Design Document for Chess

By: Ragavan Ravendran, Ashwin Vigneswaramoorthy, Shivam Suri

Introduction & Overview

In the journey of developing our chess program, our team has embraced a synergy of diverse talents and perspectives, creating not just a codebase but a tapestry of collaborative efforts, innovative problem-solving, and resilient design strategies. This project transcends mere coding; it represents a harmonious blend of strategic design, robust architecture, and innovative features. Our daily meetings evolved into brainstorming sessions where creativity and logic intertwined, fostering an environment where challenges were met with solution-oriented mindsets.

Central to our project is the Model-View-Controller (MVC) design pattern, a foundational framework that underpins the entire program and provides a clear separation of concerns:

Model - The 'Board' Class: At the heart of our chess logic, the Board class encapsulates the game's state and rules. It acts as the central hub of game data, ensuring that the state of the chessboard is always current and accurate.

View - Interface Display: The View class and its subclasses present the game to the user, offering both textual and graphical displays. This flexibility in presentation exemplifies our program's user-centric design, ensuring an engaging and intuitive experience.

Controller - User Interaction: The Controller class manages the flow of information between the Model and View, interpreting user inputs and translating them into actions on the Board, thus maintaining the program's responsiveness and interactivity.

Beyond the MVC framework, our design incorporates the Factory Pattern to manage different player difficulties, allowing dynamic player creation and showcasing our program's scalability and flexibility. Other notable features include a sophisticated mapping in the Player class for tracking potential moves, efficient piece count tracking, and the undoMove function in the Board class for retracting moves, enhancing the program's adaptability and user-friendliness.

A key highlight is the Level 5 AI with an opening book feature, which elevates the game's complexity and offers a challenging experience for advanced players. Along with additional features like the Standard Mode and extra credit implementations, our project is a testament to our commitment to creating a resilient, adaptable, and user-centric application.

Design

In the design of our chess application, we implemented the Model-View-Controller (MVC) design pattern to achieve a well-organized and modular structure. The `Board` class acts as the model, capturing the state and notifying observers about changes. On the view side, we have the `View` class and its subclasses, with `TextDisplay` and `GraphicsDisplay` as views, responsible for displaying the game state. The controller, represented by the `Controller` class, handles user input, interprets commands, and manipulates the board accordingly. This design ensures a clear separation of concerns, encapsulating functionalities within each component. The abstraction introduced through interfaces, like the `View` interface, allows for flexibility in creating different views without affecting the core functionality. Our implementation of the MVC pattern provides a scalable and maintainable architecture.

In order to implement different `Player` difficulties, we also utilized a Factory Pattern. The `Computer` class serves as the base class, defining the common interface for all CPUs. Each level of computer player, inherits from Computer and provides a unique implementation of the algorithm for making moves. `levelOne` being the easiest and `levelFive` being the hardest. The Factory Pattern is applied in the `Player` class, which is responsible for creating instances of the `Player` based on the specified player type. This pattern allows us to encapsulate the creation logic and promote code flexibility. In the `Player` class, we determine the player type requested and initialize the appropriate player level. For example, the `LevelOne`, `LevelTwo`, `LevelThree`, `LevelFour`, and `LevelFive` classes extend the `Computer` class and implement their own move evaluation algorithms. Each algorithm returns the best possible move given the intelligence of the computer.

Resilience to Change

Our software architecture is meticulously designed to ensure adaptability and resilience against evolving requirements. We have strategically implemented design patterns like MVC (Model-View-Controller), Observer, and Factory to create a robust and flexible structure. This approach significantly enhances our program's ability to adapt seamlessly to changes in the assignment specification.

A key feature of our design is the innovative use of class members to bolster this adaptability. For instance, in the Player class:

Map of Available Moves: We've incorporated a `map<Position, vector<Position>> availableMoves` field. This sophisticated mapping not only tracks each player's potential moves but also supports additional features. A prime example is the implementation of a 'guided mode' for beginners. In this mode, legal moves for each piece are highlighted, aiding new players in understanding game dynamics. The map is dynamically generated by the `renderAvailableMoves(Board* board)` function, ensuring that it only includes legal moves, such as those not putting the king in check or invalid moves.

Piece Count Tracking: The `unordered_map<char, int> pieceCount` field efficiently keeps track of the number of pieces each player possesses. This information is pivotal for developing a function that graphically displays the pieces each player has lost during a game, enhancing the user interface and game experience.

Undo move: A critical aspect of our chess program's adaptability is the ability to reverse actions, allowing players to retract their last move. This feature is facilitated by the undoMove function within our Board class. The signature of this function is `undoMove(Piece* dup, bool captured, Position startPos, Position endPos)`, and it operates as follows:

The function first identifies the piece that was moved to the endPos location and relocates it back to its original startPos. This is achieved by updating the grid's references to the moved piece and resetting its position, thereby effectively reversing the move.

If the moved piece did not capture another piece (captured is false), the end position on the grid is simply set to nullptr, indicating that it's now empty. However, if a piece was captured (captured is true), the function restores the captured piece (dup) to its original position (endPos). The grid is updated to reflect this, and the captured piece's position is also reset.

This function plays a pivotal role in enhancing user experience by allowing players to retract their moves, which is particularly beneficial in casual play or learning scenarios. It also demonstrates the program's capacity for change and adaptability. By designing the function to handle both simple moves and complex scenarios involving piece captures, our program ensures a high level of flexibility and robustness.

Implementing the undo feature based on this function can significantly improve the game's usability, especially for beginners or players who are experimenting with different strategies. It opens up possibilities for additional features, such as move history tracking, replay functionality, and more, further illustrating our commitment to a design that is resilient to change and adaptable to new requirements.

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* du`p 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

The journey of creating our chess program has been an educational experience, encapsulating not only the technical prowess required for software development but also the invaluable lessons of teamwork and project management. This project has not only produced a robust and enjoyable chess application but has also been an opportunity for growth as software developers and peers.

The implementation of the Model-View-Controller (MVC) and Factory design patterns, along with the integration of features such as player difficulties, an undo move function, and a level 5 AI with an opening book, demonstrates our commitment to creating a resilient, adaptable, and user-friendly application.

Furthermore, the project has been a testament to the importance of frequent communication, efficient project management, and the division of labor based on individual strengths. The challenges faced and overcome, such as integrating different code components and addressing unforeseen complexities, have not only honed our problem-solving skills but also highlighted the significance of regular testing and the inclusion of buffer periods in project timelines.

As we conclude this project, it stands as our collective effort, creativity, and dedication. It is a reflection of our ability to synergize diverse skills and ideas, overcoming challenges through a solution-oriented approach. The experience gained from this endeavor will undoubtedly serve as a solid foundation for our future endeavors in the realm of software development.