**Introduction**

The task at hand is to create a C# application, which will help with fixing the separation in House Team culture by providing the students an opportunity to compete against each other in a daily chess puzzle game, in which each student has a chance to help play their part in their House Team’s success throughout the year. With the main demographic of use being students, the client, who is a leader within the school, who has a rough knowledge of chess, was consulted in accordance with their plans, and the changes they saw fit for the application.

**Planning**

Before any programming could commence 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 cleared, 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 are 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 taking up 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 requirements of the application.

**Abstraction**

If this is the first time that the user has opened the application, then the user will be presented with a Welcome Form, which will allow the user to select the house team they are in, as well as the difficulty they wish to compete in. This information will be passed into the Create New User function. If the user has played in the past, the new user and welcome will not be shown, and the player’s data will be loaded. This involves their username, house team, difficulty setting, score and when they last played being passed on into the main form to load the correct difficulty puzzle onto the screen. After this has occurred, the Puzzle is loaded onto the Main form, through the CSV data of the FEN, moves, and rating, which is passed into the Puzzle class. From the puzzle class, if the puzzle is over, the isFinished, isWon, and isGameOver, are passed into the Game won or lost functions. If the puzzle is not over, the puzzle class passes the easy puzzle list, intermediate puzzle list, and hard puzzle list to the main form, alongside the isMove and isFinished Booleans. From this, the main form passes the section variable and rankNum variable to the setup board function, which will correctly display the puzzle according to the FEN set in the puzzle class. From there, the main form passes the isCapture, isLegal, and isMoved Booleans alongside the previous piece, previous position, current position, current square name and previous square name variables to the move function.

**Iterative Development**

**Iteration 1**

In the first build of the application, the majority 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. 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 number of blank spaces, with uppercase letters referring to white pieces and lowercase letters referring to black pieces. When it comes to the side to move, this is presented with either the letter “w” or the letter “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. 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. After the En Passant Ability section of the FEN, follows the halfmove clock, which represents the amount of half moves that have been made since the last capture or pawn move and the fullmove counter which is the number of moves made in the game up until that position.

Within my application, I utilised the knowledge about FENs to create my ReadFEN function, which checks if it’s a new puzzle, and if so, the board will be cleared to ensure that there is no way of overlapping board positions. Once the board is cleared, the FEN is split at any forward slash or space into 13 sections, of which, the first eight are the ranks. Using a switch case, the ranks are split into a character list, then passed into the Piece Generate function along with the rank number. This is effective as it allows for the modularisation of the code, through using generic functions that are fed parameters to complete the work that can be completed in a repetitive form if not done through an external function.

**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 represent 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, half of the array elements consist of a blank entry, with the other half containing x’s. Another example of this being used 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 that contain the pieces are regularly seen, however being surrounded by both open strings and x’s.

**Move Generation**

Within this project, one of the most integral parts of the program is move generation. This is because to make any move on the chess board, the legal moves must first 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 in the 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”.

**Iteration 2**

After displaying the first build of the application to my client, they raised the question about whether the updating 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 was acquired. 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 into three CSVs each representing their difficulty. The specifications that were determined to be incorporated were only “White to Move” puzzles, and puzzles that result in “Checkmate”.

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 its puzzle allocated to the todaysPuzzle variable.

**Iteration 3**

After adding the automation of the Daily Puzzle, the client particularly liked how the puzzles were selected on the daily basis, with the ability to play in different difficulties. However, the client did question whether it was possible for the user to complete the Daily Puzzle twice in one day, and if it was possible to have a point system.

To address this request from the client, an SQL database was created that would handle the individual user’s information, being their username, house team, current overall score, the date they last played on, and the difficulty setting that they have set. On top of this information, the Team Scores would also be kept within their own table, being added through the query “update team scores set score equals team score where team name equals house team”. Through these queries, the code can check if the user has already played on any given day, and if so, they will not be allowed to play again until the next day. This is effective as it will prevent the user from repeating the same puzzle multiple times, preventing them from gaining more points than they are entitled to achieve.

**User Interaction and Experience (2 minutes)**

* Walkthrough of the user interface.
* Discuss effective data validation and error prevention.
* Discuss user input and system output, and how these were considered in the design.

**Designer Statement**

When creating the Daily Chess Puzzle application, there were many effective and innovative features created. Some of these effective innovations that I created consist of the use of arrays and foreach loops to replace multiple if statements, using separate classes to segregate specific sections of the application, as well as the piece moving capabilities.

**Array Implementation**

When generating the pieces to be placed on the board, I initially thought of using either eight IF statements or a switch case which would add each piece onto the board. However, when considering this, I realised that instead of using these eight very repetitive IF statements, I could configure an array, what was called the pos array, which would contain the starting square of each rank’s ID in equivalence to the board array. This meant that nine values would be present within the array as the rank number will never be zero. Instead of using the eight IF statements which would all contain a FOR loop, a foreach loop was used which would cycle through the rank’s contents, with the position value increasing each time a piece was placed. This was effective and innovative as it allowed for an easier readability of the code, as well as less processing power being used for unnecessary loops.

**Classes**

Throughout the entire project, classes for the board, CSVs, Pieces, Puzzle, and SQL related code were utilised to make the overall project easier to work with and read, as well as to modularise the application. Within the Piece class, all of the functions related to making moves and setting up the board on the piece side of things are all present, with the overall implementation of the piece movement working to clear up the overall workspace. When it comes to the Puzzle class, many variables, as well as the entire puzzle mechanic being setup within this space. This form of modularisation will allow for future proofing for any future updates on the application, as the various C# files accompanied by the comments, will allow for an easy understanding of how the application works, and how each part effects the overall application. This feature is effective and innovative as this removes 1000+ lines from residing in the Main form, to residing within their relevant class.

**Conclusion (30 seconds)**

* Final thoughts and a summary of the project's success in solving the initial problem.