CS246 Final Project

Due Date 2

Final Report

Joey Maillette
Sean Lee
Vishal Jayakumar

July 26th, 2024
CS246
University of Waterloo
Spring 2024

Table of Contents

Introduction

Our CS246 Spring 2024 project was inspired by the rich history and complex strategies of the game of chess. Our team embarked on a journey to create a comprehensive chess game implementation, making it possible to interact with both text-based and graphical displays. In addition, it supports both human and computer players. Thus, whether you are playing against a computer, or with other players, it creates a fun and interactive experience for everybody. In the end, it was a difficult yet rewarding exercise in problem-solving, collaboration, and innovation.

From the get-go, we faced many challenges. Designing the UML diagram required careful planning to make sure all components, including the GameManager, Chessboard, and various player and observer classes, were well-integrated. Implementing chess rules and features required a deep understanding of the game as well. The Observer pattern introduced complexities in dynamic memory management that tested our debugging skills, which was crucial for maintaining up-to-date displays and the reliability of the system. Also, we encountered specific hurdles such as handling memory leaks, ensuring all rules of chess were satisfied, and creating a visually appealing graphical interface. We iterated through numerous design changes, testing phases, and refinements, learning from each setback and emerging stronger.

Ultimately, this project was more than an academic exercise. It was a test of our ability to tackle complex object-oriented programming problems and adapt to new challenges. Most importantly, we learned to collaborate and work together, similar to a real-world work setting for software development. Through dedication and collaboration, we not only built a functional and engaging chess game but also gained valuable insights and skills that will serve us in future endeavors.

Overview

Our chess game project is designed to be both modular and efficient, using object-oriented principles to create an interactive experience of chess. The main component is the Game Manager class, which controls the flow of the game by handling player interactions and managing the game state. It sets up the game, processes each move, tracks scores, and works with the Chessboard class to execute moves and keep the game state updated. The Chessboard class is crucial for keeping track of where pieces are and making sure moves are valid. It also acts as a subject in the observer pattern, notifying the TextObserver and GraphicalObserver classes of any changes. This setup makes sure that updates to the game are shown accurately in both text and graphical formats. Additionally, the design features separate Player classes, including Human and Computer, to manage player interactions and AI behavior. Each chess piece is represented by a subclass of the Piece class, with rules and behaviors specific to their movements and interactions.

Design

Game Manager

The Game Manager class is important in controlling the flow of the chess game, organizing interactions between players and the chessboard. It makes sure that the overall game state is accurately managed. The functionality provided by this class is central to both starting a game and overseeing it. The following detailed breakdown of its functions outlines how it interacts with other components.

The startGame(player1: string, player2: string): void function initializes the game. It sets up the initial game state by taking the names or IDs of two players, preparing the chessboard, and designating the starting player. This function ensures that all components are correctly configured before gameplay begins, providing the transition from setup to active play. The nextMove(move: string): void function handles the progression of the game by processing each player's move. It validates the legality of the move, updates the game state on the Chessboard, and checks for game-ending conditions such as checkmate or stalemate. This function maintains the game's integrity, as it ensures that each move is correctly executed and that the game evolves as intended. The printSeriesScore(): void function tracks player performance over multiple games by calculating and outputting scores. This feature provides a summary of wins, losses, and draws. The setSetupMode(enable: bool): void function tells the game manager to either exit or enter setup mode. Once in setup mode the runSetupCommand(cmd: str): void allows the manager to pass valid user setup commands to the Chessboard class. This means the manager can set board configurations and prepare the board for specific situations, by setting pieces on the board directly.

The GameManager works closely with other components. For example, it interacts with the Chessboard class, which provides functions like addPiece, removePiece, executeMove, and getState. These functions allow the GameManager to change the board and update the game state. The GameManager also works with the Observer classes, such as TextObserver and GraphicalObserver, to provide real-time updates on the game state, making sure changes are reflected in the UI. By implementing these functions and coordinating with other components, the GameManager helps to create a robust and interactive chess game, providing an engaging experience for players.

Users

The Player class and its derived classes, Human and Computer, represent the users of the chess game and manage player interactions. The Computer class and the Human class (inherits from Player) include the getId(): string function, which returns the unique identifier for each player. This ID is essential for distinguishing between players and assigning moves, scores, and other actions within the game. The Computer class also inherits from Player and introduces AI-specific functionality. The setLevel(int: level): void function adjusts the computer player's difficulty, allowing the game to change the computer's skill level from level 1 to 4 (increasing difficulty). The getMove(Chessboard): string function generates a move based on the current state of the chessboard, utilizing the selected computer level's algorithms to select the best move.

Chessboard

The Chessboard class is a critical component in this application and is responsible for managing the state of a single chess game. It owns and maintains the positions of individual pieces and is tasked with determining the legality of moves on the board. Unlike individual pieces, which are responsible for their local moves, the Chessboard handles the overall game logic, including more complex sequences such as castling, en passant, and pawn promotions.

The Chessboard interacts with several other components in the system. It communicates with the GameManager to retrieve inputs and convey high-level game state changes, such as the end of a game or an invalid move. The Chessboard also manages instances of Piece and its subclasses (Queen, Rook, Bishop, Knight, Pawn, King, BlankPiece), checking the validity of their moves within the context of the entire board. Furthermore, it acts as a subject in the observer pattern, notifying observers like TextObserver and GraphicalObserver of any changes in the game state to ensure the game is rendered correctly for users.

To provide the infrastructure for setup mode, addPiece, removePiece and setColour are all public functions that enable chess games that start from a specified board arrangement. The addPiece function adds a new piece to the board based on the provided command string, and the removePiece function removes a piece from the board as specified by the command string. Both functions update the internal representation of the board to reflect the addition/removal and involve parsing the command to determine the piece type, colour, and position. The setColour function sets the colour of a specific piece based on the command string, which is crucial for handling colour-specific rules and interactions during the game. The executeMove function executes a move on the board, returning a boolean to the gameManager indicating whether the move was successful. This involves checking the move's legality, updating the positions of pieces, and handling any special rules associated with the move. The gameDone function determines if the game has concluded and returns a pair consisting of a boolean (true if the game is over) and a string (the colour of the winning side or 'draw'), which is communicated to the GameManager for further handling. Finally, the getState function retrieves the piece located at the specified row and column on the board, used by both the game logic and UI components to query the current board state.

Pieces

The Piece class is an abstract class that defines the general layout and functionality for all chess pieces. Each piece is responsible for managing only the information relevant to its own game, independent of other pieces on the board. The composition of pieces consists of a Piece superclass followed by the subclasses: King, Queen, Pawn, Bishop, Knight, and Rook. These subclasses serve as concrete pieces, and have similar implementations with slight differences to account for their unique movements and rules.

The isValidMove(moveForward: int, moveLeft: int): bool function determines if a piece's move is valid based on its specific movement rules. This function is implemented differently in each subclass to account for the distinct allowed movements of each piece. It considers both positive and negative values for moveForward and moveLeft, where negative values indicate movements in the opposite direction (e.g., right or backward), which allows for movement in all directions depending on the piece.

The getName(): string function returns the name of the piece (e.g., "King", "Queen", "Pawn"), which the Chessboard uses to uniquely identify pieces without needing to perform type checks on the actual object. The getColour(): string function returns the colour of the piece (e.g., "White", "Black"), assisting in determining the ownership of the piece and ensuring moves follow the rules regarding alternating turns.

The Rook subclass includes additional functions, setMoved(): void and hasMoved(): bool, to keep track of its eligibility for castling, a special move in chess. Similarly, the King subclass also includes setMoved(): void and hasMoved(): bool for the same reason, to determine if castling is possible. The Pawn subclass has additional functions specific to its unique moves: enPassantAllowed(): bool checks if the pawn is eligible for the en passant move. setEnPassant(value: bool): void allows for dynamically setting whether enPassant will be enabled.

These detailed functions outline how each subclass interacts with the Chessboard. This design allows for a clear separation of concerns, where each piece class handles its own logic, while the Chessboard manages the overall game state and interactions between pieces.

UI Design Architecture

Our design implements the observer pattern in order to facilitate displaying the board in multiple forms (Text and Graphical), with the subject being the Chessboard class. So whenever the board state changes (move is made) the TextObserver and GraphicalObserver are notified of the change and can query the board state at each square and display accordingly. By using the Observer Pattern we shift the responsibility of displaying the board from the Chessboard class to purpose-built Observer classes.

TextObserver

This class is responsible for displaying the up-to-date state of the game in the Text format as described in the project guidelines and shown below,

```
8 rnbqkbnr
7 pppppppp
6 _ _ _ _
5 _ _ _ _
4 _ _ _ _
3 _ _ _ _
2 PPPPPPPP
1 RNBQKBNR
```

Where capital letters denote white pieces and lowercase letters denote black pieces. This allows the user to have a visualization of the board directly in the command line.

GraphicalObserver

This class is responsible for providing a more visually-appealing visualization of the board, using the graphics functionality provided by the XWindow library. It displays the checkered board with black and white game elements overlayed on it. The graphical window displays the board state similar to the below image:

Changes to Initial Design

From a high-level perspective, our initial design remained largely intact throughout the development process. The design patterns we selected and the clear separation of components turned out to work exceptionally well.

The primary change we encountered was an increase in the number of public methods within some classes. This adjustment was reflected in the difference between our initial and final UML diagrams. While planning, it is challenging to fully anticipate all the methods that various parts of the application will require and the data each component will need throughout the workflow. As we progressed through development, we identified additional functionalities that needed to be exposed through public methods to facilitate smooth interactions between components.

Another minor change was getting rid of the BlankPiece class and instead representing empty squares as null pointers. This change was made for simplicity since we realized we did not need a whole additional class just to represent the concept of an empty square.

Overall, the adjustments made were minor and did not deviate from our initial architecture. The increase in public methods was a natural evolution as we gained a deeper understanding of the application's requirements and interactions during implementation.

Resilience to Change

Our design is structured to be highly scalable, generalizable, and adaptable, ensuring that the code can handle changes in program specifications efficiently. This resilience is achieved through several key design choices and principles:

Custom Exceptions

We implemented custom exceptions that provide a strong guarantee for error handling and flow control across the entire application. These exceptions maintain class invariants at all times, ensuring that the program remains in a consistent state even when errors occur. By encapsulating error conditions in well-defined exception classes (Invalid user input and internal error), we make it easier to manage and recover from errors, which enhances the robustness and flexibility of the system.

GameManager as a Template

The GameManager class was designed as a template that can accommodate various types of games, whether single-player, multiplayer, or computer-players. While it incorporates slight specific knowledge about chess, its underlying infrastructure is highly adaptable and generalizable. The GameManager handles user interactions, game state management, and coordination with other components in a way that can be extended to support different games with minimal modifications.

Observer Pattern for UI

We utilized the observer pattern to implement the user interface, separating the responsibilities of displaying the game state from the game logic itself, which was key in keeping the application robust and adaptable. This design allows us to support multiple forms of UI (text-based and graphical) without coupling the UI components tightly to the game logic or to each other. Although the specific functions to render the UI might vary depending on the game, the core infrastructure and communication mechanisms remain consistent and adaptable. This approach makes it easy to add new types of observers or modify existing ones to enhance the user experience.

Use of Smart Pointers

Smart pointers were employed right at the beginning of development throughout the codebase to manage dynamic memory automatically, reducing the risk of memory leaks and dangling pointers. This choice simplifies memory management and ensures that memory is released appropriately, even as new features and logic are added. By leveraging smart pointers, we avoided a lot of memory debugging time since, when building or altering logic, we didn't have to worry about dangling pointers or forgetting deletes. Thus, this design choice made the codebase more maintainable, easier to extend, and saved developer time.

Answers to Questions

Question 1

In chess, the opening phase is especially important as it sets the stage for the rest of the game. The first dozen or so moves establish the player's control over the center of the board, prepare for piece development, and improve the security and safety of the king. The quality of these opening moves can greatly influence the outcome of the game. Having a well-researched opening book helps players follow good strategies and avoid common pitfalls.

To integrate a book of standard openings into a chess program, it would be reasonable to use a tree data structure to manage the opening moves. This structure would allow the program to store and retrieve move sequences based on the current game state. With this setup, the program can quickly check if the current board position matches any known opening and suggest the next move accordingly. Then, the opening book can be populated with common chess openings, by either manually entering well-known openings or integrating data from reputable sources, like chess.com. A database might be needed to store and search through the data efficiently for bigger collections of openings. Integration with the game engine would involve implementing a move lookup feature. This would check if the current board position is part of an opening sequence of an existing opening in the book. If a match is found, the program can suggest the next move from the opening book. However, the engine should fall back to its default algorithm methods if no match is found. Allowing user customization and updates is also important and would be a great update to the program. Users being able to add their preferred openings and variations to the book would make the overall experience better for the players. Additionally, providing a user-friendly interface to view and learn about different openings can enhance the overall chess experience. Offering move suggestions and an opening explorer tool will help players make informed decisions and improve their game.

A book of standard openings can help enhance a chess program's values and usability. The program can help players get through the early game and improve overall chess gameplay for the players by providing players with access to proven opening strategies. This feature not only supports strategic learning but also adapts to various play styles. This overall makes the chess program a more powerful tool for players at any level.

Question 2

To enable the undo functionality within the game of chess, the system must maintain a history of moves and board states. The most straight-forward implementation for this would be to record each move and its resulting board state, storing the results in a stack data structure. When a player makes a move, the current board state and the move itself are pushed onto the stack. If the player chooses to undo the last move, the system will pop the top entry from the stack and revert the board to the state stored in that entry thereby leaving the chessboard in the exact state it was in one move ago.

For implementing unlimited undos, the general approach is consistent with a single-undo functionality, except there are a few additional considerations. Firstly, the stack data structure should be capable of growing dynamically and to a large size in order to accommodate an unlimited number of moves. Additionally, specific properties managed outside the chessboard, such as special move eligibility (e.g., castling, promotions, and en passant), must also be tracked by maintaining references to previous piece states. Lastly, the user interface should also be designed to handle a potentially large number of undos, providing clear options and feedback to the player about the state of undo operations.

By using a stack to track these elements and ensuring that the system can handle unlimited undos, players can enjoy a more flexible and forgiving gameplay experience. Proper attention to memory management and user interface design further ensures that this feature enhances rather than complicates the game.

Question 3

We assume the standard free-for-all four-player chess rules. Since our design strives to maintain high levels of cohesion with low levels of coupling, we would only need to change some of the classes in order to make the program a four-handed chess game.

GameManager Class: The GameManager class has already been designed to be decoupled from the Chessboard class (which contains the specific game rules), so its single responsibility is to manage the game at a high level. The only high level difference between Chess and four-player Chess, is the addition of 2 additional players. So the only change that would have to be made to the GameManager class is to keep track of 4 Player objects, instead of just 2.

Computer Class: The getMove method of the computer class would have to be modified to use algorithms that can compute moves for four-player chess instead of standard two player chess.

Chessboard Class: This is where a majority of the modifications would have to be made, since this class encapsulates the game state. While a standard chess board class is a 8x8 grid (so a total of 64 squares), standard four-handed chess is played on a larger 14x14 grid with 3x3 squares cut out at the 4 corners (so a total of 160 squares). In our design we store a piece for each square, so we would now need the chess class to store an array of 160 Piece objects. Four-player chess is based on points (instead of simply ending on checkmate), so we would need to add member variables and methods to keep track of and modify the scores of players as points-affecting game events (checkmates, statements etc.) occur.

TextObserver and GraphicsObserver Classes: The only modification that would be required for these 2 classes is to display the expanded 160 square board instead of the standard one.

Extra Credit Features

Extra Feature #1

One extra feature that we implemented is the ability for the user to choose between multiple styles for the graphical displays. We have implemented 3 different colour schemes for the board and allow the user to select the colour scheme through the command interpreter. This entailed initializing the Graphics Observer class with the style selected by the user.. Adding this feature has made the user experience more customizable and appealing.

Extra Feature #2

Another extra feature we would have implemented but could not due to time constraints is the addition of a chess game clock. This would entail allowing the user to set up a timer for each of the Human players (obviously doesn't apply to Computer players), and having the timer decrement while waiting for the human player to input their move. This feature would give a more competitive feel to the game, while also making it a useful tool to practice for chess tournaments (where chess clocks are standard).

Final Questions

Question 1

This project taught us several important lessons about developing software in teams. First, communication was key to everything. We scheduled regular check-ins and discussed our progress, in order to stay aligned and avoid overlapping work. Also, we learned the importance of dividing tasks based on each team member's strengths, which made the process more efficient. Coordinating our efforts on a shared codebase (on Github) required careful attention to version control to prevent conflicts and to make sure that everyone was working with the latest updates.

Question 2

If we had the chance to start over, we would focus more on planning and setting more detailed deadlines from the beginning. In hindsight, we realized that having a more thorough plan and timeline would have helped us manage our tasks better and avoid last-minute errors in our code. We would also invest more time in the initial design phase to ensure that our UML was as modular and flexible as possible, which could have made it easier for us to implement. Another improvement would be to have more frequent code reviews by each other throughout the development process. This approach would have helped us catch bugs earlier and thus, we would have had a smoother and faster process overall.