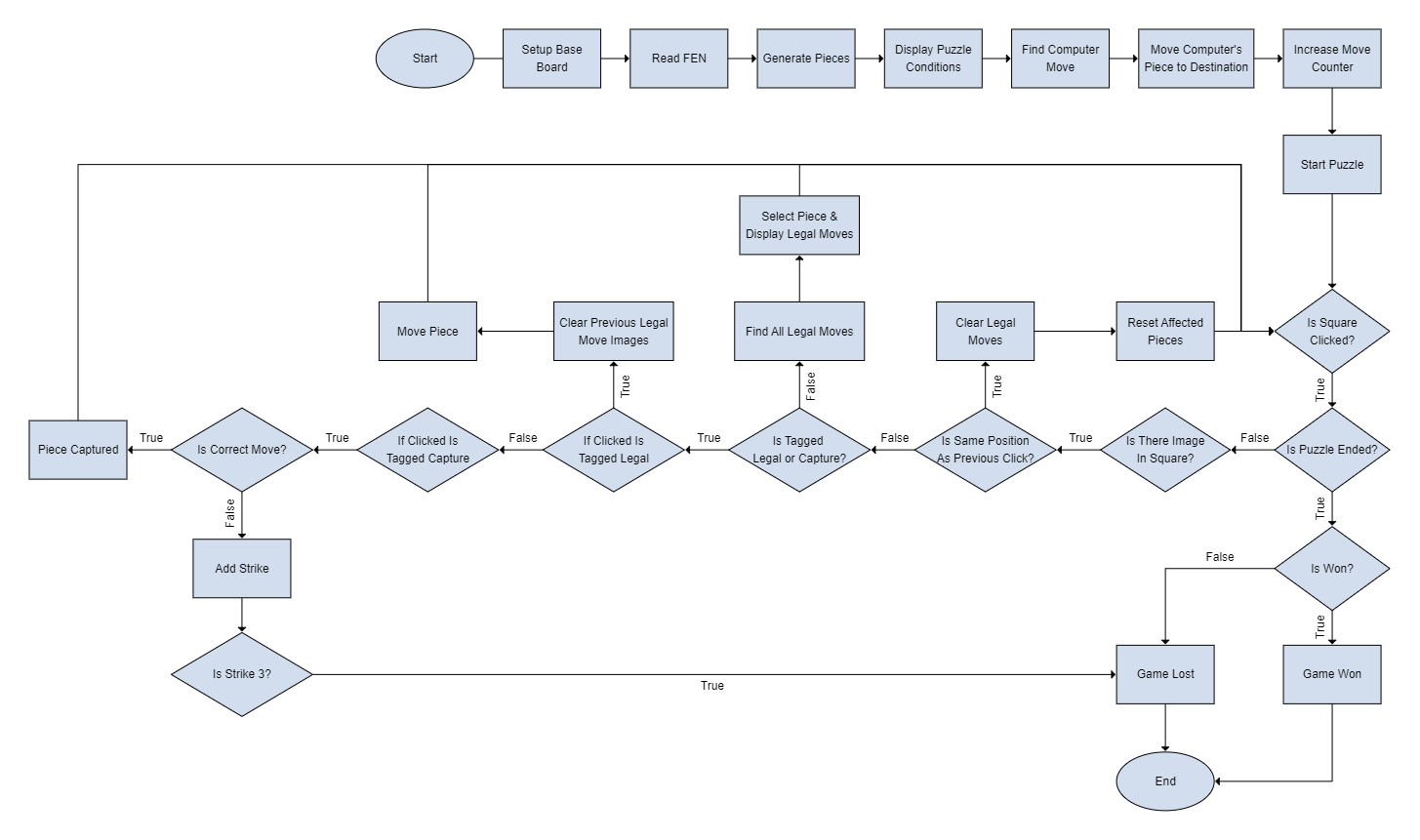
**Introduction**

**Writer’s Statement**

**Initial Planning**

Before any programming could be completed to create the ‘Daily Chess Puzzle’ solution, planning first had to be completed. This is through an initial flowchart and various UI design sketches. The flowchart illustrates that upon the startup of the application, the base board is setup. Then the position FEN of the puzzle is read, with pieces generated accordingly. After the board has been setup, the conditions surrounding the puzzle is displayed, and the computer makes their move. It is only after those steps that the puzzle will start for the user. Now that the puzzle has started, if a square on the chessboard is clicked, the playing status of the puzzle is checked, and if the puzzle has not ended, the square will be checked to see if it contains any images. This is because images are used to display pieces and the available legal moves for the selected piece. If the square clicked is the same as the most recent click, then the legal moves will be clicked, and the pieces affected will be reset. If the square clicked is not the same as the previous click and does not contain a tag saying “legal” or “capture”, then all legal moves for the selected piece is found and displayed. If the square clicked does contain the tag of “legal”, then the previous legal moves are cleared from the board, and the selected piece is moved to its new destination. If the square clicked contains the tag of “capture”, then the move is checked if it is the correct move according to the puzzle, and if it is the correct move, then the piece in question is captured. If the move is incorrect, then a strike is added. If the strike count reaches 3, then the game is lost. If all moves of the puzzle is completed before all 3 strikes have been accumulated, then the game is won.



The UI designs that were created for the application showcased three main designs. The first contained the window being vertical, with the puzzle details in the upper third of the window, and the board covering the lower two-thirds. The second design consisted of the window being horizonal and maximised, with the board the left two-thirds, and the puzzle details and moves made on the right third. The third design was similar to the second design, however it did not contain the moves made, and however showed the correct and incorrect moves made, as well as the House teams scoreboard. After consulting my client, it was determined that the third design would be used as it provided the most engaging and detailed display of all three designs, which fitted their purpose of the application.

**Iteration 1**

In the first build of the application, the most part of the application was created, with the puzzle being updated each day manually by changing the constant values of “fen” and “moves” to the correct values in accordance with each day.

Reading FEN

When reading the FEN, there are six sections within it that must be understood. These six sections consist of the piece placement, the side to move, castling ability, en passant target squares, a half move clock, and a full move counter.

<FEN> ::= <Piece Placement>

' ' <Side to move>

' ' <Castling ability>

' ' <En passant target square>

' ' <Halfmove clock>

' ' <Fullmove counter>

When it comes to piece placement, each rank is represented, being separated by a forward slash. Within each rank, there numbers used to state the amount of blank spaces, with uppercase letters referring to white pieces and lowercase letters referring to black pieces.

<Piece Placement> ::= <rank8>'/'<rank7>'/'<rank6>'/'<rank5>'/'<rank4>'/'<rank3>'/'<rank2>'/'<rank1>

<ranki>  ::= [<digit17>]<piece> {[<digit17>]<piece>} [<digit17>] | '8'

<piece>  ::= <white Piece> | <black Piece>

<digit17>  ::= '1' | '2' | '3' | '4' | '5' | '6' | '7'

<white Piece> ::= 'P' | 'N' | 'B' | 'R' | 'Q' | 'K'

<black Piece> ::= 'p' | 'n' | 'b' | 'r' | 'q' | 'k'

When it comes to the side to move, this is presented with either the letter “w” or the letter “b”.

<Side to move> ::= {'w' | 'b'}

When it comes to Castling ability, if white can castle to Kingside, then an uppercase “K” will be present. If white can castle to Queenside, then an uppercase “Q” will be present. If black can castle to Kingside, then a lowercase “k” will be present. If black can castle Queenside, then a lowercase “q” will be present. If castling is unavailable for both sides, then a hyphen will be present.

<Castling ability> ::= '-' | ['K'] ['Q'] ['k'] ['q'] (1..4)

When it comes to En Passant Ability, if en passant can be completed, the target square of the pawn will be displayed. Otherwise, a hyphen will be present.

<En passant target square> ::= '-' | <epsquare>

<epsquare>  ::= <fileLetter> <eprank>

<fileLetter> ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h'

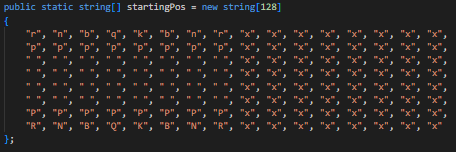
<eprank>  ::= '3' | '6'

**Board Representation (0x88)**

In the background of this application, the representation of the board is important as it is the array of the board behind the visible board, which is utilised in move generation and move checking to determine if a move is legal or not. When programming the board representation, there are two main ways that it can be addressed. This can be through implementing Bitboards, which is a set of 64 elements, which are all squares of the board, with one bit associated with each square. However, I did not find this as useful to me as using the square centric method of 0x88 to illustrate the board. 0x88 represents the board in 128-byte array, with only half of the array representing valid squares on the board. Within the initial board representation, it can be seen that half of the array elements consist of a blank entry, with the other half containing x’s. Another example of this in implementation is in the starting position array, which illustrates where each piece starts on the board in a standard game. It can be seen that as usual, the lower and upper two ranks contain the pieces are regularly seen, however being surrounded by both open strings and x’s.

A screen shot of a computer screen

Description automatically generated



**Move Generation**

Within this project, one of the most major parts of the program is move generation. This is because to make any move on the chess board, the legal moves must be checked. As this Daily Chess Puzzle application has been designed to only consist of “White to Move and Mate in x number of moves”, the pseudo-legal move generation method is used. This where the moves made by the piece isn’t checked if it leaves the king in check, as this would mean that more resources would be used to check for more possible lines, when it isn’t required to be present as the move will be returned as incorrect anyway. In the line displayed image, which reads “if Main dot board, previous position minus 16 is equal to blank string and Board dot is on board, previous position minus 16”, the isOnBoard method is called to return a Boolean. This is to utilise the 0x88 board representation, through checking if the piece is on the board through the line, “if destination and 0x88, then square is invalid”.



A computer screen shot of a program

Description automatically generated

A screen shot of a computer program

Description automatically generated

**Iteration 2**

After displaying the first build of the application to my client, they raised the question about whether the inputting of the puzzle could be automated, rather than being manually updated each day. His concerns were that if the puzzle had to be manually updated each day, then what will happen if the app were to be distributed to students to use on their school-provided computers.

In the second build of the application, the method of changing puzzles each day was altered from being done manually, to being read through a CSV, also allowing for separate difficulties to be selected for additional points. After addressing these plans with my client, it was determined that the Easy difficulty would give 3 points for perfect completion, 2 points if 1 strike is accumulated, and 1 point if 2 strikes are accumulated; the Intermediate difficulty would give 6 points, 5 points, and 4 points; and the Hard difficulty would give 9 points, 8 points and 7 points. Through using Lichess’s open puzzle database, a CSV file containing more than 3 million puzzles with ratings spread from 400 ELO to 2900 ELO. Taking these rating factors, the different difficulties were separated into Easy, being all puzzles below the ELO of 1400, Intermediate, being all puzzles between 1400 and 2000 ELO, and Hard, being all puzzles above the ELO of 2000.

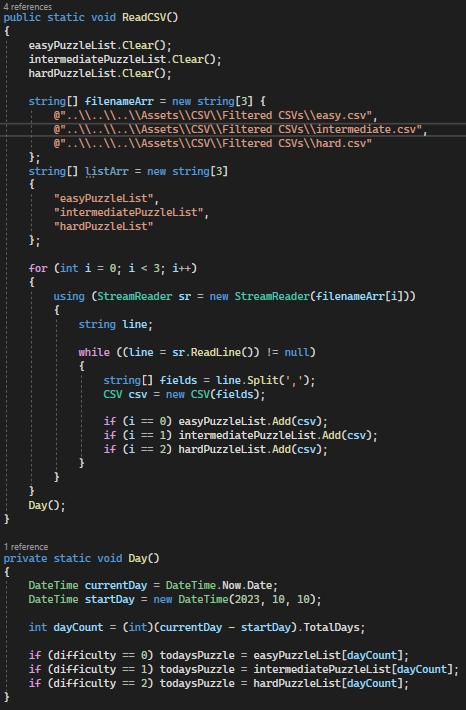
With the different specifications surrounding what sort of puzzle would be used, the master CSV provided by Lichess was filtered in to three CSVs each representing their difficulty. The specifications that were determined to be incorporated was only “White to Move” puzzles, and puzzles that result in “Checkmate”.

**Reading CSV**

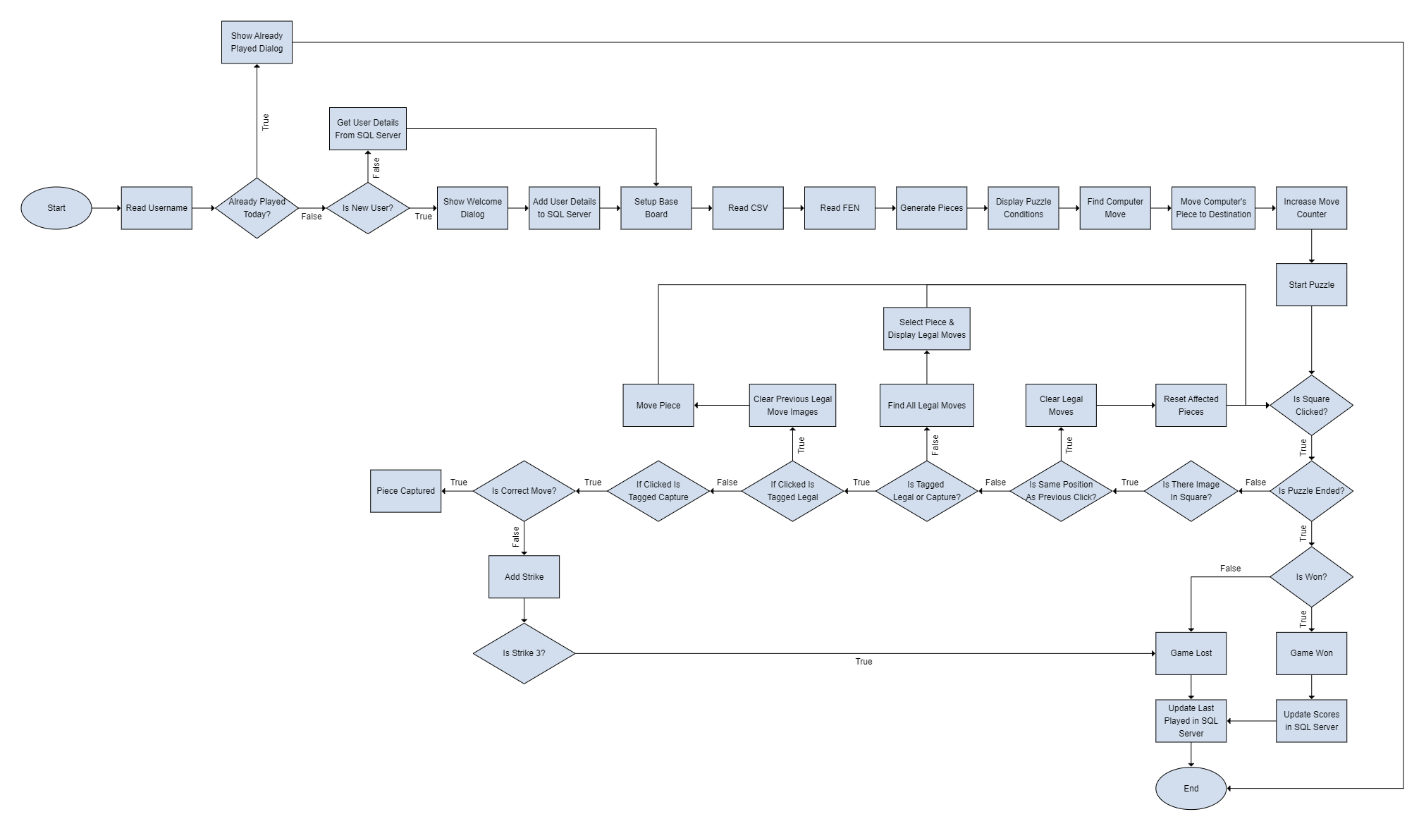
When reading the three CSVs to the application, they were added to lists to be read from each day. Within the Day() function, the start date of the puzzles, being set to the 13th of October, is subtracted from the current day to get the integer value of what day of the puzzle list that the daily puzzle is up to. From this point, the list associated with the set difficulty, has it’s puzzle allocated to the todaysPuzzle variable.

A screen shot of a computer program

Description automatically generated



**Iteration 3**

****