Introduction

In the journey of developing our chess program, our team embraced a synergy of diverse talents and perspectives. The essence of this project was not just in coding, but in weaving a tapestry of collaborative efforts, innovative problem-solving, and resilient design strategies. Our daily meetings transcended mere coordination, evolving into brainstorming sessions where creativity and logic intertwined, fostering an environment where every challenge was met with a solution-oriented mindset.

At the heart of our project was the Model-View-Controller (MVC) design pattern, a choice that exemplified our commitment to robust and flexible software architecture. The 'Board' class, the backbone of our chess logic, was the Model, encapsulating the game's state and rules. The 'Controller' class, acting as the command center, controlled the flow of information between the user and the system. And the 'View' class, our interface with the users, translated the game into both a textual and graphical display.

This MVC framework was not just a structural decision; it was a philosophical one. It allowed us to employ a 'divide and conquer' approach, dividing the program into modular components, each meticulously crafted and then seamlessly integrated. This modular design was not only about efficiency; it was about resilience. It empowered us to adapt to changes swiftly, an essential feature in a dynamic landscape of software development.

Overview

...

Design

...

Resilience to Change

...

Answers to Questions

Question 1:

In our chess game project, we implemented a feature for our CPU, particularly at level five, that utilized a standard book of openings. This implementation was a significant part of our strategy to enhance the game's competitive aspect, especially when playing against the computer.

We started by including a vector named `openingBook` in our `LevelFive` class, which inherited from `LevelFour`. This vector stored pairs of positions representing standard opening moves. These moves included a variety of popular openings like the Double King's Pawn, Sicilian Defense, Caro-Kann Defense, and Ruy Lopez Opening, among others. We populated this vector in the `loadOpeningBook` method, which was called in the `LevelFive` constructor.

The `selectOpeningMove` method in `LevelFive` was designed to select an appropriate opening move from the `openingBook`. The method iterated through the stored moves, checking if each move was valid for the current state of the board. If a valid move was found, it was executed, and then removed from the `openingBook` to avoid repetition in subsequent plays. This approach ensured that the CPU, when operating at level five, started the game with strategically strong moves, reflecting the depth of real-world chess knowledge.

If no valid opening move was found, or if the position did not match any of the opening moves in the book, the algorithm fell back to the `LevelFour` logic. This fallback mechanism ensured that the CPU remained competitive even when standard openings were not applicable.

In cases where a valid opening move was available, we compared its score with other possible moves calculated by the `LevelFour` algorithm. The opening move was chosen if its score was better or equal to that of the `LevelFour` move, ensuring that the CPU made the most strategic decision based on both the opening book and the game's dynamic state.

This implementation of the opening book not only made our CPU more challenging but also added an educational dimension to the game, exposing players to widely-recognized chess strategies and encouraging them to think more deeply about their opening moves.

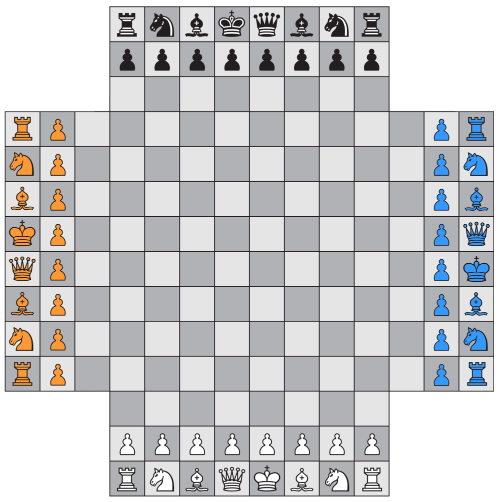
Question 2:

In our chess game project, we implemented a function named `undoMove` in the `Board` class, which played a crucial role in our game's logic. Although it wasn't designed as a player-facing feature for undoing moves, it was instrumental in our game's internal mechanics, particularly in evaluating potential future moves.

The `undoMove` function was primarily used to revert the changes made on the board by temporary moves. These temporary moves were part of our strategy to evaluate the consequences of certain actions in the game, such as predicting the opponent's responses or assessing the safety of a move. The function worked by reversing the effects of a move and restoring the board to its previous state.

The function accepted several parameters: a `Piece* dup` representing a duplicate of the piece that was captured (if any), a boolean captured indicating whether a piece was captured in the move, and `Position` objects `startPos` and `endPos` representing the start and end positions of the move. First, we relocated the moved piece back to its original position (`startPos`). This was done by updating the grid, which represents the chessboard, and setting the piece's position back to startPos. If no piece was captured (`!captured`), we simply set the end position on the grid to `nullptr`, indicating that the square is now empty. If a piece was captured (`captured`), we placed the duplicate piece (`dup`) back at the end position (endPos) on the grid and updated its position accordingly.

Question 3:



In considering the adaptation of our chess game to support a four-handed variant, we identified several key modifications that would be necessary to integrate into our existing codebase. The first major change would involve altering the board's dimensions to accommodate the four-player format. Depending on the chosen style, whether a `12x12` board or a `14x14` board with `3x3` cutouts in each corner, we would adjust the `boardSize` constant appropriately. For the latter style, we'd introduce a new constant `invalidSquares`, a `vector<Positions>` to represent the unplayable squares in the corners.

Additionally, the game's mechanics would need to be expanded to support four players. This would involve constructing two more player fields in our `Controller::run` method, adapting the start method to initiate a game with four players, and modifying the `switchTurns` method to cycle through four players instead of two. Furthermore, we would extend the functionality of our `isCheckmate`, `isCheck`, and `isStalemate` methods by adding an integer parameter to assess the game state relative to each player pair.

Another critical aspect would be the introduction of new win conditions suitable for a four-handed game. This would include implementing a `hasWon` method which would process the board to determine the winners and possibly rank them, accommodating the possibility of multiple winners in this format. Additionally, the `Colour` string for the pieces would be expanded to include new colours, such as "`orange`" and "`blue`", to differentiate the additional players.

Extra Credit Features

In our quest to exceed the standard requirements, we integrated two exceptional extra credit features into our chess program, elevating its complexity and user experience.

Standard Mode: This feature epitomizes user convenience and adherence to classical chess rules. Upon selection during the setup phase, the program automatically initializes a chessboard, arranging the pieces in their standard positions according to traditional chess rules. This mode ensures a seamless transition for users into the game, especially for those familiar with the conventional setup of chess. This is implemented by creating a new option within the controller class code which allows the user to select 'standard', upon selecting standard the function `standardBoardSetup()` is called, which is a function contained in the board class. In turn, this function handles the piece placing logic on the actual board's grid.

Level 5 AI with an Opening Book: The crown jewel of our extra credit features is the Level 5 AI, a sophisticated advancement over its predecessor, Level 4 AI. This AI utilizes an innovative opening book strategy, enhancing the game's complexity and providing a challenging experience for advanced players.

The essence of the Level 5 AI lies in its use of a meticulously curated opening book, a collection of strategically selected chess moves designed to give the AI a competitive edge in the early stages of the game. These opening moves encompass a range of well-known strategies, from the aggressive King's Pawn Opening to the more subtle Queen's Gambit. This book of opening moves is constructed using the `loadOpeningBook()` function, in turn this function populates the vector, `vector<std::pair<Position, Position>> openingBook`, where `Position` is a struct contains an x and y coordinate with the first position in the pair being the starting position, and the second position in the pair is the ending position.

The AI is programmed to select the most appropriate opening move based on the current board configuration, ensuring a dynamic and unpredictable game experience.

Underneath its sophisticated facade, the Level 5 AI is built upon the robust foundation of the Level 4 AI, inheriting its advanced algorithms and decision-making capabilities. However, it distinguishes itself by the inclusion of the `selectOpeningMove()` function. This function intelligently assesses the board and chooses an opening move from the book. The Level 5 AI assigns a score to each opening move (defaulted at 8), if a valid move aligns with the AI's color and meets the game's rules, its score is compared with all the other moves it can do other than an opening move such as, capturing opponent pieces and avoiding capture. If the opening move has the highest score, that move is executed, if not the move with the highest score is executed.

This blend of predefined strategic openings and adaptive mid-game tactics enables the Level 5 AI to simulate a more human-like and challenging opponent. Players facing this AI must not only contend with its sophisticated mid-game strategies but also navigate through its diverse array of opening moves, each designed to set the tone for a strategically complex and engaging game.

Final Questions

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

Working on the chess game project in a team of three taught us several valuable lessons about developing software in a collaborative environment. One of the first things we learned was the importance of clear and consistent communication. Our team frequently discussed our progress and challenges, which helped us avoid misunderstandings and ensured we were all aligned with our goals.

We used GitHub for version control, which was crucial for managing the different parts of the code we were working on simultaneously. It made it easier to integrate these parts without much hassle. This experience highlighted the importance of familiarizing oneself with tools like Git, which are essential in modern team-based software development.

Another key lesson was the importance of efficient project management and the division of labor. We broke down the project into smaller tasks and assigned them based on our individual strengths and interests. This approach not only made the workload manageable but also allowed us to learn from each other by working on different aspects of the project.

The project also significantly enhanced our adaptability and problem-solving skills. Working in a team requires flexibility and readiness to tackle unexpected issues or changes in project requirements. We encountered various challenges, such as integrating different parts of the code, optimizing performance, and ensuring the game's user interface was intuitive. Addressing these challenges required us to think creatively, explore new solutions, and sometimes revise our initial plans.

Lastly, the project underscored the importance of good documentation and clear comments in the code. Since the project was a collaborative effort, ensuring that our code was understandable to all team members was essential for efficient progress.

2. What would you have done differently if you had the chance to start over?

Reflecting on our approach to the chess game project, one significant oversight was our decision not to compile our code until the third day. In the initial enthusiasm of development, we were focused on writing and integrating various parts of the code, but we delayed the compilation process. When we finally compiled, we were met with a multitude of errors and bugs. This situation was a stark reminder of the importance of frequent compilation and testing in software development. The errors ranged from simple syntax mistakes to more complex logical errors, which were challenging to debug due to the volume of untested code. This experience taught us a valuable lesson about the necessity of regularly compiling and testing code, even in its early stages.

In hindsight, while our creation of a timetable with due dates, deadlines, and goals for the chess game project was a step in the right direction for team organization, we realized that it lacked a crucial element: leeway for unexpected delays. Our schedule was tightly packed, and we adhered to it rigorously, but we didn't account for the inevitable uncertainties and challenges that often arise in software development. There were instances where we fell a day or two behind our set goals, mainly due to unforeseen complexities in coding certain features or integrating different parts of the game. This led to multiple delays and increased pressure on the team. If we were to do it again, we would factor in a buffer period for each phase of the project, allowing for some flexibility. Incorporating this leeway would have not only reduced stress but also provided us with the necessary time to tackle issues more thoroughly, ensuring a smoother and more manageable project flow.

Conclusion

...