Chess Design Document

By: Parsh, Samson and Siddharth

Introduction

In this project, we have developed a C++ implementation of a chess game that features both a graphical display in XWindows and a text display in the terminal. Our implementation leverages key object-oriented principles and design patterns covered in our CS247 class to ensure a modular, maintainable, and scalable codebase. The game supports all standard chess rules, including special moves like castling and en passant, and accommodates both human and computer players across multiple difficulty levels, as specified in the Chess project guidelines provided. This report outlines our final design, implementation details, member responsibilities, and development process.

Overview

Our project is structured using a design that separates the responsibilities of different components to ensure clarity and maintainability. The Board, Square, and Piece classes represent the state of the chessboard. The Observer, XWindow, and Text/GraphicsDisplay classes handle the display of the chess game to the users, following an Observer design pattern. The Controller and Player classes manage user interactions with the chess game. The Piece and Player classes serve as base classes for the various piece and player types (e.g., King, Queen, Knight, Robot, Human), which inherit and extend their functionality. This design effectively utilizes inheritance and polymorphism, allowing us to add special behaviors without significant changes to the implementation. We also incorporate enumeration and Move objects for additional scalability and organization.

During our brainstorming sessions, we considered using FEN (Forsyth-Edwards Notation) to translate the state of the chessboard into a string. However, we implemented our desired functionality, such as the undo feature, without using FEN notation. Instead, we maintained a detailed stack of moves to manage the game state. Throughout the development process, we

refactored many functions and classes to simplify the design and adhere to best practices, avoiding "code smell." We focused on making our code modular and easy to extend, so we can add new features in the future without having to change a lot of existing code. This way, our project stays easy to manage and can grow smoothly as we add more to it.

Updated UML

Throughout our development, we made several significant changes to improve our code's clarity, maintainability, and adherence to good design practices. One of the key changes was deciding which class should act as the subject in our observer pattern implementation. Initially, we considered having the Square class as the subject. However, as we progressed, we realized it was cleaner and made more sense for the Board class to be the subject. This change was driven by the need to reduce coupling and improve cohesion.

By making the Board class the subject, we encapsulated the entire game state within a single class, allowing for a more cohesive and modular design. The Board class maintains a 2D vector of Square pointers (“board”), which represents the chessboard. This approach allows the Board class to manage all aspects of the game state, including piece movements, checking for check and checkmate, and handling the undo functionality. This centralization simplifies the implementation and makes the code more understandable and maintainable.

Additionally, we decided not to include methods like isPinned(), getPoints(), and move() in the Piece class. Instead, we moved all movement logic and game state management to the Board class. This decision further reduced coupling by ensuring that the Piece class only handles piece-specific behavior, while the Board class manages the overall game logic. This separation of concerns improves cohesion, as each class has a well-defined responsibility.

By refactoring our code to use the observer pattern with the Board class as the subject, we ensured that changes to the game state are efficiently communicated to the display components. This design follows the principles of modularity and extensibility, making it easier to add new features or modify existing ones without disrupting the overall structure.

Design

We applied several design patterns that we learned in class to enhance the modularity and flexibility of our chess game.

Observer Pattern (Lec 18)
This pattern is implemented to update the display in response to changes in the game state. The **Board** class extends the Subject class, and the TextDisplay and GraphicsDisplay classes act as observers that get notified whenever there is a change in the game state. This ensures that the display is always in sync with the current state of the chessboard.

```
class Board : public Subject {
private:
    std::vector<std::vector<Square*>> board;
    Controller* controller;
```

Open/Closed Principle (Lec 19)
The Open/Closed Principle states that code should be open to extension but closed to modification. We adhered to this principle by designing our classes to allow easy extension without modifying existing code. For example, if we wanted to add a new piece type with special abilities, we could do so by extending the **Piece** class without altering its existing implementation. It would look something like this:

```
1  #include "piece.h"
2
3  class CustomPiece : public Piece {
4      public:
5          std::vector<Move> getValidMoves() const override {}
6  };
```

Minimized Coupling and Maximized Cohesion (Lec 18):

We minimized coupling and maximized cohesion by ensuring that each class has a single responsibility and interacts with other classes through

well-defined interfaces. This reduces dependencies between classes and makes the codebase easier to maintain and extend. For instance, the **Controller** class interacts with the **Board** and **Player** classes through clear methods, reducing direct dependencies.

```
class Controller {
    private:
        Board* board;
        Player* currentPlayer;
    public:
        void runGame(const Move& move) {
            board->movePiece(move);
            currentPlayer->makeMove(*board, from, to, promotePiece);
        }
};
```

Liskov Substitution Principle (Lec 19)

The Liskov Substitution Principle enforces that derived classes should be substitutable for their base classes. We ensured that all derived classes followed an "is-a" relationship with their base classes. For example, every specific piece type (e.g., King, Queen) is a specialized type of the **Piece** class and can be used interchangeably wherever a **Piece** is expected.

```
class Queen : public Piece {
public:
    Queen(Colour c);
```

Resilience to Change

Our design is structured to support adaptability and extensibility through the use of well-defined class hierarchies and modular components. Key classes such as **Controller**, **Player**, **Piece**, **Board**, and **Square** are designed with clear separation of concerns, making it straightforward to introduce new features or modify existing ones. For instance, the **Piece** class is an abstract base class with pure virtual methods, which allows specific piece types like **King**, **Knight**,

and **Bishop** to implement their own movement logic. This makes it easy to add new pieces, such as a hypothetical Monkey piece, by simply inheriting from the Piece class and defining the specific behavior. Similarly, the Player class is an abstract base class from which both **Human** and **Robot** players inherit. This design allows for the seamless introduction of new player types, like a hypothetical MonkeyPlayer, without altering the core game logic.

Additionally, our system employs a **Controller** class that centralizes game logic and coordinates interactions between players and the board. The Board class, which contains Square objects, and each Square can hold a Piece, encapsulates the state of the game. This hierarchical structure allows for changes in one part of the system without cascading effects. For example, updating the scoring system in **ScoreBoard** or modifying the rules for piece movement in derived classes does not affect the Controller or Board classes. Our design also leverages separate compilation, ensuring that changes in one module necessitate recompilation of only that module, rather than the entire program. This efficient compilation strategy (as we use a Makefile), makes it easy to track dependencies and only rebuild the necessary components, as taught in our class. This approach not only speeds up development but also reinforces the independence and resilience of each module, making the system robust against changes in specifications.

Answers to Questions

Q1: We could implement this by using FEN notation. After translating the current state of the chess board into a string at the start of a player's turn, the player would have the option to ask for an opening move/hint. This would require support for a respective keyword input (e.g. "Hint") in the Controller.runGame() loop. Then, we would have created an Opening class composed of a dictionary/map of string keys (FEN boards) that pair to lists of Move objects that essentially store all valid moves to be made given a current board. This list of Moves would be outputted to the current player for them to select and play.

Q2: Note that we did in fact implement the "undo" functionality so we will explain how we did it. To implement the undo functionality, we used a stack

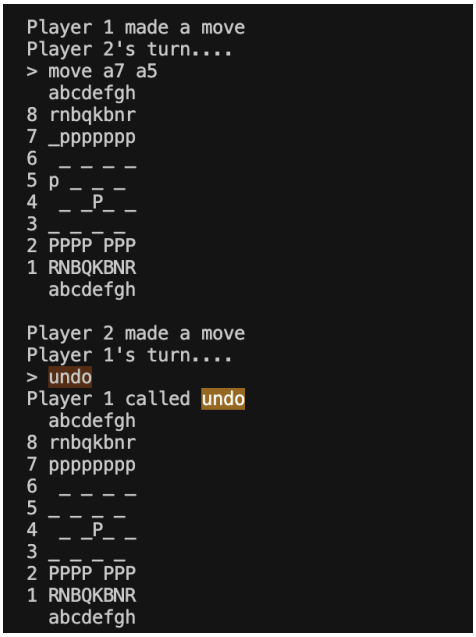
data structure to keep track of all moves made during the game. Each move is recorded as a Move object, which contains detailed information such as the piece that was moved, the original and destination squares, any captured pieces, and special move types like castling, en passant, and promotion. This stack serves as a history log, allowing us to undo moves in reverse order. The undoMove method works by popping the last move from the stack and reverting the board to its previous state based on the stored details. This includes restoring moved pieces to their original squares, reinstating captured pieces, and handling special move types appropriately. For instance, if the last move was a capture, the captured piece is placed back on the board; if it was castling, the rook is moved back to its original position. This approach supports an unlimited number of undos, as long as the stack is not empty, ensuring that players can revert to any previous state during the game. This method allows players to correct mistakes and reconsider their strategies without altering the initial state of the board, thereby enhancing the overall gameplay experience.

Q3: To support free-for-all 4–handed chess, we would change the dimensions of the Board to 14x14 Square objects and set the four 3x3 corner Squares to be empty (set each Piece attribute to nullptr) since those aren't considered part of the playable board. We would also list all those null coordinates in a constant list for Move boundary comparisons. The Controller would have to loop through 4 player turns instead of 2 and the Color enum would be updated the same. Lastly, we would have a tracker attribute in the Board class to signify the amount of King pieces still in play, since the check(mate) functionality would now loop through each standing King after every turn because multiple checks can happen. The win condition would be when only one King is left. A point system would be implemented to give pieces, promoted pieces (using a promotion flag), checks, checkmates and stalemates their respective values as per the rules. The pieces of eliminated players would be "grayed out/non-interactable" with an elimination flag. The rest of the game's functionality would remain the same, as legal moves still follow the regular rules.

Extra Credit Features

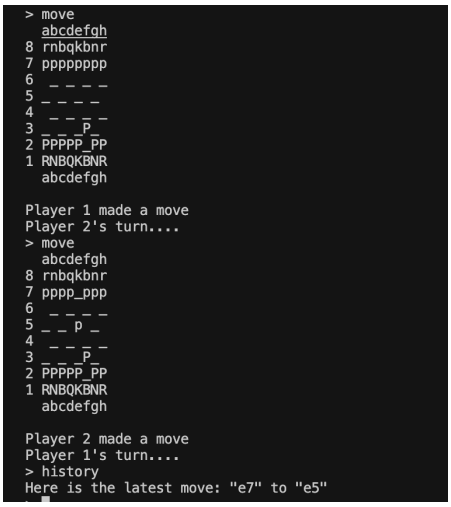
We have come up with the following extra credit features.

Undo Move: For the undo move feature, we implemented the ability for players to undo their last move. This was challenging because we had to meticulously track and revert each type of move, including captures, en passant, castling, and promotions. The undo functionality required maintaining a stack of moves (**moveStack**), where each move is recorded with all necessary details to revert the board to its previous state (piece color, type, etc.). This feature enhances gameplay by allowing players to correct mistakes and rethink their strategies, providing a more flexible and forgiving playing experience.



Get History: This feature proved incredibly useful during the development of the computer players. By typing "history," players can see the exact movements of the pieces generated by the computer without needing to visually inspect the board. This functionality supports all combinations of computer and human players. It is particularly handy for human players who

want to review the last move made. For example, in a game between two computer players, after making a move, players can type "history" to get the exact move in chess notation.



Final Questions

Q1: Working on this project taught us several valuable lessons about teamwork in software development:

Communication is KEY: We regularly met up at E7 to work together and if someone couldn't make it, we made sure to keep everyone on the same page and ensured that issues were quickly addressed. We used a Discord group chat to stay organized and coordinate our efforts effectively.

Roles and Responsibilities: Dividing tasks based on each team member's strengths allowed us to work efficiently and make progress in parallel. The design document we submitted for DD2 was incredibly helpful, providing a timeline with member responsibilities that acted as a structured calendar. We often referred to it to see what our next tasks were.

Git and GitHub: Using Git was essential for managing our codebase. It made collaboration smoother and allowed us to track changes, fix mistakes, and work on different features simultaneously. Dealing with merge conflicts was a pain, but the struggle helped us become better software engineers.

Working with each other: Pair programming improved our code quality and allowed us to learn from each other. When one person got stuck, explaining the issue to someone else often led to a solution. We did this for almost every bug, and it was a great way to troubleshoot together.

Valgrind and testing: We learned the importance of thorough testing and debugging. This was crucial, especially when we realized we were managing memory poorly and had to switch to smart pointers late in the process. This experience underscored the value of good memory management practices from the start.

Q2:
If we could start over, we would definitely use smart pointers from the beginning. We struggled a lot with memory management, spending countless hours using Valgrind to test for and fix segmentation faults and memory leaks. Using smart pointers would have handled much of the memory management for us, saving us from these issues. This change could have freed up almost two days' worth of time, which we could have dedicated to implementing additional features and conducting more thorough testing. Smart pointers would have streamlined our development process, allowing us to focus more on enhancing the game's functionality and ensuring a more robust implementation.

Conclusion

Our chess implementation project has been a challenging but rewarding experience. By following key object-oriented principles and design patterns learned from class (CS247), we created a maintainable and scalable codebase. Significant design decisions, such as using the observer pattern with the Board class as the subject, enhanced our code's clarity and functionality. Implementing extra credit features like undo move functionality also improved the game's user experience and robustness. Overall, this project deepened our understanding of software development principles and teamwork. The lessons we learned and skills we gained will be valuable in future, and our final implementation stands as a testament to good software design and collaboration.