CS246 Final Project Design

Sourish Das & Andy Jung

1 Introduction

Throughout the past week, we began the daunting task of creating a deliverable, fully functional chess game that incorporates the principles of C++20 and object-oriented programming. Although the journey was certainly not the easiest and we faced a lot of difficulties, especially when trying to debug some rather trivial mistakes, it really helped us grow our understanding of developing software in a collaborative effort.

2 Overview

Our chess game is structured around several key classes and components that work together to create a cohesive and functional system. Each class plays a crucial role in managing different aspects of the game, from representing the chessboard and pieces to handling player interactions and displaying the game state.

Our chess game is structured around several integral well-designed classes that all work together to create a cohesive and functional system. We wanted each class to manage a specific task in the game, and also be quite independent of each other to maximize cohesion and minimize coupling minimize coupling. Although we had a very strict structure set out when we first designed the premature UML more than a week ago, however as time passed, and the needs changed, we had to adapt our structure quite a lot as well.

Initially, we presumed that we needed to utilize the Decorator Design Pattern for all the pieces in the game. However, when we began coding the first version of our game, we ran into issues of trying to initialize objects of our abstract classes. After further research and thought, we realized that the pieces of a chess game don't require a Decorator Design Pattern whatsoever, since the moves of each piece are well defined in the rules of the game itself, and the behaviour does not change dynamically throughout the game for the most part. The only instance where the functionality is extended in a sense, is when a pawn is promoted, however, even in that instance the new functionality is well defined, and that of a select few other pieces.

Our other major change was the scrapping of a class that will be responsible for tracking the game's move history. Since all the moves were being carried out in the board class, we shifted the contents of our originally proposed game history class to there. This allowed for us to avoid the creation of extra classes that do not serve a meaningful purpose.

Apart from these two major changes, the structure of the originally proposed UML was not altered too much in our final version. Here's a detailed overview of the main components:

Board: The `Board` class is the central component of our chess game, managing the 8x8 grid of squares and the pieces placed on them. It is responsible for initializing the board, placing and

moving pieces, and checking for special game conditions such as check and checkmate. The board also provides methods for querying the state of individual squares and pieces, ensuring that all game logic is centralized and easily accessible. Also, board is responsible for notifying the observers to update the game visually.

Square: Each square on the board is represented by a `Square` object. This class encapsulates the details of a square, including its coordinates and the piece that occupies it (if any). The `Square` class provides methods for accessing and modifying the square's state, making it a crucial part of the board's structure.

Piece: The `Piece` class serves as the base class for all chess pieces. It defines common attributes and methods for all pieces, such as their symbol and movement rules. Specific piece types (e.g., `Pawn`, `Rook`, `Knight`) inherit from `Piece` and implement their own movement logic. This inheritance structure allows for easy extension and modification of piece types.

Player: The Player class handles player actions and interactions within the chess game by using a unified approach for both human and computer players. When instantiated, the class configures itself based on the provided name, distinguishing between human and various levels of computer players. The type of player influences the strategy used for making moves.

For human players, the makeMove method prompts the user for input and processes it to execute a move. It ensures the move is valid according to the current game state and the player's turn. For computer players, the class uses different algorithms tailored to each level of difficulty. These algorithms evaluate possible moves based on the game state, with more advanced levels employing more sophisticated strategies.

Move: The `Move` class represents a move in the game, including details such as the starting and ending squares, the piece being moved, and any special conditions (e.g., castling or promotion). This class centralizes move-related logic, ensuring that all moves are correctly executed and validated.

Game: The `Game` class is the central controller of the chess game, managing the overall flow and state of the game. It coordinates between the board, players, and observers, handling turn management, move validation, and checks for win conditions.

Observers: Observers were implemented to keep the game state synchronized with the visual representation of a chess board. TextObserver outputs the chessboard on the standard output, along with the current game state and turns. GraphicsObserver is responsible for rendering the current state of the game onto X11 window. Both observers are notified of changes in the board state and update their displays accordingly.

3 Design

4 Resilience to Change

5 Answers to Questions

For the most part, we believe that the solutions we proposed initially are still the optimal method of solving these complex problems, however, due to the change in our project structure, we have slightly altered our solution for the undo function to correlate more closely to the codebase we currently have.

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.**

   To implement a book of standard openings in our chess program, we would use a tree data structure to represent the opening sequences. Each node in the tree would represent a board state, with the root node representing the initial position. Each child node would represent a possible move from its parent node. We would compile a database of standard openings from trusted sources such as chess.com. This data would include move sequences and associated win/draw/loss statistics. Using this data, we would construct the tree, where each sequence of moves is represented as a path from the root node to a leaf node or an intermediate node. Each node would store the move that leads to it, the resulting board state, and statistics (win/draw/loss percentages) for that move.

   During the opening phase of the game, the game engine would refer to this opening book tree. When it is the computer's turn to move, it would look up the current board state in the tree, retrieve possible responses (child nodes), and select a move based on the associated statistics, such as preferring moves with higher win percentages.

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

   Unlike in the initially planned

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. (If it's important to your answer, state whether you're assuming free-for-all or team rules and then answer the question. You don't need to get too specific into the rule set changes in answering the question though; your focus should be more on what would need to be altered at the high level of the design?)**

   If we were to develop a four-handed chess variant, we would assume a free-for-all format, where each of the four players competes independently.

   We would need to first account for a change in the chess board itself. Typically, a four-handed chessboard is a larger, 8x8 central square board with 2x8 attached to each side for each player's pieces. We would need to adjust our board representation to reflect this larger grid, ensuring that each player's pieces are positioned correctly at the start of the game.

   Another major change would come in the form of managing the game state for two more players. This would involve needing to adjust the game loop to cycle through four players instead of two, ensuring that turns are taken in the correct sequence. The movement rules for pieces would largely remain the same, but the program would need to account for interactions between pieces from four different players. This involves updating our move validation logic to check for threats and captures from all three opponents. Additionally, we would need to ensure that special moves (e.g., castling, en passant) are correctly implemented in the context of a four-player game. We would need to redefine the victory conditions to suit a free-for-all game, and need to implement a system for tracking the player eliminations throughout the course of

the game. Instead of a simple win/loss result, the program should be able to handle multiple outcomes, such as when one player is checkmated or all players but one are eliminated.

The user interface would also require significant adjustments to display the larger board and the additional players. This includes updating the board rendering logic, adding controls for selecting and moving pieces for each player, and providing visual indicators for whose turn it is. The UI would also need to display the status of all four players, such as their remaining pieces and whether they are in check.

By making these changes, our chess program can be adapted to support a four-handed chess game, providing a new and exciting way for players to engage with the game. This transformation involves significant modifications to the board layout, player management, movement rules, victory conditions and user interface ensuring a seamless and enjoyable experience for all four participants.

6 Extra Credit Features

Although we had planned to implement several extra features, and have even built the core of the logic to enable these, due to time restrictions, we had to focus more on the functionality of the game itself, rather than improving the user experience.
Having used chess.com countless times in the past, we wanted to implement most of the same features, with the undo and redo features being at the top of our priority, since it is common for players to make mistakes while playing the game online.
We have the core structure and logic implemented, as a vector of Move objects is stored in our Board class, and the reversal of the from and to coordinates would enable this feature to be operational, however, while implementing the logic for this command, we ran into several segmentation faults that we were not able to fix in time, hence the feature had to be scrapped last minute.

1. **Displaying Captured Pieces**

   What was left of this feature

7 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?**

   Throughout this project, I learned several valuable lessons about developing large programs, whether working alone or as part of a team:

   (a) **Organization**: We've learned to effectively organize our heirarchy for effective programming. Also, we've learned that version control is crucial for developing a program as a team. We have struggled at first, as our codebase differed as we worked on different functions asynchronously. However, after using Github and the branching function, merging and comparing our code was much easier, improving our efficiency.

   (b) **Testing & Debugging**: We learned the importance of memory management, and the debugging & testing that goes with it. Thorough testing helps catch errors early and ensures that the code functions as expected. Debugging skills are essential for identifying and fixing issues, and a well-organized codebase makes this process more efficient.

(c) **Consistency and Documentation:** Maintaining consistency in coding style and documenting the code thoroughly are essential practices. They help ensure that the code is readable and understandable, whether you're working alone or with a team. Good documentation provides context and instructions, which can be invaluable when revisiting the code after some time or when handing it over to other developers.

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

   If I had the chance to start the project over, there are several key improvements I would make. Firstly, I would use version control software from the beginning. This would have allowed for better management of code changes, easier collaboration, and a more systematic approach to tracking progress and reverting to previous versions if needed. Additionally, I would invest more time in organizing the UML diagrams from the outset to ensure that the class relationships and responsibilities are clearly defined and well-structured. This would help avoid confusion and facilitate a more coherent design process. Finally, I would ensure a solid understanding of each class's role and functionality before diving into implementation. This would help in creating a more streamlined and efficient development process, minimizing potential issues and ensuring that the design aligns closely with the project's requirements.

8 Conclusion