CSS, should also be such that it should be compatible with PCs, tablets and mobile phones.

Link back to design requirements

Each module will be used together to achieve the design objectives that are relevant to the GUI. Here are the design objectives that this component should be able to achieve:

1, 2, 3, 4, 5, 6, 7, 8, 9, 17, 23, 24, 25, 26, 27, 29, 30.

Chess Computer: All major algorithms

Discuss the use of layers of abstraction in the use of classes

Emphasize:

* Board state class (layers of abstraction)
* Game class (use of database)
* Minimax function (optimisations)
* Add flowcharts and pseudocode
* Ensure you explain how this helps meet a design specification

The Chess Engine / Chess Functions:

The Chess Engine component should be able to make various inference to determine information about a chess board. It should also be possible to implement moves to get a new resultant board state. This will be needed in the minimax search completed by the Move Engine as well as to implement user’s moves.

Here is a further decomposition of the Chess Engine to identify its functionality and properties.

Diagram

Description automatically generated

Due to the nature of the problem, there are various reasons why OOP (object oriented programming) is needed. These include:

* I need to keep track of many different board states at once in a minimax search
* I have already grouped the responsibilities of the module into variables and function that use these variables. This can be easily translated to properties of a Board\_State class and method that reference these properties
* The module will be easier to develop and test if OOP is used as variables and methods form different board states will be grouped withing self-contained objects and so will not interfere with each other (as global variables might).

So here is the design for a class that will possess this functionality:

Text

Description automatically generated with medium confidence

Due to the fact that I need to keep track of multiple different board states and child board states at once when performing a minimax search, I decided that it would be best to make the objects produces by the Board\_State class immutable. This means that the make\_move will return a new child board state object rather than mutate the existing object. This prevents me needing to make copies of the board state before making moves whenever I want to preserve the original board state. It also prevents any properties being set erroneously so long as the make\_move and constructor methods are used correctly.

The pieces\_matrix variable will be a 2 dimensional array of pieces to represent the board layout. It can be accessed by a position vector by converting the position vector to a row and column in the 2d array:

Row = 7- vector.j

Column = vector.i

Piece = pieces\_matrix[row][column]

The most important function in this module is the generate legal moves function which is essential for determining if the game is over (no legal moves) as well as performing the minimax search and validating the users move.

It returns an array of moves. Each move is encoded as a position vector (for the piece moving) and a movement vector (the direction is moves in).

In order to represent these vectors, I will create a vector class which will include both the vector’s data: integers for the i and j components as well as functions relevant to vectors.

Here is the pseudocode and UML diagram for my vector class:

FUNCTION sqrt(x):

    RETURN x\*\*(1/2)

END FUNCTOIN

FUNCTION square(x):

    RETURN x\*\*2

END FUNCTOIN

CLASS Vector():

    // 2d vector has properties i and j

    i: int

    j: int

    // code to allow for  +  -  and  \*  operators to be used with vectors

    FUNCTION add\_vectors(self, other):

        ASSERT isinstance(other, Vector), "both objects must be instances of the Vector class"

        RETURN Vector(

            i=self.i + other.i,

            j=self.j + other.j

        )

    END FUNCTOIN

    FUNCTION subtract\_vectors(self, other):

        ASSERT isinstance(other, Vector), "both objects must be instances of the Vector class"

        RETURN Vector(

            i=self.i - other.i,

            j=self.j - other.j

        )

    END FUNCTOIN

    FUNCTION multiply\_vectors(self, multiplier: int | float):

        RETURN Vector(

            i=int(self.i \* multiplier),

            j=int(self.j \* multiplier)

        )

    END FUNCTOIN

    // check if a vector is in board

    FUNCTION in\_board(self):

        // Assumes that the current represented vector is a position vector

        // checks if it points to a square that isn't in the chess board

        RETURN self.i in range(8) and self.j in range(8)

    END FUNCTOIN

    // alternative way to create instance, construct from chess square

    @classmethod

    FUNCTION construct\_from\_square(This\_Class, to\_sqr):

        // Example from and to squares are A3 -> v(0, 2) and to B4 -> v(1, 3)

        to\_sqr = to\_sqr.upper()

        letter, number = to\_sqr

        // map letters and numbers to 0 to 7 and create new vector object

        RETURN This\_Class(

            i=ord(letter.upper()) - ord("A"),

            j=int(number)-1

        )

    END FUNCTOIN

    // this function is the reverse and converts a position vector to a square

    FUNCTION to\_square(self) -> str:

        letter = chr(self.i + ord("A"))

        number = self.j+1

        RETURN f"{letter}{number}"

    END FUNCTOIN

    FUNCTION magnitude\_and\_unit\_vector(self):

        magnitude: float = sqrt(square(self.i) + square(self.j))

        // unit\_vector: Vector = self.\_\_mul\_\_(1/magnitude)

        // RETURN magnitude, unit\_vector

        RETURN magnitude

    END FUNCTOIN

END CLASS

Text

Description automatically generated

I will use Vector objects to describe the position of each piece and how it is moving. These objects are used as parameters for some methods like make\_move and as return values for generate\_legal\_moves.

In order to implement the static\_evaluation and generate\_legal\_moves functions I am going to further decompose the problem by using a pieces class.

I will aim to make a class for each chess piece. The resulting objects will be able to represent both black and white pieces. Each object will have 2 main responsibilities:

* Given where it is on the board, determine its own utility
* Given where it is on the board, ignoring check, determine the movement vectors by which it can move.

Each piece object will contain a starting utility and a 8\*8 matrix of additional utility. Depending on where it is on the board, the value matrix may increase or decrease the pieces utility. The value matrix is meant to describe how some pieces are move valuable if they are in a certain area of the board (e.g. knight near the centre of the board). The sum of all these utility values will be used by the board state class to find the static evaluation of the board as a whole.

In addition each piece will contain a function that will return a list of movement vectors. Each piece will use the pieces matrix (board positions) and its position vector (where it is) to identify where it can move to. All the different piece will have different ways of doing this:

* For instance the knight will simply need to check the contents of the square it would move to for each of its 8 possible vectors. The king is similar
* The pawn needs to use rules around the contents of its neighbouring squares (e.g. are they empty of do they contain an enemy) to decide if it can move forward or diagonally. It additionally may need to overload the constructor of the pieces class to include a flag to determine if it has moved yet. This will affect it can move forward 2
* Pieces like the rook, bishop and queen must iteratively test increasing length vectors in a given direction until a non-empty square is hit. Then they cannot go any further in that direction and should start iteratively checking vectors in other directions.

Due to the fact that each piece contains some logic that is different (where it can move) and some logic that is shared (calculating its utility score) I have decided to use parent classes. All logic that is generic and shared will be implemented in a parent Piece class. In addition, an abstract class called Abstract\_Piece\_Interface will be used to ensure that every child piece has the relevant methods and properties.

By having the Piece class Inherit from the Abstract\_Piece\_Interface class and by having each individual piece class (e.g. Pawn) inherit form Piece I will ensure that:

* I keep repeated logic to a minimum
* Every piece class has the same external interface and so can be used in the same way.

Here the whole UML diagram for these classes:

Graphical user interface

Description automatically generated

This is too small to see but should illustrate the structure I am trying to show. I will now show 3 sub-diagrams to make the details easier to read:

Graphical user interface, application

Description automatically generatedGraphical user interface

Description automatically generated

Graphical user interface

Description automatically generated

Using the abstract base class will mean that an error will be raised if any child class is initialised without all the necessary attributes and methods. This should ensure that the interface between the classes is that same and should help prevent logic errors later.

A picture containing text, sign, outdoor

Description automatically generated

In addition, the use of the pieces an vector sub components allows me to further break down the problem through layer of abstraction. At the bottom is the Vector class, this is at the lowest level and is performing individual vector calculations. This module can be tested to ensure it is robust. Then the Pieces component can abstract all that logic and simple deal with vectors to try to determine the moves a given piece can make. Then at the highest level, all of the previous detail is abstracted. All the board state class needs to do is collate all of these individual pieces moves together into a big array and filter it to remove child board states that cause check. This reduces the complexity of a function like generate legal moves within the board state class. This is because the details for each which way each piece can move are abstracted as decomposition has been used to create a subcomponent to solve this issue.

This should make writing the otherwise tricky legal moves and static evaluation functions easy. All the other functions in the Board\_State class should either be easy to write (e.g. get piece at vector) or should able to make use of the generate legal moves function (e.g. the is\_game\_over function).

The Move Engine algorithm:

The move engine should act as a large and complex function that is composed of many sub functions. It takes as input a board state and returns as output a best move and a score for how good that move is estimated to be.

As discussed earlier, I cannot perform the British Museum Algorithm on a chess decision tree as the number of computations needed is so high that the problem is intractable. Instead I will use a recursive function called the minimax function to explore the decision tree to a given depth, use heuristics to estimate each players advantage at the terminal nodes and then back propagate to find the best move. I am calling the tree that I am using to represent the possible chess games form a given board state a decision tree as each node represents the decision of which move to pick.

Here is an example of this process. I have here a decision tree which I created to represent the problem with all the details of chess abstracted. The tree has been explored to a given depth using the legal moves of parent board states. Then the terminal nodes have been given a score:



(source: <https://en.wikipedia.org/wiki/Minimax>)

We can see that in the above decision tree we have simplified the problem by illustrating each node as only having 2 children.

The terminal nodes are given a score, this could be infinity is the game is over or a number estimating how favourable the game is using a heuristic. One player aims to maximise this score and the other aims to minimise it. In this way, chess is a zero-sum game as one players gains are another players losses. We can see that all chess board where the maximising player can decide a move are represented with circular nodes and the minimisers moves are represented with square nodes.

As complete information is assumed, the idea of picking a move instead becomes deciding which of the child board states is most favourable for you to create. Because of this, we can see that at depth 3, the values assigned to the rectangular nodes are the lowest scores of the child nodes. This is because these are the moves that are best for the maximiser.

Because the assumption is made that each player is rational and as there is complete information, the maximising player can anticipate that when presented with these board states, the minimising player will act in these ways. As a result, the maximising player at depth 2 can attribute nodes at depth 3 with the value that the minimising player will rationally pick. Then the maximising player picks the highest score among these nodes.

In doing this, the final score and move is the best that can be guaranteed, assuming the opponent plays optimally. This means that a when a player makes a sub optimal play, this leads to a better outcome than expected for the opponent.

In my program, the static evaluation function from the Chess Engine module will be used to provide the heuristic that will evaluate the terminal nodes. The computer will be the minimiser and the user will be the maximiser.

This algorithm will allow me to tackle the game of chess as it provide a heuristic solution that is good enough. The algorithms is still slow as it has an exponential time complexity. We can define the branching factor at a given stage in the game to be the number of legal moves available from a given board state and the depth to be the number of moves ahead that we look. Now the number of static evaluation of terminal nodes is proportional to the branching factor to the power of depth. This means that the function has an exponential time complexity O(2^n). This means that the solution algorithm cannot be used to a high depth as it has a non-polynomial time complexity.

I will add a series of optimisation to my program. The simplest of optimisation to add. It works to reduce redundant calculations by “pruning” parts of the tree to prevent unnecessary computation.

Here is a screenshot from the following video that illustrates this concept well: <https://www.youtube.com/watch?v=l-hh51ncgDI>

A picture containing timeline

Description automatically generated

The minimax function now uses 2 additional arguments that are passes on to recursive calls. Alpha represents the best that the maximiser can already achieve and beta represents the best that the minimiser can already achieve. Another way of looking at this is to say that alpha is what the maximising player can achieve without further exploration and beta is the best that the minimiser has to allow the maximiser to achieve. Because of this, if alpha is greater than or equal to beta, it means that further exploration of child nodes is not needed as the opponent minimiser would never let you the maximiser get to this path as it is already too good for them to allow.

This is shown in the above diagram. Nodes are ladled with their alpha and beta values. In this case, white is the maximiser and black is the minimiser. We can see from the inequalities on some nodes how the alpha beta pruning indicated that a certain branch of the tree can be “pruned” and no longer explored.

I also intend to add caching to improve the efficiency of my minimax algorithm. To do this I will store the parameters and return value of each minimax function call as an entry in a database (structure explained later). This will allow me to prevent redundant searches in future. Cached results could be retrieved from the database if they match the search being performed exactly or if they match a recursive function call within a search (e.g. cached depth 1 results could be used as part of a depth 2 search later).

To maximise efficiency I will not store the whole board state in my database, instead I will store its SHA256 hash as a hexadecimal string. Due the size of 2­­­­256­ I can make the safe assumption that there is a 0 chance of collision. Therefore each board state can be encoded as a small binary sting.

To write the result of a minmax search to the cache I will delete all previous cached results that are at a lesser depth and then add the new search to the database.

To retrieve data form the cache (for a search of a given board state at a given depth) I will hash the board state and then query the table for matching hashes. This will be efficient as the table will be indexed by hash. The depth of any stored records will then be checked, if it is insufficient then it will be ignored. Otherwise, if the cached search has a depth greater than or equal to the needed depth, the database entry will be returned.

Further optimisation that I intend to add to the algorithm to improve efficiency or efficacy include:

* Variable depth search. This would model the variable depth feature of Deep Blue by allowing certain parts of the tree to be searched to an additional depth if check is encountered. These branches are likely to be more important to search at they could help to cause or avoid checkmate.
* Pre-sorting the moves best to worst (using a heuristic or a recursive depth n-2 search) before performing a recursive minimax search of each move to improve pruning
* Concurrency: The minimax function can be made concurrent by breaking up the legal moves to examine into sub arrays. Then many threads could be used to concurrently perform minimax searches of these segments of the legal moves at once.
* Insurance policy + self-interrupting: This is the idea that a minimax call at a lesser depth takes exponentially less work. This means that it only takes around 1/30th more work to do a depth 1 then 2 then 3 search of a board state than to just start with a depth 3 search. This means that the algorithm could be structured to keep searching for a given amount of time and then stop itself and return the best result. Unfinished searches could be added to the cache allowing for additional search time in future to improve the quality of the result.

The implementation of the self-interpreting feature will allow me to create a minimax function that will search the decision tree for a given amount of time.

* This will allow for more than 2 difficulty settings: depth 1 and depth 2 as a greater depth will likely be infeasible in the context of the limited thinking time of a chess AI.
* This will remove the inconsistent thinking time of setting the AI to a given depth. This problem arises as the number of legal moves can vary greatly form 1 to 50. This can cause great variation in how long an minimax search to a given depth can take
* This idea of a certain amount of time to explore will also allow me to achieve my design objective (13 and 14) of making a learning adversarial AI. This will be possible by combining caching with a set exploration time. This will allow the AI to continue where it left of when it encounters a bord state again (like the opening positions) and search in greater depth to improve the quality of its move.

Here is the pseudocode for a basic minimax function. It will be able to calculate the computers best move from a given board state. It featured alpha beta pruning to improve its efficiency.

// default arguments allow for the function to be provided only the board state and be able to find the computer move

FUNCTION minimax\_search(board\_state: Board\_State, depth: 2, alpha=-infinity, beta=+infinity, is\_maximizer=False)

    // base case

    over, winner = board\_state.is\_game\_over\_for\_next\_to\_go()

    IF depth == 0 or over THEN

        // use static\_evaluation for score

        best\_score = board\_state.static\_evaluation()

        best\_move = None

        RETURN best\_score, best\_move

    END IF

    // recursive case

    // set initial variables

    best\_move = None

    IF is\_maximizer:

        best\_score = - infinity

    ELSE:

        best\_score = + infinity

    END IF

    // iterate through legal moves and get child board states

    legal\_moves = board\_state.generate\_legal\_moves()

    FOR EACH move in legal\_moves

        // get child board state

        from\_v, move\_v = move

        child\_board\_state = board\_state.make\_move(from\_v, move\_v)

        // recursive call to evaluate child board state

        child\_score, child\_move = minimax\_search(child, depth - 1, alpha, beta, not is\_maximizer)

        // depending on if player is maximizer affect if a lower or higher score is better

        // update alpha / beta and update best score and move variables as necessary

        IF is\_maximizer:

            alpha = max(alpha, child\_score)

            IF child\_score > best\_score:

                best\_score = child\_score

                best\_move = child\_move

            END IF

        ELSE:

            beta = min(beta, child\_score)

            IF child\_score < best\_score:

                best\_score = child\_score

                best\_move = child\_move

            END IF

        END IF

        // pruning, stop exploring more legal moves

        if beta <= alpha:

            break

    END FOR

    // return the score and move corresponding to the best child

    RETURN best\_score, best\_move

END FUNCTION

Game Manager:

This module is to be in charge of managing a chess game. It should be able to keep track of whose turn it is and include validation to ensure that only valid moves are made.

Due to the fact that a web server may have may sessions with clients at once, I will need to be able to store the data for multiple games at once. Additionally I want to be able to save all data relevant to the game as binary to be restored later. These factors mean that using global variables, that can be accessed by all function in the game manager module, to keep track of the game is not feasible. Instead I will again use OOP.

Here the game class that I will use when creating my console chess prototype:

Graphical user interface, text, application, email

Description automatically generated

And here is the game class that I will use to facilitate the chess games that are played on my webpage:

Graphical user interface, text, application

Description automatically generated

The main difference between them is that the website game class has the capacity to play at different difficulties. In addition it doesn’t need to build a table of data to give running commentary of the game in the console. Instead it creates a dictionary of data about the computers move that is sent to the webpage to allow for the computer’s move to be displayed.

The implement user move and computer move methods in both classes contain validation to ensure that it is that player’s turn to go. In addition, the user move is checked to ensure that it is a valid move.

It is also important that this object can be pickled as this will allow it to be serialized to binary data which can be stored in a database. The game object’s binary can then be read form the database and deserialized in order to reload the game for the user to finish.

GUI logic and API client server connection:

\* Discuss validation

\* Discuss how data is transferred (diagrams of API etc)

\* Use of database

* Add flowcharts and pseudocode
* Ensure you explain how this helps meet a design specification

As previously mentioned, I will use a system where data is exchanged between the client and server using the WebSocket framework.

Diagram

Description automatically generated

(diagram repeated here)

There will be 5 main WebSocket events that will be used to exchange data between the client and the server.

After the server updates it python Game\_Website object, a standardised block of JSON data will be sent to the client to provide the client all the data needed to update the GUI.

Here is an example of this json data for the start of a chess game (opening positions):

{"difficulty": "medium", "next\_to\_go": "W", "game\_over\_data": {"over": false, "winning\_player": null, "victory\_classification": null}, "legal\_moves": [[[0, 1], [0, 1]], [[0, 1], [0, 2]], [[1, 0], [-1, 2]], [[1, 0], [1, 2]], [[1, 1], [0, 1]], [[1, 1], [0, 2]], [[2, 1], [0, 1]], [[2, 1], [0, 2]], [[3, 1], [0, 1]], [[3, 1], [0, 2]], [[4, 1], [0, 1]], [[4, 1], [0, 2]], [[5, 1], [0, 1]], [[5, 1], [0, 2]], [[6, 0], [-1, 2]], [[6, 0], [1, 2]], [[6, 1], [0, 1]], [[6, 1], [0, 2]], [[7, 1], [0, 1]], [[7, 1], [0, 2]]], "pieces\_matrix": [[["B", "\u265c"], ["B", "\u265e"], ["B", "\u265d"], ["B", "\u265b"], ["B", "\u265a"], ["B", "\u265d"], ["B", "\u265e"], ["B", "\u265c"]], [["B", "\u265f\ufe0e"], ["B", "\u265f\ufe0e"], ["B", "\u265f\ufe0e"], ["B", "\u265f\ufe0e"], ["B", "\u265f\ufe0e"], ["B", "\u265f\ufe0e"], ["B", "\u265f\ufe0e"], ["B", "\u265f\ufe0e"]], [[null, null], [null, null], [null, null], [null, null], [null, null], [null, null], [null, null], [null, null]], [[null, null], [null, null], [null, null], [null, null], [null, null], [null, null], [null, null], [null, null]], [[null, null], [null, null], [null, null], [null, null], [null, null], [null, null], [null, null], [null, null]], [[null, null], [null, null], [null, null], [null, null], [null, null], [null, null], [null, null], [null, null]], [["W", "\u265f\ufe0e"], ["W", "\u265f\ufe0e"], ["W", "\u265f\ufe0e"], ["W", "\u265f\ufe0e"], ["W", "\u265f\ufe0e"], ["W", "\u265f\ufe0e"], ["W", "\u265f\ufe0e"], ["W", "\u265f\ufe0e"]], [["W", "\u265c"], ["W", "\u265e"], ["W", "\u265d"], ["W", "\u265b"], ["W", "\u265a"], ["W", "\u265d"], ["W", "\u265e"], ["W", "\u265c"]]], "check": false, "pieces\_taken": {"B": [], "W": []}, "move\_history": []}

Here we can see that many data points have been pre-calculated and provided to the client UI. This includes an array of legal moves as vector pairs and an 8\*8 2d array of the pieces matrix. Sub strings like this "\u265f\ufe0e" correspond to a chess pieces character that is not in the standard ASCII character set but instead in the larger UTF-8. To ensure compatibility, it has been encoded by the python json module so that the JSON is ASCII compatible as it is sent between the client and the server. JSON is a useful abstract data structure to use as modules exist to convert python objects to JSON and JSON to JavaScript objects. This means that this standard makes transferring data between the client and the server easier.

On the server side I will need the following functions handler functions to process data receives on each of these 5 events. These handler functions will cause some mutation to the game object held within the server client session and then return JSON data to update the client side about the game.

Here is the pseudocode for this:

FUNCTION generate\_game\_update\_data()

    game = get\_game\_in\_session()

    next\_to\_go = game.board\_state.next\_to\_go

    difficulty = game.difficulty

    legal\_moves = list(game.board\_state.generate\_legal\_moves())

    legal\_moves\_serialised = serialize\_legal\_moves(legal\_moves)

    pieces\_matrix = game.board\_state.pieces\_matrix

    pieces\_matrix\_serialised = serialize\_pieces\_matrix(pieces\_matrix)

    next\_to\_go\_in\_check = game.board\_state.color\_in\_check()

    over, winning\_player, victory\_classification = game.check\_game\_over()

    game\_over\_data =  {

        "over": over,

        "winning\_player": winning\_player,

        "victory\_classification": victory\_classification,

    }

    pieces\_missing = {}

    FOR EACH color IN ("B", "W")

        pieces\_missing[color] = list(map(

            serialize\_piece,

            game.board\_state.generate\_pieces\_taken\_by\_color(color)

        ))

    END FOR

    move\_history = game.move\_history\_output

    payload = {

        "difficulty": difficulty,

        "next\_to\_go":  next\_to\_go,

        "game\_over\_data":  game\_over\_data,

        "legal\_moves":  legal\_moves\_serialised,

        "pieces\_matrix": pieces\_matrix\_serialised,

        "check": next\_to\_go\_in\_check,

        "pieces\_taken": pieces\_missing,

        "move\_history": move\_history,

    }

    RETURN payload

END FUNCTION

@bind\_socket\_handler("get\_update")

FUNCTION get\_update()

    result = generate\_game\_update\_data()

    RETURN result

END FUNCTION

@bind\_socket\_handler("implement\_computer\_move")

FUNCTION implement\_computer\_move()

    game = get\_game\_in\_session()

    assert game.board\_state.next\_to\_go == "B"

    assert not game.board\_state.is\_game\_over\_for\_next\_to\_go()[0]

    move\_description = game.implement\_computer\_move\_and\_report()

    set\_game\_in\_session(game)

    outgoing\_payload = generate\_game\_update\_data()

    outgoing\_payload["computer\_move\_description"] = move\_description

    RETURN outgoing\_payload

END FUNCTION

@bind\_socket\_handler("reset\_game", respond=False)

FUNCTION reset\_game()

    old\_game = get\_game\_in\_session()

    difficulty = old\_game.difficulty

    new\_game = Game\_Website(difficulty=difficulty)

    set\_game\_in\_session(new\_game)

END FUNCTION

@bind\_socket\_handler("implement\_user\_move")

FUNCTION implement\_user\_move(incoming\_payload)

    game = get\_game\_in\_session()

    assert game.board\_state.next\_to\_go == "W"

    assert not game.board\_state.is\_game\_over\_for\_next\_to\_go()[0]

    user\_move = tuple(

        deserialize\_move(

            incoming\_payload["user\_move"]

        )

    )

    game.implement\_user\_move(user\_move)

    set\_game\_in\_session(game)

    RETURN generate\_game\_update\_data()

END FUNCTION

@bind\_socket\_handler("change\_difficulty", respond=False)

FUNCTION change\_difficulty(incoming\_payload)

    game = get\_game\_in\_session()

    new\_difficulty = incoming\_payload["new\_difficulty"]

    game.difficulty = new\_difficulty

    set\_game\_in\_session(game)

END FUNCTION

Graphical user interface, text, application

Description automatically generated

(diagram repeated)

The client side JavaScript need 3 main sets of functions. As preciously discussed, these are:

* Client side request handlers, these functions should handle making requests to the sever and receiving the servers response to return.
* Client side DOM interaction, these functions will affect the DOM (HTML displayed) to display outputs
* Client side board state class, this will contain all the data from the server about the current bord state and will manage the logic or receiving user inputs and making the relevant requests to the server vie the handlers

The Client side request handler will consist of the following functions:

ASYNC FUNCTION external\_get\_update()

    PASS

END FUNCTION

ASYNC FUNCTION external\_implement\_computer\_move\_and\_update()

    PASS

END FUNCTION

ASYNC FUNCTION external\_reset\_game()

    PASS

END FUNCTION

ASYNC FUNCTION external\_implement\_user\_move\_and\_update(from\_vector, to\_vector)()

    PASS

END FUNCTION

ASYNC FUNCTION external\_change\_difficulty(new\_difficulty)

    PASS

END FUNCTION

Each of these asynchronous functions will send the relevant data to the server and then wait for a response which will then be returned. The functions will be asynchronous as this allows other code to be executed while this function is idle (waiting for server response).

I will need the following functions to affect the DOM and display outputs to the user:

FUNCTION create\_board\_widget()

    PASS

END FUNCTION

FUNCTION get\_square\_at\_vector(v)

    PASS

END FUNCTION

FUNCTION add\_pieces(board)

    PASS

END FUNCTION

FUNCTION add\_highlighting(board)

    PASS

END FUNCTION

FUNCTION clear\_board()

    PASS

END FUNCTION

FUNCTION update\_main\_title(board)

    PASS

END FUNCTION

FUNCTION update\_pieces\_taken(board)

    PASS

END FUNCTION

FUNCTION set\_selected\_difficulty(board)

    PASS

END FUNCTION

FUNCTION set\_widget\_move\_history(board)

    PASS

END FUNCTION

FUNCTION update\_board\_widget(board)

    update\_main\_title(board)

    add\_pieces(board)

    add\_highlighting(board)

    set\_selected\_difficulty(board)

    update\_pieces\_taken(board)

    set\_widget\_move\_history(board)

END FUNCTION

create\_board\_widget()

let board = new Chess\_Board(

    update\_board\_widget,

    update\_main\_title

)

FUNCTION handle\_radio\_button\_click(radio\_button)

    PASS

END FUNCTION

FUNCTION handle\_square\_click(i, j)

    board.handle\_square\_click([i, j])

    update\_board\_widget(board)

END FUNCTION

FUNCTION reset\_button\_click()

    board.reset\_game()

END FUNCTION

FUNCTION concede\_button\_click()

    // update\_main\_title(null, "You Conceded The Game")

    board.concede\_game()

END FUNCTION

The chess class will have the following methods. It must be able to handle each of the 4 inputs. These include board square click, difficulty radio button click, reset button click and concede button click. It must make the appropriate decisions of how to handle these inputs. This may involve affecting the DOM using or making a request to the server using the above function.

Here is the UML diagram for the client side Chess\_Board class

Graphical user interface, text

Description automatically generated

This UML diagram shows only one class. This it’s the Chess\_Board class that will be used to represent data about the current board state client side. The properties and methods given have appropriate types included. Some of the types are piece types or vector types. The will in practice be implemented as a standardised format of array containing 2 values. A vector will an array of 2 integers and a piece will be an array of 2 strings. I have names these types in the UML diagram to make it clearer.

Describe database use (save games and minimax cache) and other persistent storage:

In my programs, most of the data stored will be contained within objects. I will make use of some key data structures in addition to these objects. These will be highlighted below.

In the client side JavaScript I will use a dictionary data structure which uses string keys and contains a list for each value. This list will contain a hexadecimal hash and a function.

Here is the pseudocode:

previous\_data\_hashes = {

    "pieces\_taken": [None, (board) => get\_hash\_of\_data(board.pieces\_taken)],

    "move\_history": [None, (board) => get\_hash\_of\_data(board.move\_history)],

    "piece\_layout": [None, (board) => get\_hash\_of\_data([board.pieces\_matrix, board.possible\_to\_vectors, board.selected\_from\_vector])],

    "highlighting": [None, (board) => get\_hash\_of\_data([board.possible\_to\_vectors, board.selected\_from\_vector])],

    "difficulty": [None, (board) => get\_hash\_of\_data(board.difficulty, stringify=False)]

};

FUNCTION update\_as\_necessary(board, update\_function, hashes\_table\_key)

    old\_hash, compute\_hash = previous\_data\_hashes[hashes\_table\_key];

    let new\_hash = compute\_hash(board);

    IF new\_hash != old\_hash THEN

        // hashes are different, updating table

        previous\_data\_hashes[hashes\_table\_key][0] = new\_hash

        update\_function(board)

    END IF

    // ELSE hashes are the same so don't update

END FUNCTION

FUNCTION update\_board\_widget(board)

    update\_main\_title(board);

    update\_as\_necessary(board, add\_pieces, "piece\_layout")

    update\_as\_necessary(board, add\_highlighting, "highlighting")

    update\_as\_necessary(board, set\_selected\_difficulty, "difficulty")

    update\_as\_necessary(board, update\_pieces\_taken, "pieces\_taken")

    update\_as\_necessary(board, set\_widget\_move\_history, "move\_history")

END FUNCTION

The function get\_hash\_of\_data represents a generic SHA2 hash function.

Each entry contains a hash of the data relevant to a certain widget on the board and a function to generate the updated hash.

The function update as necessary checks the hash of the current data against the old hash in order to decide if the relevant DOM update function must be run. If the hashes are the same, the function is not run which improves performance. If the hashes are different, the function is run and the hashes are updated.

I felt that this table is necessary as some of the DOM update methods can be visually noticeable to the user and so I don’t want to be recreating my HTML widgets every time if there is not change in output to display.

I used a dictionary data structure to organise the hashed and functions into the widget / data category that they corresponded to. I used a list as the 2 items that I wanted to store were of different types.

In my program I also used a 2 dimensional array of pieces. This was useful as it allowed me to very naturally represent that tabular layout to the chess board in an 8\*8 array. I will use a 2 dimensional array called pieces matrix in both the frontend and the backend.

The backend 2d array may contain values that are either None or a Piece object. The classes that will represent pieces were defined above.

Here is an example of how the pieces matrix would in the backend, using pieces objects (showing starting positions):

STARTING\_POSITIONS: array[array[Piece | None]] = [

    [

        Rook(color="B"),

        Knight(color="B"),

        Bishop(color="B"),

        Queen(color="B"),

        King(color="B"),

        Bishop(color="B"),

        Knight(color="B"),

        Rook(color="B")

    ],

    [Pawn(color="B"),]\*8,

    [None,]\*8,

    [None,]\*8,

    [None,]\*8,

    [None,]\*8,

    [Pawn(color="W"),]\*8,

    [

        Rook(color="W"),

        Knight(color="W"),

        Bishop(color="W"),

        Queen(color="W"),

        King(color="W"),

        Bishop(color="W"),

        Knight(color="W"),

        Rook(color="W")

    ]

]

The frontend pieces matrix will use an array of 2 characters to represent a piece. These character would correspond to the piece’s color and its symbol / type. This array pair of characters will likely be the easiest way to represent the chess board and will definitely be used in JavaScript and JSON data representations of the board. This would means that the pieces matrix will use a 3 dimensional 8\*8\*2 array of characters as appose to the 2 dimensional 8\*8 array of Piece objects

Here are some examples of describing pieces in this way:

['B', '♜']

['W', '♟︎']

And here is how I would then use this principle to describe a whole chess board (starting positions shown):

[[["B","♜"],["B","♞"],["B","♝"],["B","♛"],["B","♚"],["B","♝"],["B","♞"],["B","♜"]],[["B","♟︎"],["B","♟︎"],["B","♟︎"],["B","♟︎"],["B","♟︎"],["B","♟︎"],["B","♟︎"],["B","♟︎"]],[[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null]],[[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null]],[[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null]],[[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null]],[["W","♟︎"],["W","♟︎"],["W","♟︎"],["W","♟︎"],["W","♟︎"],["W","♟︎"],["W","♟︎"],["W","♟︎"]],[["W","♜"],["W","♞"],["W","♝"],["W","♛"],["W","♚"],["W","♝"],["W","♞"],["W","♜"]]]

I will also make use of a database in my program. This database will have 2 main uses:

* It will use entries in a table to store cached minimax function results for reuse
* It will use an entries in a different table to store saves games and a corresponding cookie ID contained withing the user’s browser

I will aim to make use of an external library like SQL-Alchemy of SQLite3 to implement my database. This will allow me to defend my program against accidentally running malicious SQL queries as a result of SQL injections attack. This is needed as the value held in a cookie in the users browser will be used to reference the saves chess game binary in the database. If this value were to be manually altered to be a harmful SQL query, I want to ensure that no deletion of data can be allowed.

Using a library should also make the process of saving and reading data to and from the database earlier. If I use SQL-Alchemy I can make use if an ORM (Object Relational Mapper) to directly describe how my objects can be converted to and entries in my database.

I will be able to create my 2 tables and the 2 indexes I want with the following SQL commands:

CREATE TABLE "Minimax\_Cache" (

        primary\_key INTEGER NOT NULL,

        board\_state\_hash VARCHAR,

        depth INTEGER,

        score INTEGER,

        move VARCHAR,

        PRIMARY KEY (primary\_key)

);

CREATE TABLE "Saved\_Games" (

        primary\_key INTEGER NOT NULL,

        cookie\_key VARCHAR,

        raw\_game\_data BINARY,

        PRIMARY KEY (primary\_key)

);

CREATE INDEX board\_state\_hash ON "Minimax\_Cache" (board\_state\_hash);

CREATE INDEX cookie\_key ON "Saved\_Games" (cookie\_key);

This should result in a database with the following structure being created:

Graphical user interface, application, Teams

Description automatically generated

The above entity relationship diagram shows that database structure that I aim to create. There is not relationship between the tables.

In addition, It could be argued that the use of integer primary keys is unnecessary as I am effectively using the board state hash and cookie key as primary keys. This is because I intend to use these values to search for rows in each table, this is why I have created indexes by these values. In many ways I think it would make sense to make these columns the primary keys, however many ORM libraries (including SQL-Alchemy) do not allow custom primary keys. This means that I must have an incrementing integer primary key, even if it isn’t used, due to the limitations of the framework with which I intend to implement my database.

Due to the exponential nature of the minimax algorithm. A single depth 2 search will cause around 900 depth 0 searches (static evaluation) to be completed, assuming a branching factor of 30. This will increase exponentially at a greater depth and could pose scalability problems for the database. In addition, depth 0 cache (stored depth 0 searches in the minimax cache table) is the least valuable of cached minimax results. This is because it can be calculated the quickest and so the cached item prevents the least computation. If the depth 0 cache was stored in a database in secondary storage, the slow read times (relative to time to compute) and large space requirement could result in no overall efficiency gain from the cache.

As a result I intend to implement the above database structure in both a local “.db” file in secondary storage and in separate logical database that will exist in RAM. This will allow me to store depth 0 cache in the volatile database. This will improve read write speed and prevent the persistent database form becoming overfull. This should still present efficiency gains as static evaluation are likely to be reduced within a game. It is not necessary to store these permanently as the chance of encountering the same chess board again is sufficiently low that only high value cache is worth storing.

With this explained redundancy aside, I believe that my database design is already normalised. This is because my 2 tables have no relationship and store no duplicated data. Therefore there is no redundancy and so this design is already in first normal form due to its simplicity.

The use of indexes will increase the amount of space needed to store each entry, as well as increasing write times. However, the reduction in retrieval times should make indexes on certain columns a worthwhile trade of as I interned to query the tables by these columns every time.

Going forward, if further tables needed to be added to the database (e.g. containing user data) then normalisation would need to take place again to ensure that the database remained as efficient as possible.

* Include UML diagram and ensure structure is normalised

Variables and key data structures used in these key algorithms:

Do to the heavy use of OOP in my prosed design, global variable will be rare. I outlined the properties of each of my classes using UML diagrams earlier. I have also outlined key data structures that were used earlier.

In addition to this, I intend to use a variable to keep track of the client server session on the server side. This should be able to contain the serialised binary for the specific game object being used in each of the browsers games, while the user has the website open. There is already a way of achieving this within the Flask framework using a flask session object. This uses temporary cookies on the client’s browser to store some data (the Game object) while the tab is open.

I intend to read the cookies of the client browser, when the session is first created. If a cookie has already been created by this website then I will query the database for the relevant game object and restore it into the session.

When the session closes I will save the game object to the database and create a cookie on the clients browser containing the cookie key of that saved game’s database entry. This will allow the game to be reloaded if the user visits the website.

Create a table of key data structures and variables and include there data type as well any key validation

Complete trace tables and dry runs of key algorithms:

Try and trace through these key algorithms:

* Some vector calculations
* Piece movement: Trace through the algorithm for a specific piece on a specific board states
* Then trace though check function assuming that piece movement functions are all working
* Then trace though generate legal moves, assuming check function and piece movement working
* Then trace through game over
* Try to trace though with alpha beta pruning, a basic chess board that it towards the end to show the optimal move

I will now aim to trace though some of my main algorithms. As this system uses layers of abstraction, high level functions such as the minimax function would be very difficulty for a human to trace in its entirety. This is because a single minimax search needs to run many board state functions, even move pieces functions and a huge volume of vector calculations. As such I will aim to simplify where appropriate.

In the case of the minimax function, I will use a general decision tree and abstract away the process of determining static evaluations. I will focus on traversing the tree and then picking the appropriate score.

I will use a diagram of a tree that I have created to trace though the minimax algorithm. I have traced the algorithm though by annotating the diagram at various points.

Firstly I traced though a standard minimax function without any pruning.

A diagram of a system

Description automatically generated with low confidence

Here we can see a simplified illustration of a depth 3 search by the minimiser. The tree has been simplified by only showing 2 of the children of each node. The static evaluations of the terminal nodes are written at the bottom. Circular nodes represent a decision by the minimiser whereas rectangles are for the maximiser.

A picture containing necklet, accessory

Description automatically generated

Here we can see that -6 is the lowest score among 5 and -6 and so the minimiser choose -6. As a result, the current best score of the parent maximiser move is also now -6.

A picture containing necklet, accessory

Description automatically generated

We can see that this process is repeated with the minimiser picking -5. As there is complete information and the maximiser anticipates that the 2 child nodes, the minimiser can get a score of -6 and -5 respectively, therefore to maximise its score, the maximiser picks the second child with a current best score of -5.

A picture containing necklet, accessory

Description automatically generated

This process repeats again.

A picture containing text, map, necklet, linedrawing

Description automatically generatedthen finally, all parts of the tree have been searched and the minimiser decides that the best score that they can guarantee is -5 (at a depth of 3).

Now I will trace though the minimax function more slowly to show its method when using alpha beta pruning.

Diagram

Description automatically generated

Here we see again a recursive search of depth 3 by the minimiser. I have only filled in the values of nodes once they are reached by the algorithm. Above we see the recursive call has passed its starting values of

alpha = -infinity

beta = infinity

down in all its recursive calls until the base case is reaches and a static evaluation is performed to get a value of -6.

Diagram

Description automatically generated

We then see how after examining the first child, the indicated minimiser node (minimisers are circles and maximisers are rectangles) has updated it beta value and its current best score to -6. It then continues to search the children and the next static evaluation is 14.

Diagram

Description automatically generated

We see how, because the previous value of -6 is better than 14 for the maximiser, the values of beta and best score remain unchanged in the minimiser node (indicated with a tick). It parent maximiser node has not successfully searched its first child and so updates its alpha and best score values to -6.

Diagram

Description automatically generated

We then see this process repeat recursively. Each time, the alpha and beta values form the parent node are passed down to the search at a lesser depth. (they are never returned up the tree)

Diagram

Description automatically generated

The minimiser node is updated with a new beta and best score value of -11. It should be noted at this point that the pruning condition: alpha >= beta has been met. This is not relevant in this example, but if the minimiser node had more than 2 children, it could stop examining them now.

Diagram

Description automatically generated

We then see that the minimiser 2nd minimiser node on the third level has finished its search. This means that the 1st maximiser node on the 2nd level has also finished its search with no change to its variables. This causes the variables of the root node to update as its first child has been searched.

Diagram

Description automatically generated

We then see the right side of the tree get recursively searched with alpha and beta values passed down by the root.

Diagram

Description automatically generated

Diagram

Description automatically generated

The 3rd minimiser node on the 3rd layer has now been completed. The alpha and best score values of the 2nd maximising node of the 2nd layer are now updated.

Again we see that alpha >= beta. This means that the 2 static evaluations on the right were prevented as they were not necessary. This can be logically explained as the minimiser at the root node has to decide which of the 2 maximising nodes on the 2nd layer are the worst for the maximiser. Without exploring the last part of the tree, we can see that the alpha value for the right maximiser node is -5 which is already higher that the -6 alpha value of the left node. This means that no matter what the scores are in the unexplored part of the tree, the right half of the tree is already better for the maximiser and so the minimiser at the root node will always pick the left maximising child node.

Diagram

Description automatically generated

This means that the depth 3 search is completed with a final score of -6. If no pruning was used, the final score would still remain the same.

I have now accurately traced the minimax algorithm through in a graphical manner. This traceback exemplified the order in which it searches nodes. It also shows how the values of alpha and beta are updated and passed to child nodes. With the previously defined pseudocode, I can apply this searching method to chess. The use of a best move variable in the pseudocode will also ensure that the move is returned and not just the score.

Validation:

The final program would feature both client side and server side validation. This should allow invalid inputs to be detected so that they do not cause invalid outputs.

My final program will have 4 main inputs:

* The user can click any square on the chess board
* The user can click the reset button
* The user can click the concede button
* The user can click the radio buttons that control difficulty

Client side validation needed:

It should not be possible for the user to provide an invalid input to the concede and reset buttons. The button should be able to be used as any time. This includes when the game is over but also during both the user and computer’s go. The only input the user should be able to provide is a Boolean, did they click the button? yes or no? Both of these inputs (pressing and not pressing the button at any given stage in the game) are valid.

It should also be impossible for the user to select an invalid difficulty. The difficulty settings will be changed using radio buttons. This means that when the user select one option, all previous options should be unselected. Using radio buttons (e.g. as opposed to a text input) will also reduce the range of inputs that can be given as only valid difficulties can be selected. I will make the program automatically select one of the radio buttons (e.g. default of medium) when the page loads. This should ensure that at all points in the program, exactly one of the possible difficulties will be selected.

I will also use client side validation to decide when the client should ignore erroneous and invalid click events from the user. To input a move the user will click 2 squares on the chessboard: a move from square and a move to square. I will use highlighting to make this intuitive.

The client side Chess\_Board class should contain an array of legal moves that the user can make at any one point. After the user move these should be updated by the server. The Chess\_Board class will use an additional 2 properties to allow the user to input their move and add validation in the process:

* Selected\_from\_vector: This variable will contain none of a position vector in the form [i, j] where i and j are integers
* possible\_to\_vectors: This variable will be an array that will contain position vectors (in the form [i, j] where i and j are integers)

Before the user first clicks a square, the selected\_form\_vector will be set to None and the possible\_to\_vectors will contain an empty array.

When the user clicks a square, the position vector of that square will be checked to see if it is an item in the possible\_to\_vecotors variable. It the clicked square’s position vector is a in the possible\_to\_vecotors property then the user move will be made. If not then the user has not selected a valid to square.

I will then check if the clicked square contains a white piece. If It does then it will be treaded as the user’s input for their move from square (which piece they want to move). The position vector for the clicked square will be stored in the property selected\_from\_vector. The possible\_to\_vecotors will then be updated by iterating through the array of legal moves and adding all possible legal moves that start by moving the selected piece.

This method will allow me to:

* Ensure that the user cannot move the opponent’s pieces. This is because only clicking one of the users own pieces will allow them to input a from square. Clicking other squares will cause the selected vector property to be set to none.
* Ensures that the once the user has selected one of their pieces to move, the cannot the select an invalid movement square move to. Only squares in the position vector array possible\_to\_vecotors (a subset of legal moves for where the selected piece can move) can be selected as a to square

All invalid square click events will be ignored.

Server side validation:

I will also add server side validation to prevent invalid inputs.

If the user tries to bypass the JavaScript and make a move that they should be able to an error will be caused on the server. Due to the use of the Flask framework, this error will be caught. While it will be logged to the console and it may prevent that client from continuing with their chess game, it will not cause the server to crash or disrupt other users.

My server side validation should only be used when users deliberately bypass of tamper with the client JavaScript. As a result I will focus on preventing these invalid inputs from causing unexpected behaviour. I will not produce an error message that will be sent to the client to explain the issue as it will have been caused deliberately.

Below is a list of the various exceptions that should be raised on the server to top it and the relevant invalid input they represent. I will validate the input and then deliberately raise and exception to prevent the server using the invalid input which could result in unexpected behaviour. I will create the custom exceptions below myself by creating a python class that inherits form Exception.

|  |  |
| --- | --- |
| Invalid input to server | Resulting Exception |
| Invalid difficulty provided to the change difficulty handler | ValueError(“difficulty not recognised”) |
| Not the user’s turn, and yet the client has tried to implement a user move | NotUserTurn() |
| Not the computer’s turn and yet the client has tried to implement a computer move. | NotComputerTurn() |
| Illegal move: it is the users turn but the provided move isn’t legal | InvalidMove() |
|  |  |

Functionality that each prototype will have:

Prototype 1:

The aim for this prototype is to successfully tackle a simpler problem with a similar approach. I will aim to create a website that will allow uses to play tic tac toe against the computer.

The users will have access to a 3 by 3 board with which to play the game. A title will declare that it is their turn to move first. The board’s squares will respond to their mouse clicks, allowing them to put a cross in any of the 9 squares. There will also be a reset button to allow for the game to be restarted.

I will create an API connection between the server and the client using either WebSocket HTTP. This will allow the client to request the user’s move. The server will then respond with a move which the client will implement on the board. The game will continue until the computer wins or the game is a draw.

The computer will use alpha beta pruning and the minimax algorithm to use a decide the computer move. The approach will be to use the British Museum algorithm to look ahead and search the whole decision tree, through to the 9th turn where all games end. This will allow the computer to perform a search and decide the best move to use.

This prototype will help me work towards chess and will help me partially achieve a variety of success criteria. These include:

* Working towards items 2, 3, and 4 by adding client side validation to ignore clicks when it is not the user’s turn
* Working towards items 6 and 7 as I will need to graphically show the computer and user move
* Working towards items 9, 10 and 26 as I will create a first iteration of a working minimax algorithm that can then be later adapted to play chess.
* Working towards item 17 as I will need to create a similar client server API connection for this simpler program
* Working towards item 31 as I will need to create a reset button in this game

Prototype 2:

This prototype should begin to address the complexities of chess. It will include a working version of the 3 main components of the chess computer:

* It will include a fully finished Chess Engine / Chess Functions module that can make inferences about a chess board. This must include detecting if the game is over and identifying all legal moves
* I will have created a working prototype of the Move Engine Component. This version of the minimax function will include heuristics and search to a given depth instead of using the approach of the British Museum Algorithm. The move engine should also feature some more optimisation including variable depth checking and pre-sorting moves
* I will also create a working prototype of the Game Manager module that is able to keep track of and manage an individual chess game

I will also aim to create a console based chess game what uses my game manager module. It should provide a text based interface with which the user can play chess against the computer

I will also aim to begin exploring how best to implement final chess webpage GUI. I will attempt to use a framework called VUE to achieve this. I will then build upon my progress with the user interface in prototype 3.

Prototype 3:

This will be the final prototype of my iterative development cycle. It will aim to create a program that meets all the design criteria. Doing this will enable me to create a solution that meets all essential, high priority and desirable success criteria items. I can then add more featured from the success criteria if I have move time later.

This means that the third prototype should feature a webpage GUI that the user can access to play a game of chess.

The Move Engine should be fully optimised to maximise efficiency and efficacy.

The Game Manager should be fully implemented and include multiple different difficulty setting.

­

Test Plans

Functional white box testing during development:

Test plan: iteration 1:

For this iteration, I will test a variety of aspects of the final webpage and game to ensure that the chess AI, validation and GUI behave as anticipated.

To do this I will perform the following tests:

* Valid: Try clicking a square when the game loads and see that it now contains an “X” and I should see that another square fills with a “O”.
* Invalid: I will then test what when I click the 2 filled squares, nothing happens.
* Valid: I will test that when I click another square, I am able to make my second move and put an “X” there
* Valid: The game should be able to successful identify when the user has lost. It should freeze the board and display this
* Valid: The game should be able to successfully identify when the game is a draw and produce the appropriate output.
* Valid Extreme: The game should be able to successfully identify when the user wins and handle this. The chess AI should prevent this ever happening so I will perform the test by loading a board state into the board so that the user is a move away form winning.
* Valid: The reset button should work in resetting the board in all 3 game states: the user’s move, the computer’s move and when the game is over.

Test Plan for Prototype 2:

The testing for this prototype will come in 2 main parts:

Firstly, I will create automated test for the Chess Engine and Move Engine to ensure that these components are working correctly.

To test the Chess Engine I will test the various functions that I have created for a variety of inputs and expected outputs. I will store the test data in external files and use the python unit test library to complete the tests.

It is important that the functions in the Chess Engine are robust as they will be used by many of the other modules. For instance the Move Engine Module and Client Side validation rely on the generate legal moves and is game over functions. If these have logic errors, this could produce knock on logic errors in the wider program.

I will create automated tests to test the following functions for a variety of inputs and expected outputs:

I will test the method of the Vector class:

* Vectors are equal
* Add vectors
* Subtract Vectors
* Multiply Vectors
* Check position vector in chess board
* Check the vector can be converted to a square (e.g. F5 or A2)

I will then test the generate movement vectors method of the piece class for each pieces.

To do this I will check that the piece can move where I expect on an empty chess board.

I will the create a populated chess board that is filled with other pieces. This will allow me to test the vectors by which a piece can move are affected by the pieces around it, resulting in the expected legal moves (ignoring check).

I will then test various methods of the Board State class:

* I will test the generate all piece method for various piece layouts
* I will test the get piece at vector method for various pieces at different places in various boards
* I will then test that the generate pieces of a specific color method works. I will do this by checking that it can give me all the black and white pieces for a variety of chess board states.
* I will then test that the is color in check method works for a variety of board states and edge cases (including checkmate).
* I will test that the game over function is able to correctly recognise then the game is over in a variety of board states as well as say the nature: “checkmate / stalemate” an winner “none / w / b”
* I will test that the generate legal moves function is able to correctly identify all the legal moves available to a given player for a variety of board states. This will include testing extreme data such as game over situation where there aren’t any legal moves.

I will then test the Move Engine’s Minimax function.

To do this I will create a variety of tests where 2 different computer bots play against each other. All my test will have a “good bot” that I expect to win and a “bad bot” that should in loose. I will use my Game class to have the 2 bots play against each other. I will output the moves that they made and the running score to a file. By ensuring that the good bot beats the bad bot and by checking the graph of the game to see that the win was progressive (gradually took pieces then won, not was loosing and the bad bot then made a mistake) I can test that the good bot is better.

Being able to create a test like this where I can confirm that one bot is better than another will enable me to:

* Confirm that my minimax algorithm is making intelligent moves by having it beat a randomly moving bot
* Confirm that a minimax search at a greater depth gives me a better move by testing that bots with a greater depth beet bots with a lesser depth.
* Confirm that variable depth checking produce better moves. To do this I will test 2 depth n bots against each other to see that the good bot with a variable depth of 1 beats the bad bot.

I will create the following test to test my minmax algorithm: (good bot on the left, bad bot on the right)

* Depth 1 vs random mover
* Depth 2 vs random mover
* Depth 3 vs random mover
* Depth 2 vs depth 1
* Depth 3 vs depth 1
* Depth 3 vs depth 2
* Depth 1 + 1 variable depth vs depth 1
* Depth 2 + 1 variable depth vs depth 2

I will then also complete a small amount of basic tests on the console chess game to ensure that the Game Manager is doing its job:

* Test that the computer waits for the users go
* Test that the user cannot go during the computer’s turn.

Post development testing against development criteria (black box alpha testing):

Acceptance testing for success criteria (beta testing):