Final Report

Introduction

This project involves the development of a chess game application that adheres to standard chess rules and supports both human and computer players. The game offers an interface displaying the game board and pieces. The application is designed to accommodate different levels of computer player difficulty and supports various game commands and modes.

Overview

The overall structure of the project is built around several key classes, each responsible for different aspects of the chess game. The primary classes include Game, Board, Piece, Player, and various specialized pieces such as King, Queen, Bishop, Knight, Rook, and Pawn. The Game class orchestrates the gameplay, managing turns, piece movements, and the game state. The Board class handles the layout and state of the chessboard. The Piece class and its subclasses encapsulate the behavior and movement rules of each type of chess piece. The Player class represents the participants in the game, which can be either human or computer players. Additionally, the system includes observer classes for updating the display, whenever the game state changes.

Design

The design of the chess game leverages object-oriented programming principles to ensure high cohesion and low coupling across its components. This structure facilitates maintainability, scalability, and adaptability.

Piece Abstract Class

Each piece subclass (King, Queen, Bishop, etc.) encapsulates its specific movement rules. Methods like canMove and listOfEndPositions define how pieces move according to chess rules, including handling special moves like castling, en passant, and pawn promotion. This design

ensures high cohesion as each class is responsible solely for the behavior of a specific piece, making the code easier to understand, modify, and extend.

Game Class

The Game class manages the overall game state, including turn-taking, checking for checkmate, stalemate, and check conditions. It uses methods like isGameOver, isValidMove, and makeMove to control the flow of the game and enforce rules. This class serves as a central coordinator, ensuring that the game progresses correctly and adheres to the rules. By delegating piece-specific logic to individual piece classes, the Game class maintains low coupling, focusing only on the game's high-level logic and state management. The Game also works in tandem with the command interpreter to facilitate moves between players and computers.

Observer Pattern for Display Updates

To keep the display consistent with the game state, the observer pattern is employed. The Board class maintains a list of observers (GraphicsObserver and TextObserver) that are notified of changes to the board state. This ensures that any update to the game state is immediately reflected in the display, whether it's a graphical interface or a text-based console. The use of the observer pattern promotes low coupling because the Board class does not need to know the specifics of how the display works—it only needs to notify the observers of changes. It also encapsulates all rendering logic ensuring code maintainability.

Computer Player Levels

The design employs a strategy similar to the factory method pattern to create instances of different levels of computer players. Instead of having a separate factory class, the createComputer function is implemented in the main file. This function dynamically creates computer players based on the specified difficulty level. The Computer class and its subclasses (Level1, Level2, etc.) implement varying strategies for move selection, using methods like getRandomMove and getRandomPiece.

This approach ensures high cohesion within each computer player subclass, as each one is responsible for its strategy logic. It also promotes low coupling, as the game logic does not need

to change when adding a new level of difficulty—the createComputer function can simply instantiate the new subclass. This design enhances maintainability and scalability, allowing new levels of computer players to be added seamlessly.

Resilience to Change

The design of the chess game is highly adaptable to potential changes and extensions due to several key features. First, the modular design encapsulates piece-specific logic within subclasses of the Piece class. This modular approach ensures that changes to movement rules or the addition of new piece types can be made seamlessly without affecting other parts of the codebase. For instance, if a new piece with unique movement capabilities were to be introduced, it could be implemented as a subclass of Piece with its specific movement logic, leaving the existing system unaffected. This approach enhances maintainability and makes the system more extensible.

Moreover, the use of the observer pattern for display updates significantly enhances the system's flexibility. Observers such as GraphicsObserver and TextObserver are notified of changes to the board state through a well-defined interface, ensuring that the display remains consistent with the game state. This pattern decouples the board logic from the display logic, allowing for easy modifications or additions of new display types without altering the core game logic. For example, if a new graphical interface or an enhanced text-based display were to be introduced, it could be added as a new observer, seamlessly integrating with the existing notification mechanism. The notify method in the Board class calls the update methods of all registered observers, ensuring that any state change is immediately reflected in all displays.

In addition, the flexible command interpreter within the Game class is designed to be easily extendable. This interpreter uses a command pattern, where each command is encapsulated as an object with a specific execution method. New commands or variations of existing commands can be added by simply creating new command classes that implement a common interface. For instance, new commands for advanced features like custom game setups or additional game modes can be incorporated without disrupting the existing functionality. The command interpreter then dynamically invokes these commands based on user input, making the system

highly adaptable to different input methods or game modes. This flexibility ensures that the game can evolve and incorporate new features or respond to user feedback effectively.

Furthermore, the design employs a factory-like method for creating instances of different levels of computer players. Although not a strict implementation of the Factory Method pattern, the createComputer function in the main file dynamically instantiates the appropriate computer player based on the specified difficulty level. The Computer class and its subclasses (Level1, Level2, etc.) encapsulate the logic for different AI strategies, using methods like getRandomMove and getRandomPiece to determine moves. This approach ensures that the instantiation logic is centralized and easily modifiable, enhancing code maintainability and scalability. By using this factory-like method, new levels of computer players can be added without modifying the existing instantiation logic, promoting an open-closed principle where the system is open for extension but closed for modification.

Overall, the design's modularity, use of the observer pattern, flexible command interpreter, and factory-like instantiation method collectively contribute to a robust and adaptable chess game system. This architecture not only simplifies the current implementation but also paves the way for seamless integration of new features and improvements, ensuring the system remains flexible and maintainable over time.

Answers to Questions

Q1: Standard Openings

To implement a book of standard openings in the chess program, the initial step involves constructing a comprehensive database of opening moves and their corresponding responses. This database should incorporate historical win/draw/loss percentages for each opening sequence, providing the computer player with data-driven strategies for the early phase of the game. During gameplay, the computer player references this database to select moves that align with established opening strategies, ensuring the opening phase follows well-known patterns and enhancing the computer player's effectiveness.

The key idea is to encapsulate this functionality within a dedicated class that manages the selection of openings based on the moves made by both the player and the opponent. This class can act as a decorator on top of the standard AI or human player class, allowing it to function as a "regular" AI or human player after the standard moves are processed. This approach minimizes changes to the Board class and only requires minor modifications to the Controller class to accommodate the decorator pattern. By using interfaces, the Controller and Board classes do not need to differentiate between standard openings, user input, or AI moves—they simply rely on the move() method from the public interface.

The Opening class can represent each opening, encapsulating essential information such as initial moves and possible responses by the opponent. The initial moves can be stored in an array or vector of strings, representing the sequence in algebraic chess notation (e.g., e5, Nf3). Similarly, the opponent's responses can be structured in a comparable way. A map can be used to store instances of these openings for efficient retrieval and insertion. The Opening class can be populated manually, algorithmically, or loaded from an external source file, including popular openings like the King's Gambit, Ruy Lopez, and the Italian Game for White, and defenses such as the French Defense, Caro-Kann Defense, and Sicilian Defense for Black.

A search algorithm can run during the game to match the current board position with entries in the opening book, identifying the closest match and potential variations. This information can then be displayed to the player, providing insights into the current opening being played and suggesting subsequent moves. Optionally, the program can update opening game statistics based on actual game outcomes, tracking win, draw, and loss percentages for each opening to keep the database current.

The user interface should display the current opening name and suggest moves based on the opening book, offering players insights into professional opening strategies. An option to turn off the opening book should be provided, allowing players to practice without preset strategies. Periodic updates from online databases or community contributions can keep the opening book relevant and up-to-date, ensuring it evolves with contemporary chess knowledge.

Q2. Undo Feature

Implementing an undo feature in the chess program requires the use of a stack data structure to store the history of moves. Each move is encapsulated in a `Move` object, which contains all necessary information such as the starting and ending positions, any captured pieces, and any special conditions like castling or en passant. This `Move` object is pushed onto the stack, creating a chronological record of all moves made in the game.

To perform an undo operation, the program pops the most recent `Move` from the stack and reverses its effects on the board and game state. This involves restoring the pieces to their previous positions and reintroducing any captured pieces back onto the board. For unlimited undos, the stack grows as needed, accommodating an unrestricted number of moves.

The user interface includes a clear command for undoing moves, with optional confirmation prompts to prevent accidental reversals. This feature allows players to correct mistakes and experiment with different strategies without restarting the game, enhancing the overall user experience by supporting a more forgiving and exploratory gameplay environment.

Q3. Four-Handed Chess

Creating a four-handed chess game involves several significant modifications to the existing two-player structure. Firstly, the number of players would be increased from two to four, requiring the Game class to be updated to manage four players instead of two. The Player class would also need adjustments to accommodate additional players, potentially including new attributes to manage team affiliations or player turns.

The board size would be expanded to accommodate the extra players, likely resulting in a larger grid with additional squares. The Board class would be updated to handle this new layout, and the display logic within the GraphicsObserver and TextObserver classes would be enhanced to support the larger board and more complex player interactions. This could involve redesigning the graphical interface to clearly differentiate between the four sets of pieces and their movements.

User input handling would be modified to allow inputs from all four players, which might include changes to the command interpreter to recognize and process commands from multiple sources. Additionally, the AI logic for computer players would need to be significantly updated to consider moves from all four players. This would require complex decision-making algorithms that account for the interactions between four sets of pieces and the strategic implications of moves from each player. Overall, transforming the game to support four-handed chess would involve extensive changes across multiple components of the system, ensuring seamless integration and smooth gameplay for all participants.

Extra Credit Features

For Chess, the extra credit features available were the use of smart pointers (efficient memory management) and an advanced computer component difficulty (level4). Our group initially decided that we would implement smart pointers to help with memory management. The plan was to create a working version of the game and then alter any data fields that were stored on the heap to be managed with smart pointers. Since our whole project was built without smart pointers, we found trying to reincorporate them back in was challenging. We discovered that using smart pointers requires careful planning of classes in advance to accommodate the selectivity of who gets to access the data. Ultimately, our solution was to revert back to regular memory management with “new” and “delete” to not force smart pointers into our project. For the computer opponent, we planned to make a bot with the best possible skill level given the time available. At first, we used the power of random numbers to select moves from a list of pseudo moves that we stored in each piece. To upgrade this logic, we tried prioritizing capturing moves. The main challenge was to get the bot to avoid getting it’s own pieces captured. To overcome this we developed a method that simulated an “undo” like simulation to see what events might transpire after a move.

Final Questions

Q1: 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?

A1: Initially when our group was in the planning stage we were able to neatly organize who would develop what features and in what timeframe. When we actually started developing, we found that keeping up with the deadlines we set was extremely difficult. We ran into many unforeseen circumstances which can mostly be attributed to the fact that all three of us had different ideas and opinions on design ideas. Our group would discuss strategies often but when it came time to implement we would all go about it differently. This taught us that everyone thinks differently and if you can objectively judge each opinion you can enhance each others strengths.

Q2: What would you have done differently if you had the chance to start over?

A2: If we had the chance to start over we would reprioritize certain elements of the game. As was suggested in the project guidelines we would go back and create a working command interpreter first. From there we would build out features one by one, while making sure the existing features did not have conflicts and remained operational. Our largest challenge was figuring out how to test our code because our command interpreter was not done first so we were not able to compile and test our code until the very end. Another strategy we would employ if we could start earlier would be to implement smart pointers. We found it difficult to track down memory leaks and dangling pointers. As the project scaled to be larger and larger the time spent debugging grew exponentially. If we had used better memory management principals like smart pointers we would have saved much more time.

Conclusion

This chess game project combines object-oriented design, modularity, and advanced algorithms to create a robust and flexible chess application. By addressing the key challenges in movement rules, game state management, display updates, and computer player strategies, the system provides a comprehensive platform for playing chess. The design's resilience to change ensures that the game can be easily extended and adapted to new requirements, making it a versatile tool for both casual play and advanced study of chess strategies. The project was a brilliant learning experience and provided intellectually challenging problems.