**Design Part 2:**

Decomposing the problem:

In order to achieve the bulk of the points in my success criteria, I must create a system that includes a chess computer and a user interface. I will decompose the whole solution into these 2 parts and a form of API to connect the 2.

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

Description automatically generated

The Benefits of Using This Decomposition:

This decomposition identifies 3 separate large components of the solution with different responsibilities.

* The Chess Computer should be able to analyse different chess board states in order to determine legal moves, detect check etc. It should also be able to determine a good legal move for any board state in order to allow the user to play the computer at chess. Additionally a Game component will be needed to keep track of a chess game: determining whose go it is, taking inputs and providing outputs such as when the game is over
* The GUI Webpage is responsible for providing an intuitive API connection to the objects that describe the chess computers logic, server side. It also allow for inputs to be validated to ensure that the final program is robust.
* The API connection is responsible for allowing the GUI webpage to send inputs and requests to the chess computer and get outputs to display to the user in response.

By considering each of these components separately, we can ensure that each component has a core subset of the whole systems responsibilities. This will allow for each component to be more easily planned out and developed.

Additionally, identifying sperate components and the separate skills needed to create them allow for each component to be developed and tested in parallel before being combined. In a professional project, a this decomposition could allow different members of a team with different skill sets to work in different parts of the program in which they have more expertise.

Chess Engine:

In order to create a chess engine I will need 3 main systems:

* An algorithm / collection of objects to keep track a single chess board state
* A variant of the minimax algorithm that should be able to determine the best move for a given board state.
* A game object or program that can keep track of turns and taking in user inputs.

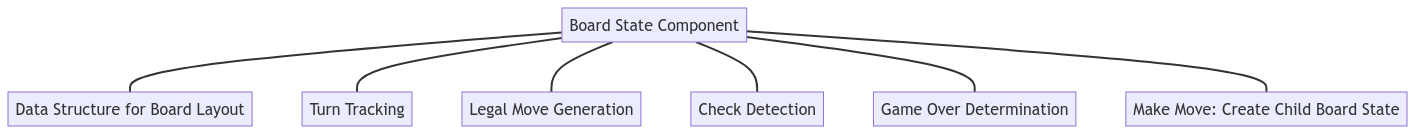
Graphical user interface, application

Description automatically generated

Board State Component:

This component should be responsible for analysing and understanding the current board state. This should include:

* Keeping the layout of pieces in the board in a data structure.
* Keeping track of whose turn in next.
* Determining the legal moves available to the player who is next to go (accounting for rules around check).
* Determining if the current board state is in check
* Determining if the game is over from the current board state and the nature of how the game is over (e.g. who won or was it a stalemate)
* Allowing a move to be made to produce a new board state object



Move Engine Component (Minimax Algorithm):

This component should allow me to perform a minimax tree search from the current board state. It should features some optimisation that distinguish it from the vanilla minimax algorithm. This algorithm may take the form of a function or an object depending on its final complexity. I will use unit tests to ensure that faster variants of the algorithm produce the same output as the original algorithm. The algorithm should assume that the opponent will play optimally and maximise its utility with this consideration.

While no tree data structure will exist in memory at any given time, it will construct a logical decision tree that the recursive algorithm will navigate. In this tree, board states will be nodes and moves will be arcs. This algorithm is dependant on the board state management component which must be able to generate legal moves, implement and implement them to create many new child board states. The decision tree will be explored to a given depth (so many moves ahead). Once the leaf / terminal nodes are reached, a utility score to evaluate the board state will be generated. This score will be based on the game’s outcome if over (who has won or if the game is a draw). If the game is not over at the terminal node it will use a heuristic that will account for pieces lost and piece distribution to estimate how favourable that board is to a given player.

The algorithm should then backtrack from the terminal nodes and back propagate back to the root node to determine the best move, assuming the opponent also plays optimally.

Game Program:

Where as the board state component represented a single board state (a snapshot of a game), this component should allow for a whole chess game to be described and contained in a single object. `

API Connection:

This component will be made up of server side code and client side JavaScript. It must facilitate the exchange of information about the chess game between the client side user interface and the server side chess computer. This should allow for the client side interface to be relatively dumb, allowing for the bulk of the logic to take place server-side. This means that I will not create duplicates of functions like generate legal moves in JavaScript, instead a fast API connection will allow for the client to request this information from the server.

I will user WebSocket for the connection as it has a few desirable properties. Specifically it allows for a direct TCIP connection between the client and the server meaning that latency will be minimal. Additionally the connection between the client and the server is full duplex. This means that it acts like a phone connection in that it is 2 way, meaning both the client or server can send data without prompt.

I will for the most part make use of a general structure for the WebSocket connections. Most events (e.g. “make\_user\_move”) will be implemented as 2 separate events for the request and response respectively (e.g. “make\_user\_move\_request” and “make\_user\_move\_responce”). Data will be exchanged in a JSON format. A general response will be created to update the client on the specific attributes that make up the chess game.

I will also make use of sessions and cookies to make the exchange of data easier.

I will use sessions to store a python object that describes the current state of the chess game for each client of the webserver. This will allow me to send less information. For instance the client JavaScript will only need to send a pair of vectors when informing the server about a user’s move. Data such as the layout of the pieces for that specific game is already held in the session’s game object.

Cookies will be used to store a small amount of data in the clients browser. This will be in the form of a database ID for a saved chess game. When a session is created (triggered by the client visiting the website) the browser’s cookies will be checked. If there is a cookie containing an ID corresponding to a saved chess game on the server, then that game will be restored into the session and sent to the client. When the session is about to be closed (when the user closes the tab), the game object contained withing the session will be serialised to binary and stored in a database. A cookie containing the primary key of this database entry will be stored in the browser to allow the game to be stored. This will allow the user to resume a game where they left off. If this is not what they wanted, they should be easily able to reset the game with a corresponding button.

Here are the main API routes that I expect to use to convey data between the client and the server.

Diagram

Description automatically generated