UNIVERSITY OF WATERLOO

CS 247: Software Engineering Principles

Spring 2024

Project Final Report

Chess

Team Member Full Name	UserName and Student Number
Jenny Hui	j2hui - 20988854
Anna Li	a5li - 20990179
Jason Liu	yj7liu - 21044181

1 Introduction

This report includes various sections such as structure, design decisions, implementation details, and additional features of the chess game project for CS247.

The purpose of this project is to develop a comprehensive chess game that includes standard functionalities such as legal moves for all chess pieces and players taking turns, user interactions through text and graphics display, and advanced features including multiple computer player levels and special moves. This project was designed using object-oriented programming principles to ensure a well-structured and change-resilient codebase.

2 Overview

The chess project is structured around the following key components:

Game Management (Game Class): Handles the overall game flow, including starting and initializing the board, prompting steps, human/computer player taking turns, game states (such as checkmate, resign, stalemate), and scoring.

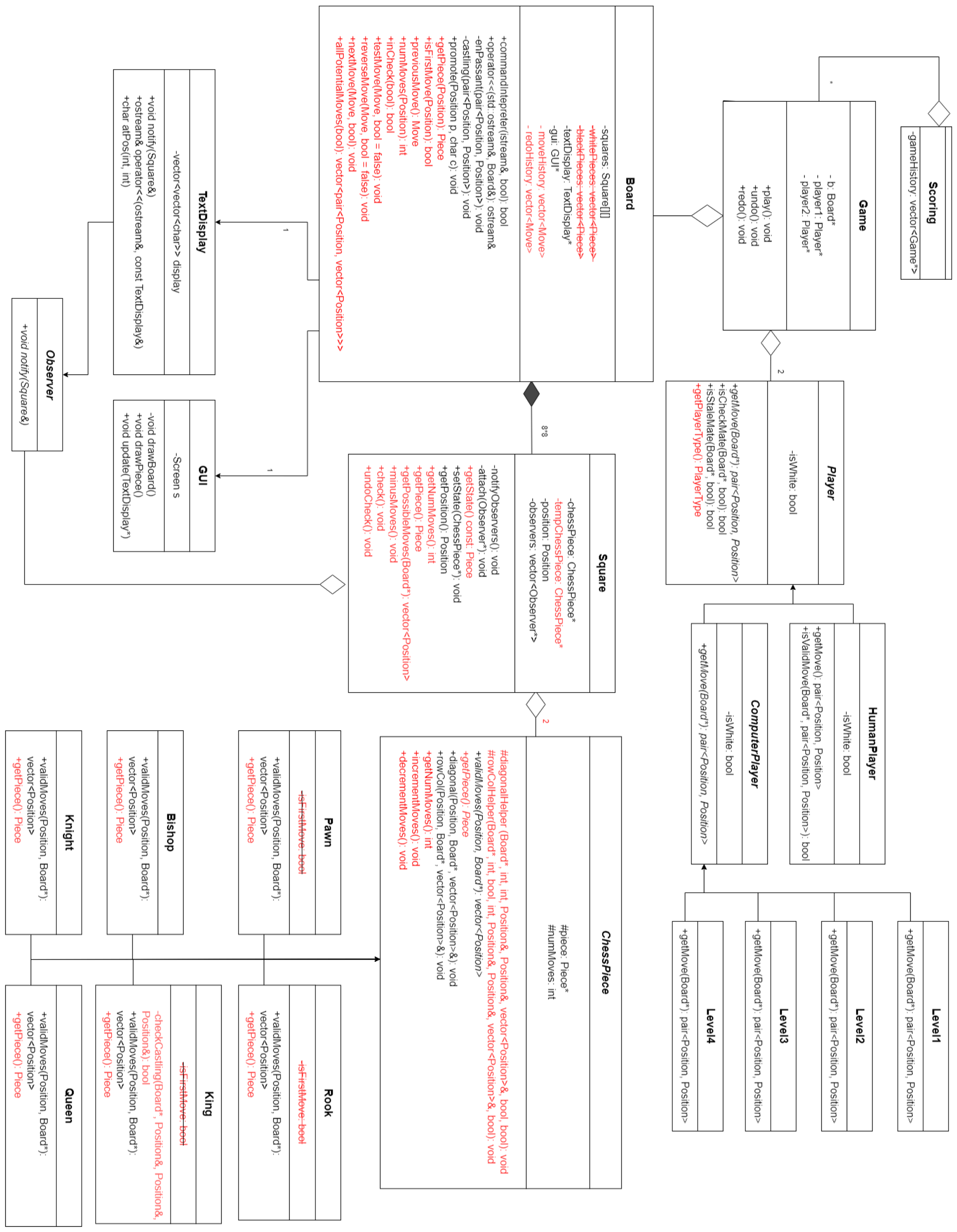
Board Representation (Board Class): Manages the 8*8 checker grid chessboard layout, move history, interpreting commands (move, add, remove), making moves on the chessboard, piece placement with special rules (En passant, castling, pawn promotion), and move validation.

Player Interaction (Player Class): Defines player types (human or computer) and their interactions with the game, including checking stalemate and checkmate.

Chess Pieces (ChessPiece Class): All types of chess pieces inherit from ChessPiece, which each implements the specific behavior and movement rules for each type of chess piece.

Display and UI: Manages the graphical and text-based display of the game state by printing the chessboard after any move or changes.

3 Updated UML


Scoring
-gameHistory: vector<Game*>
*
Game
- b: Board*
- player1: Player*
- player2: Player*
+play(): void
+undo(): void
+redo(): void
2
Player
-isWhite: bool
+getMove(Board*): pair<Position, Position>
+isCheckMate(Board*, bool): bool
+isStaleMate(Board*, bool): bool
+getPlayerType(): PlayerType
HumanPlayer
-isWhite: bool
+getMove(): pair<Position, Position>
+isValidMove(Board*, pair<Position, Position>): bool
ComputerPlayer
-isWhite: bool
+getMove(Board*): pair<Position, Position>
Level1
+getMove(Board*): pair<Position, Position>
Level2
+getMove(Board*): pair<Position, Position>
Level3
+getMove(Board*): pair<Position, Position>
Level4
+getMove(Board*): pair<Position, Position>
Board
-squares: Square[][]
~~-whitePieces: vector<Piece>~~
~~-blackPieces: vector<Piece>~~
-textDisplay: TextDisplay*
-gui: GUI*
- moveHistory: vector<Move>
- redoHistory: vector<Move>
+commandinteperter(istream&, bool): bool
+operator<<(std::ostream&, Board&): ostream&
-enPassant(pair<Position, Position>): void
-castling(pair<Position, Position>): void
+promote(Position p, char c): void
+getPiece(Position): Piece
+isFirstMove(Position): bool
+previousMove(): Move
+numMoves(Position): int
+inCheck(bool): bool
+testMove(Move, bool = false): void
+reverseMove(Move, bool = false): void
+nextMove(Move, bool): void
+allPotentialMoves(bool): vector<pair<Position, vector<Position>>>
1
1
8*8
TextDisplay
-vector<vector<char>> display
+void notify(Square&)
+ostream& operator<<(ostream&, const TextDisplay&)
+char atPos(int, int)
GUI
-Screen s
-void drawBoard()
+void drawPiece()
+void update(TextDisplay*)
Observer
+void notify(Square&)
Square
-chessPiece: ChessPiece*
-tempChessPiece: ChessPiece*
-position: Position
-observers: vector<Observer*>
-notifyObservers(): void
-attach(Observer*): void
+getState() const: Piece
+setState(ChessPiece*): void
+getPosition(): Position
+getNumMoves(): int
+getPiece(): Piece
+getPossibleMoves(Board*): vector<Position>
+minusMoves(): void
+check(): void
+undoCheck(): void
2
ChessPiece
#piece: Piece*
#numMoves: int
#diagonalHelper (Board*, int, int, Position&, Position&, vector<Position>&, bool, bool): void
#rowColHelper(Board*, int, bool, int, Position&, Position&, vector<Position>&, bool): void
+validMoves(Position, Board*): vector<Position>
+getPiece(): Piece
+diagonal(Position, Board*, vector<Position>&): void
+rowCol(Position, Board*, vector<Position>&): void
+getNumMoves(): int
+incrementMoves(): void
+decrementMoves(): void
Pawn
~~-isFirstMove: bool~~
+validMoves(Position, Board*): vector<Position>
+getPiece(): Piece
Bishop
+validMoves(Position, Board*): vector<Position>
+getPiece(): Piece
Knight
+validMoves(Position, Board*): vector<Position>
+getPiece(): Piece
Rook
~~-isFirstMove: bool~~
+validMoves(Position, Board*): vector<Position>
+getPiece(): Piece
King
~~-isFirstMove: bool~~
-checkCastling(Board*, Position&, Position&, Position&): bool
+validMoves(Position, Board*): vector<Position>
+getPiece(): Piece
Queen
+validMoves(Position, Board*): vector<Position>
+getPiece(): Piece

4 Design

Separate Compilation: In our chess project, separate compilation is applied by dividing the code into multiple header (.h) and implementation (.cc) files. This separation helps in organizing the codebase logically and functionally.

Observer Pattern: Used in TextDisplay and Observer Class with methods like attach(), notify(), and update(). The Square class implements the observer pattern to notify the TextDisplay and GUI classes of changes. This ensures that the display is always up-to-date with the current game.

Template/Strategy Pattern: Used in the Player class hierarchy. The Player class defines a common interface for getMove methods, and different strategies are implemented in HumanPlayer and ComputerPlayer subclasses. This allows the game to dynamically choose between different move generation strategies. Also used in ComputerPlayer abstract class with concrete subclasses Level1, Level2, Level3, and Level4 representing different difficulty levels. Each of the levels has its own getMove(Board*) method to add new strategies easily.

Inheritance & Polymorphism, Dynamic Dispatch: Used in ChessPiece abstract class with concrete subclasses for all piece types (Pawn, Bishop, Knight, Rook, King, Queen). Virtual methods validMoves for move validation and getPiece use dynamic types to return the correct values for each respective type of chess pieces. This enables dynamic dispatch, where the method implementation is determined at runtime based on the actual type of the object. Polymorphism allows the program to handle different types of chess pieces uniformly through pointers or references to the ChessPiece base class.

Composition & Aggregation: Composition is used in the Board class composed of multiple Square objects. The Board class manages these components, allowing for complex structures to be treated uniformly. Each Square also contains two ChessPiece (tempChessPiece and chessPiece) which is aggregation because Square does not manage memory of the chess piece.

Encapsulation: Our design heavily relies on encapsulation, where data is hidden within classes and only accessible through public methods. This is evident in classes like ChessPiece, Player, and Board, where private members are accessed via public getter methods.

Single Responsibility Principle (SRP) and Centralization of Logic: Special moves like castling and en passant are handled within the Board class. This centralizes the logic for these moves and ensures they are validated consistently. Castling, pawn promotion, and en passant both have their own designated methods, adhering to SRP.

The following two sections discuss how the above mentioned patterns and principles help us maximize cohesion and minimize coupling.

Maximizing Cohesion: As mentioned earlier in the Observer Pattern section, cohesion is maximized by ensuring each class has a single, clear responsibility. The Square class manages chess pieces and notifies observers of changes, while the TextDisplay and GUI classes handle the display of the game state. This clear separation ensures high cohesion within each class. Also, the Template Pattern in the Player class ensures specific strategies are implemented within each subclass, keeping strategy code cohesive. Furthermore, the SRP and centralization of logic enhance cohesion by assigning the Board class the responsibility of managing the game board and validating moves, including special moves. Methods like castling() and enPassant() clearly define responsibilities.

Minimizing Coupling: As discussed above, coupling is minimized through several design patterns and principles. The Observer Pattern decouples the Square class from the display classes (TextDisplay and GUI) allowing changes in the display logic without affecting the Square class. The Template/Strategy Pattern in the Player class encapsulates different move strategies within their own classes to reduce dependency. The Template/Strategy Pattern in the ComputerPlayer class enables dynamic strategy selection, with each level implementing the getMove(Board*) method allows new strategies to be added without affecting existing code. Inheritance and polymorphism in the ChessPiece class provide a common interface for all chess pieces, allowing the Board and other classes to interact with chess pieces without needing to know their specific types. The SRP and centralization of logic isolate special move logic within the Board class which reduces dependencies between classes. Composition and aggregation in the Board class uses ChessPiece without owning its lifecycle to reduce coupling. Encapsulation hides data within classes and provides public methods to minimize dependencies.

5 Resilience to Change

The design of this chess project is resilient to various changes due to its structured approach. Here are specific ways in which the design supports flexibility and adaptability, allowing for modifications with minimal impact on the overall system:

Rule Changes: One of the key aspects of resilience to change is the ability to modify the rules of the game without impacting other parts of the system. Each chess piece's movement logic is encapsulated within its respective class, such as Pawn, Bishop, Knight, Rook, Queen, and King. This encapsulation ensures that if the rules for a specific piece need to change, only that piece's class needs to be updated. For instance, if the movement rules for the pawn were to change, only the Pawn class would need to be modified. The rest of the system, including other piece classes and game logic, would remain unaffected. This modular approach makes the codebase highly adaptable to changes in game rules, ensuring that modifications can be made quickly and efficiently without risking unintended side effects.

Feature Extensions: The use of abstract base classes such as Player and ChessPiece allow the addition of new features or player types. For example, adding a new difficulty level involves creating a new subclass of ComputerPlayer and implementing the move generation logic specific to that difficulty level. Specifically, we could create a new class, say Level5, that extends the ComputerPlayer class. Within Level5, we will implement the getMove method to define how this new level decides its moves. Similarly, if a new type of player, such as a remote player interacting over a network needs to be added, we support this by extending the Player class. We could create a new subclass such as RemotePlayer and implement the getMove method to handle move inputs from a remote source. Furthermore, for adding a new piece with unique movement rules, we could extend the ChessPiece class. Suppose we want to add a piece called Giraffe with new movement rules. We could create a Giraffe class that inherits from ChessPiece and implement the validMoves method to define the Giraffe's movements. This allows the new piece into the game, with the Board and other game mechanics remaining unchanged.

New Syntax for Input: Supporting new syntax for input can be managed by extending the existing input handling mechanisms. Right now we only support moves in algebraic notation, adding support for long algebraic or descriptive notation can be done by updating the Board class's commandInterpreter method - it handles input commands such as adding or removing pieces and setting up the game. If we wanted to support descriptive, advanced, or long algebraic notation, we would add logic to parse and interpret these commands in the commandInterpreter.

UI Changes: The separation of game logic and display logic ensures that changes to the user interface do not impact the game mechanics. The TextDisplay and GUI classes handle the presentation of the game state, while the game logic resides in classes such as Game, Board, and Player. This separation allows the user interface to be modified or extended independently of the game logic. If we want multiple text displayers, we can just create TextDisplayEnhanced that implement the abstract class Observers and output respective symbols.

Scalability: The modular design of the chess project facilitates scalability which allows the implementation of new game modes or customized rules with minimal changes. For instance, implementing a timed game mode can be achieved by extending the existing Game class and adding new methods to handle the timing logic. We could create a TimedGame class that inherits from Game and add attributes for time limits and methods to start, stop, and check the timer. This new class would override the play() method to include time management. Similarly, custom board sizes can be changed by extending the Board class and adding methods to handle the new dimensions. For example, we could create a CustomBoard class that initializes the board with different dimensions and overrides methods to validate positions and manage pieces according to the new size.

Other potential features, such as a new scoring system could extend the Game class or create a ScoreManager class to handle different scoring rules. Alternate win conditions can also be done by modifying the game flow in the Game Class accordingly.

6 Answers to Questions

Question 1: Discuss how you would implement a book of standard openings if required.

Answer: To implement a book of standard opening moves, we can create a data structure to store all the possible openings (about 120). Using a tree structure can be effective because we can check moves by following children, which saves space. Each node would represent a move, including the starting square, ending square, and piece type (e.g., e4, e5, Pawn). It would point to child nodes representing the next possible moves. We need to add a private field prevMove and a public method getPrevMove() in the Game class. At the start of the game, the current pointer will point to the root of this tree-like data structure. For each move made by the human player, the Game class will store it as the prevMove. When the computer player makes the next move, it would call getPrevMove() and compare this returned move with the children of the node the current pointer is pointing to. If the move matches a child, the pointer moves to the matched child. The move in the matched child is returned in the getMove() method of the computer player, and prevMove is updated. If no match is found or the only child is null, the program exits the opening phase and follows the computer's strategy.

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?

Answer: To implement an undo feature in the chess game, we would need to store all the moves in a stack. Or, at least a data structure that has the first in last out properties. Each time a player makes a move, this move will be stored on the stack along with some information about the board (such as player's turn, eligibility of en passant or castling, etc.) to help the undo process. When a player tries to undo, the program will pop the most recently added move from the stack (history field in class Game) and reset the game to the state before this move. In this case, the use of a stack will allow for unlimited undos from the start of the game, since every move that was made during the game is stored in this stack, allowing us to restore every previous move just by popping the stack multiple times. Additionally, if we want to offer the possibility of the "redo" feature, the popped move from stack (history field in class Game) can be stored in a queue data structure (redoHistory field in class Game), which is first in first out, to allow the most recent undo move to be restored.

Question 3: Outline the changes that would be necessary to make your program into a four-handed chess game.

Answer: To create a four-player chess variation, several modifications are needed, including changes to the board, player positions, winning conditions, point system, and game mechanics.

Board Size: The board must be larger to accommodate additional players. One version of four-player chess uses a board with three rows of eight squares added on each side, creating a board with 160 squares in a “+” shape. We can modify the squares field in Board to be a 14x14 grid of Square type, with each 3x3 corner marked as invalid. The validMoves(Position, Board*) methods in each chess piece class need to be updated accordingly.

Displaying Player Positions: We need four different colors to represent each player in the graphics version. In the text version, the grid squares should be modified to take two characters, such as P1 for a Pawn belonging to player 1.

Winning Conditions: The game ends when only one player remains not checkmated, with the goal to accumulate the most points. We need to add an integer array field in the Game class called points to track each player’s points. The Scoring class needs updating to increment the score of the player with the highest points in the previous game.

Pieces' Point System: Points are earned by capturing opponents’ pieces, with each piece type with a different number of points. We need to add a getPoint() abstract method in ChessPiece and a concrete getPoint() method in each subclass (Pawn, King, Queen, Bishop, etc.) to return their respective points when captured. Pawn promotions to queens when reaching the eighth rank also need adjustment, with the promoted queen earning only one point when captured, requiring a private field in the Queen class called isPromoted() to ensure getPoint() returns the correct value.

Game Mechanics: Actions like checkmate, stalemate, and resignation also earn points, so these methods need updating. Bonus points for checking multiple kings require the checkMate() method to check all possible positions and award points accordingly. The game mechanics need modification to accommodate these changes, including updating turn-taking rules, adjusting check and checkmate detection logic. The display, both text and graphical, needs updating to reflect this new variation and ensure clarity and user-friendliness for four players.

7 Extra Credit Features

Memory Management: The use of smart pointers (shared_ptr, unique_ptr) and vectors ensures efficient memory management. This approach eliminates the need for explicit delete statements and reduces the risk of memory leaks. The program handles all memory management automatically, ensuring safe and efficient use of resources.

Pretty UI: We included images of each chess piece on the board to enhance the user interface. This was challenging due to performance issues, so we explored using Unicode characters for chess pieces. Unfortunately, Unicode characters are only available in a few fonts, and the student LINUX server does not support those, which limited our options.

Command Line Graphics Choice: To provide flexibility during demonstrations, we added a command-line option to toggle the display with or without pictures. This feature was hard to implement because we couldn't recompile the program during the demo. However, it allows users to enable or disable the visual elements dynamically, improving usability and performance.

Fake En Passant Capture: The edge case where pieces are in the exact same position as an en passant but instead, there is another piece to capture at the destination square. In this case, we need to detect a fake en passant situation, and only capture the piece on destination without erasing the pawn next to the one making the move. This was difficult to find and implement. We solved this by adding to Board Class a vector moveHistory which has move and piece type information, and also checking whether the previous move is by pawn moving forward 2.

Displaying Computer Moves: After we tested the human vs. computer feature with someone who actually plays chess, they said it was very hard to see what the computer player moved based on our text and graphics display. To make the game more user-friendly, the computer's previous moves are now displayed in algebraic notation, such as "Black E7 TO E5". This replaces the less informative "Black computer is making a move" message. This enhancement makes it easier for players to follow and understand the computer's actions.

Two Kings Left is a Draw: This situation happens quite a lot more than expected when we play two computers against each other - even when they are different levels. To end the game properly, we implemented a rule that declares a draw when only two kings are left on the board. This rule reflects the reality of chess where a game with only kings remaining cannot be won. By including this rule, we ensured the game adheres to standard chess endgame scenarios.

Current Total Move Count: The game now tracks and displays the total move count. This feature helps players keep track of the game's progress and is essential for certain game strategies and rules. It also provides useful feedback for players to gauge the duration and complexity of the game.

Level 5 Computer Difficulty: We introduced a Level 5 difficulty setting to challenge more experienced players. This level requires advanced algorithms and strategies to provide a competitive experience. By including multiple difficulty levels, we cater to players of various skill levels, enhancing the game's replayability.

Timer Implementation: A timer feature has been added to the game, allowing players to track the duration of their moves and the overall game. This feature adds a layer of challenge and realism, as time management is a crucial aspect of competitive chess. The timer ensures players remain engaged and can practice their speed and decision-making under pressure.

8 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?

Answer: Developing this project taught valuable lessons in planning, team collaboration, object-oriented design patterns, thorough testing, and the importance of clear communication and version control. We learned that developing software involves a lot of planning before we start actually coding, such as creating UML diagrams, planning for deadlines, and distributing work among group members. Communication is critical because if some part of the code changes and affects other parts that others are working on, it’s crucial to let them know the updates. Regular updates and use of communication tools helped us stay informed and quickly address any dependency issues. Integration testing showed us that instead of dividing and conquering, it’s better to work out a very simple version of the program together and add new features on top. This approach identified and resolved integration issues early, creating a stable foundation for further development. Developing test cases for individual functionality is important on top of integration testing because it narrows down what’s not working as intended. Unit tests ensured each component worked correctly in isolation, which makes identifying and solving bugs more straightforward.

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

Answer: If given the chance to start over, a more detailed initial design phase could have saved time by identifying potential issues early. Spending more time on UML diagrams would have been particularly beneficial, as a solid design from the beginning can prevent many problems later in the project. Additionally, implementing automated testing from the start would have helped the debugging process and made identifying issues quicker and easier. Developing a comprehensive test suite at the beginning, rather than while coding, would have ensured that all functionalities were thoroughly tested, making the debugging process much faster and more efficient. Better documentation and code comments would also have facilitated easier collaboration and maintenance. Another improvement would have been spending more time researching and integrating advanced libraries. Due to time constraints, we couldn't explore and utilize more libraries that could have enhanced the chess game. Given more time, learning about and incorporating these advanced libraries would have significantly improved the game's functionality and performance. By adopting these strategies, we could have created a more robust, maintainable, and feature-rich chess game, optimizing both development efficiency and final product quality.