Family Name: r26li Family
Partners: r26li, rjeon, z75wang
Game: Chess

Introduction

This documentation outlines the design and the choices we made while implementing the game chess. The program is made utilizing C++ and the observer design pattern taught in the course CS246 in Fall 2023. We will discuss our updated UML while comparing to our initial version, then delve deeper into the superclasses and their subclasses to discuss their purpose and functionalities they hold. Then finally, we will be answering initial questions that were asked in the project guideline to end off our documentation.

Overview

The structure of the project consists of the Board class that owns the TextDisplay, GraphicsDisplay, and the Piece class, the Piece class is the parent class of the 6 type of pieces, aas well as a none-type piece for distinguishing a piece between a valid/invalid piece later. The player handling is done with the Player parent class and its subclasses Human, Computer, and the four Level classes.

Main

Our main.cc file is where the program begins, since it is the initial part of the program where the board is initialized by the user, and hands users the instructions about the game. The file handles user input, calls the corresponding functions to the input, and prompts the user for invalid commands or inputs. When the game begins, the main function generates a graphics window (an Xwindow object), a Board object, and initializes a Color variable to keep in track of which player's turn it is (between black and white). Then according to the user's commands, it creates a standard board, or creates a customized board that starts our blank.

The commands for the main include:

- setup, where user can initialize a customized board
- game, where user can start the game with their desired player
- move, where user can move a place around
- undo, where user can undo their last move

Board

The main part of our program that handles one of the core functionalities is the Board class. This class owns a grid of Pieces, and has the move() methods and other methods that check for the different formations of the pieces, that determines a win/loss.

1. **Move**

The functionality of moving a piece from its initial position to the new position set by the user happens in this class. When the user inputs the command “move”, the main sends the cin to the move() function of the player to generate all the valid new positions from the old position the player can make, then the Board parses through all of the possible new positions to check if the new position is one of the valid positions that the player can make.

2. **Checkmate**
   The checkmate functionality in our chess program is implemented through the checkMate() function in the Board class. This function takes a color as an argument and determines whether the current state of the board results in a checkmate for the specified color. The checkmate condition is met when the player is in check (inCheck function) and there are no valid moves left for the player to escape from check (staleMate function). The checkMate function utilizes the inCheck and staleMate functions to make this determination. If the player is in check and there are no valid moves that can be made to escape check, then the function returns true, indicating that the player is in checkmate. The inCheck() function checks whether the specified color's king is currently in check. It does so by updating the danger zones for both the black and white pieces and then checking whether the king's position falls within the danger zone.
3. **Stalemate**
   The staleMate function, on the other hand, determines if the player is in a stalemate. It iterates through all the player's pieces and their possible moves. If, for every possible move, the player is still in check, then the function returns true, indicating that the player is in stalemate. These functions collectively contribute to the checkmate functionality in the chess program, ensuring that the game correctly identifies when a player is in a position where they have no valid moves to escape checkmate.
4. **Check**
   The program implements the check functionality through the inCheck() function in the Board class. This function takes a color as an argument and checks if the king of that color is currently in check. The inCheck() function updates the danger zones for both black and white pieces and then checks whether the position of the specified color's king falls within the danger zone. If it does, the function returns true, indicating that the king is in check. This function is essential for detecting when a player's king is under immediate threat, allowing the program to respond accordingly during the game. The updateDangerZone() function is called within inCheck() to refresh the danger zones for all pieces on the board, ensuring accurate information for determining the check condition.

Player – Human, Computer (Level 1,2,3,4)
This Player class is an abstract base class to the classes Human and Computer (Level 1,2,3,4).

The Human class enables human players to input moves via console. The getNextMove function reads and processes the input, constructing a Move object that represents the intended move. The Computer class, representing a computer player, uses a basic heuristic to evaluate and select moves. The getNextMove function creates potential moves for each piece on the board, considering threats and capturing opportunities. The public move() function then uses the move returned by getNextMove to make the move on the board, following the non-virtual idiom. Different Level classes implement a scoring mechanism for move evaluation. These classes showcase the flexibility of the design, allowing for diverse player types in a chess game, each with distinct decision-making processes.

Piece
The Piece abstract class serves as the foundation for various chess pieces, providing common functionalities shared among them. Each piece, including the `King` subclass, inherits from this class. The `Piece` class contains methods to retrieve information about a piece, such as its position, color, and capture status. It also includes mechanisms for observing and notifying observers about moves. The `King` subclass, specifically, implements its own `getPossibleMoves` method, determining the valid moves for a king on the chessboard. Additionally, it features a `castling` function, allowing the king to perform castling under certain conditions. This design facilitates code modularity and extensibility, enabling easy integration of new chess piece types with shared functionalities.

TextDisplay
The TextDisplay class provides a textual representation of the chessboard and its pieces. The constructor initializes an 8x8 grid, and the pieceToChar method translates the piece type and color into their corresponding ASCII values. The display gets attached to each piece in the grid, then updates the display whenever it gets notified. The `notify` method updates the display based on the given move, adjusting the characters at the old and new positions accordingly. Empty squares are represented by spaces or underscores based on the square's color. The overridden *operator<<* facilitates the output of the entire chessboard to the designated output stream. The class follows the Observer pattern, with the `notify` method responding to changes in the board and updating the display accordingly. This design allows for a clear and concise visualization of the ongoing chess game within a terminal or console environment.

GraphicsDisplay
The GraphicsDisplay class is designed to visually represent the chessboard and its pieces using a graphical user interface (GUI) with the help of the *Xwindow* library. The constructor initializes the graphical window, displaying the numeric and alphabetic coordinates along the sides of the board. The *pieceToChar* method translates the piece type and color into their corresponding ASCII values. The notify method responds to changes in the chessboard, updating the graphical display accordingly. It calculates the pixel coordinates of the old and new positions based on the

grid, fills the corresponding rectangles with the appropriate background color (white or black), and renders the moved piece using the *Xwindow* library. The color of the piece is determined by its player color, with black pieces shown in red and white pieces in blue.

Design

Please refer below for the updated UML diagram.

Changes that were made

The revised UML diagram incorporates significant enhancements to the Board class, introducing additional functions to manage diverse piece formations, including scenarios like stalemate, promotions, and checks. A pivotal addition is the implementation of the `getNextMove` function in the Player class, facilitating the retrieval of the potential next move for a player's piece. To cater to distinct levels of computer opponents and their varying behaviors, we've introduced four subclasses under the Computer class. This ensures compliance with level-specific requirements. In terms of visualization, we've extended the functionality of all pieces by integrating the `notify` function. This function notifies observers, namely TextDisplay and GraphicsDisplay, ensuring real-time updates whenever a piece is moved or captured. These modifications enhance the flexibility, functionality, and observer pattern implementation within our design.

Model-View-Controller Architecture

Our system architecture adheres to the Model-View-Controller (MVC) design pattern, providing a clear separation of concerns and promoting maintainability. The MVC architecture is structured into three main components:

1. **Model** (Board, Player, Piece)
   - Board Class: Represents the game state and logic. It encapsulates the rules, manages the pieces, and handles various game scenarios. This serves as the core model of our system.
   - Player Class: Represents the players in the game. The abstract Player class defines the common interface for both human and computer players, acting as a model for player-related functionalities.
   - Piece Class Hierarchy (King, Queen, etc.): Represents the individual chess pieces. The hierarchy allows for extensibility, accommodating new pieces with minimal impact on the existing codebase.

2. **View** (TextDisplay, GraphicsDisplay)
   - TextDisplay Class: Implements the textual representation of the game state. It observes changes in the pieces and updates the display accordingly. This adheres to the observer pattern, enhancing the separation of concerns.
   - GraphicsDisplay Class: Represents the graphical display of the chessboard. Like TextDisplay, it observes the changes in the pieces and updates the graphical

representation. The observer pattern ensures loose coupling between the model and the view.

3. **Controller** (Player)

- Player Class (and its subclasses): Part of the controller layer, responsible for obtaining moves from the users or computer players. The abstract Player class defines the common interface, ensuring consistency in how moves are obtained.

Observer Pattern Integration

The observer pattern is integrated into our MVC architecture to enable automatic updates in the views whenever there is a change in the model. The pieces (subjects) are observed by both TextDisplay and GraphicsDisplay (observers). When a piece changes its state, it notifies its observers, triggering the views to update accordingly.

This integration enhances the flexibility of our design, allowing for easy addition of new display types without modifying the core logic. It also ensures that changes in the model do not directly affect the views, promoting a more modular and maintainable codebase. Overall, the combination of MVC and the observer pattern provides a scalable and adaptable architecture for our chess program.

Resilience to Change

Our design exhibits a high degree of resilience to changes in the program specification, providing flexibility and adaptability to accommodate various modifications. The use of polymorphism and abstraction principles allows for easy addition or modification of pieces and player types without impacting the core structure. For instance, the Player class serves as an abstract base class, enabling the introduction of new player types by extending this class.

The observer pattern, implemented with the TextDisplay and GraphicsDisplay classes, enhances the adaptability of the system. The decoupling of subjects (pieces) and observers (displays) ensures that adding new display types or altering existing ones does not require changes to the core piece logic. Additionally, the notify function enables seamless integration of new display functionalities without affecting the existing codebase.

The Board class, designed to handle various game scenarios, supports extensibility. New functions can be added to address additional rules or game states without disrupting the existing functionalities. This modular approach facilitates the incorporation of rule changes, promotions, or additional game logic.

The creation of different Computer subclasses allows for straightforward adjustments or additions to computer player behaviors. New levels or variations in computer player strategies can be introduced by extending or modifying these subclasses.

Questions from DD1

Question 1

Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. See for example https://www.chess.com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

Implementing a book of standard opening move sequences in our chess program would involve creating an *Openings* class to store and manage recognized opening strategies. The opening book would be populated with well-known opening sequences, and the Board class would be modified to check for matches against these sequences during gameplay. Computer players, specifically the Computer class, would be enhanced to consider the opening book when selecting moves, allowing them to follow predefined sequences up to a specified depth. The opening book can be dynamically updated to incorporate new strategies, and the program should handle deviations or transpositions from known openings. Logging mechanisms record game outcomes for analysis, and user interaction options enable players to choose whether the computer utilizes the opening book.

Question 2

How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?

The implementation of an undo feature in the chess program would involve maintaining a comprehensive move history, storing crucial details about each move and capturing the entire game state after each move. A vector would facilitate the storage of moves and their corresponding game states. The undo functionality allows players to reverse their last move, restoring the game state to its previous condition. To support unlimited undos, players can iteratively undo moves until reaching the initial game state. Integration of redo functionality allows players to move forward through the move history. The update of game displays, such as TextDisplay and GraphicsDisplay, ensures that the user interface accurately reflects the current game state after each undo or redo operation. This approach provides a resilient mechanism, allowing players to navigate through their moves seamlessly while preserving the integrity of the chess game.

Question 3
Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.

- Board Representation: The Board class will need to be updated to represent this new board layout and manage the starting positions of pieces for four players.

- Game Logic: The Board methods for move validation move(), and win condition checking must be updated to reflect the new movement patterns and victory conditions of four-handed chess.

- Player Management: The system will need to handle four Player instances instead of two. Turn management will have to be updated to cycle through four players.
  The Player class may need additional attributes or methods to handle alliances or interactions between players

- User Interface: Both TextDisplay and GraphicDisplay attributes in the GameBoard class must be adapted to render the larger and uniquely shaped board.

- Piece Class: The Piece class (and its subclasses) may require updates to the possibleMove() method to handle new types of moves or piece behaviors specific to the four-handed variant.
  Game Rules: Any hard-coded rules will need to be revisited and updated to align with the four-handed variant's rule set. This could involve:

- The Computer class AI would need to be significantly more complex to handle strategy and tactics in a game with three opponents.

Final Questions

Question 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?

When developing software in a team, the primary lesson learned is the importance of clear and consistent communication. A team must align on goals, timelines, and responsibilities. This experience teaches the significance of version control systems like Git, which allow multiple people to work on the same codebase simultaneously without conflicts. Regular meetings and updates are crucial to maintain momentum and address issues promptly. The biggest takeaway from writing a large program is that a lot of testing and debugging needs to be done every time when new method(s) and logic are written. Therefore it’s important to set milestones and write test cases to make sure new changes won’t break any previous tests.

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

We could use smart pointers (shared_ptr or unique_ptr) for ownership, which would allow us to reduce the time and effort put into preventing memory leaks and implementing exception safety. We could also make a cell class to be owned by the board to make it easier to implement the text and graphic displays. We could have a game class to control the flow of the game so that the implementation of the main function could be easier and the coupling could be reduced. We would also find a better method to handle chess moves to make implementing en passant and castling easier.