Design Documentation

by Haibo Sun and Fred Sun

1, Introduction:

The group design document will serve to give the reader a high-level overview of our Chess project and answer the questions in the project guidelines. The most important parts are Game, Board, various types of ChessPiece (Pawn, Knight, Rook, Bishop, Queen, King), Player, Tile, and display classes (Text, Graphics). It strictly adheres to object-oriented principles for modularity and ease in maintenance.

2, Overview:

Game:
This is the class where all the general states for the chess game application will reside. It will mainly contain references to the Board, Player objects, and status of the current game. The Game class should call Board and Player constructors which initializes the state of the board and pieces and all the necessary objects to play the chess game. This class also controls the player's turn procedure, where the moveCommand method processes a certain move in the turn, following which the state of the game is updated. Further in the Game class are methods to ascertain the condition of the game; the isValid, isValidMove, and isKingInCheck methods report if a player makes a reasonable move and determines if a king is in check respectively. These methods ensure that the rules of the game are kept intact and further ensure a clear method of endgame detection.

Board:
The Board class represents a chessboard, which is an 8x8 grid composed of Tile instances. It manages the positioning and movement of chess pieces on the board. The Board class includes methods for placing pieces, removing pieces, and moving pieces, ensuring that all moves adhere to chess rules and that the board's state is updated accordingly. The Board class interacts closely with Game class for move validation and board state updates, ensuring a seamless integration between the board and the pieces.

Tile:
The Tile class represents an individual square on the chessboard. It has attributes for occupancy and color, providing a clear representation of the board's state. The Tile class includes methods to check if it is occupied (point p is nullptr or not) and to set or get the piece that occupies the tile. These methods facilitate effective communication between the Board and Piece classes, ensuring efficient management of pieces placed on or moved across the board.

Piece:
The Piece class is an abstract base class for all chess pieces: Pawn, Knight, Rook, Bishop, Queen, and King. It defines common attributes such as position and color, providing a consistent representation for all chess pieces. The ChessPiece class declares virtual methods

for move generation (fetchAllmoves()), which are implemented by its subclasses. This abstraction allows for flexibility and extensibility, enabling the easy addition of new piece types by extending the Piece class and implementing the required methods.

Pawn, Knight, Rook, Bishop, Queen, King:
These classes extend the Piece base class and provide specific implementations for move validation and generation through the fetchAllmoves() and getHasMoved() methods. Each subclass represents a unique type of chess piece with distinct movement rules. For example, the Pawn class includes logic for pawn-specific movements such as forward movement and diagonal captures, while the Knight class implements the L-shaped movement unique to knights. This specialization ensures that each piece type behaves according to chess rules, maintaining the integrity of the game.

Player:
The Player class represents a player in the game, which can be either human or computer-controlled. It includes attributes for player type (HUMAN or LEVEL 1, 2, 3, 4, 5) and color. The Player class handles player-specific actions, such as making a move using the move method. It abstracts player behavior, allowing for the addition of different player types and strategies without altering the core game logic.

Observer(Observer Pattern):
The Observer components are implemented using the Observer Pattern, ensuring that any updates to the game state automatically notify all display classes. The Observer class defines the interface for objects that need to be notified of game state changes. Text and Graphics inherit from Observer and provide implementations for displaying the game state in text and graphical formats, respectively. In this pattern, Game is the subject, and observers are notified of each state change. This pattern ensures that the game state is consistently reflected in the display components, maintaining synchronization between the game logic and the user interface.

3, Updated UML

4, Design Principles:

Observer Pattern:
The classes Observer, Text, and Graphics implement the observer pattern. This pattern ensures that any changes in the game state, as defined inGame, are automatically updated among the observer components to keep them current. Game maintains a list of observers, and any change in the game situation is notified to these observers. This separation between game logic and rendering logic enhances modularity and simplifies the addition of new display types.

Single Responsibility Principle:
The Single Responsibility Principle is strictly adhered to, with each class in the design having only one responsibility. Game manages the game stateBoard handles the board and the placement of pieces on it, Piece and its subclasses manage piece behavior, and Observer classes manage visualization of the game. This separation of concerns makes the code easy to maintain, read, and write.

5, Resilience to Change:

The chess game application design is modularized, making it resilient to change. Updating or replacing one component does not affect other parts of the system. For example, adding new piece types or changing rules for moves requires changes only in the specific classes (subclasses of Piece) but not throughout the entire design. The implementation of the Observer Pattern allows for the addition of new display types with minor modifications to existing code, increasing the system's flexibility.

High Cohesion:
High cohesion ensures that each class or module within the project is responsible for a single, well-defined task or functionality. This clear definition of responsibilities simplifies making changes or extending features.

Piece and its Subclasses:
Each subclass of Piece (King, Queen, Pawn, Knight, Rook, Bishop) is responsible for describing the rules and movements of a single chess piece. For example, the Pawn class handles pawn-specific movements such as forward movement and diagonal captures, while the Knight class implements the unique L-shaped movement pattern. This specialization keeps all piece-specific logic encapsulated within the corresponding subclass, ensuring clear and consistent behavior. This high cohesion facilitates maintaining and extending the code base, as changes in piece behavior are isolated to the relevant subclass.

Player Classes:
The Player class represents the properties and functionalities of players in the game, maintaining attributes such as player type (HUMAN or LEVEL 1,2,3,4,5). The Player class encapsulates all actions a player can perform, such as making a move, without dealing with the details of move validation or execution on the board as these would be handled inside the

Game class. This keeps cohesion within the Player module, allowing for better organization and maintenance of player-related functions. The subclasses Human and Levels describe how moves are determined according to player type, increasing cohesion within the Player module.

Observer Classes (Text, Graphics):
The Observer classes, Text and Graphics, focus solely on UI-related functionality. Their task is to display the game state in textual or graphical form, respectively, without dealing with chess rules or game state management. These responsibilities are delegated to Game, which uses the Observer Pattern to provide the necessary information. This high cohesion ensures that UI modifications or the introduction of new display modes are isolated from the core game logic, making the display classes easy to handle and extend.

Low Coupling:
Low coupling ensures that classes interact via well-defined interfaces, minimizing their dependencies on each other. This design principle promotes a flexible and maintainable system, where changes in one part of the system have minimal impact on other parts.

Game and Board:
The Game class interacts with the Board class using public methods such as placePiece, removePiece, and init or initDefault. This interaction is complemented by clear and easily understandable interfaces, so Game does not require knowledge of the inner workings of Board. Therefore, changes within the Game implementation, such as move validation or board setup, can be made without affecting the Board class.

Game and Player:
The interaction between Game and Player objects is characterized by low coupling. The Game class uses the public methods of the Player class to drive player actions, such as move and turn operations. This design ensures that Game is not aware of the concrete implementation of player actions, allowing it to easily incorporate different player types (Human and Levels). Changes to player behavior or the addition of new player types can be made without affecting the Game class, encapsulating player-related logic within the Player classes.

Separation of Game and Display Classes:
The use of the Observer Pattern greatly decouples the game logic from the display logic. Game maintains a list of observers, such as Text and Graphics, and notifies them of state changes. This decoupling ensures that changes in the game's state do not directly affect the display components. Observer classes simply observe notifications and do not require details of game state management. This approach provides flexibility in adding new display types and modifying existing ones without impacting the main game logic, enhancing the system's flexibility and maintainability.

By adhering to the principles of high cohesion and low coupling, the chess game application design ensures a modular, maintainable, and extendable system. Each class has a well-defined

role and interacts with other classes through clear interfaces, making it simple to change and enhance the game's functionality.

6, Answer to Questions:

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, this link, 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.

Answer:
To implement a book of standard openings, we can develop an Opening class dedicated to logging the initial moves of each player along with the game's outcome—win, draw, or loss. This historical data will be stored in a map structure within the Game class, mapping each unique sequence of moves to their corresponding performance statistics. This map will be updated after each game to reflect the latest results against certain other openings used by opponents, continuously refining our database. This will allow the chess engine to provide strategic recommendations based on empirically derived success rates, thereby improving the player's opening repertoire.

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:
For the undo feature, we plan on creating a stack called PrevMoveStack. This class will contain all the Move type objects specifying starting location, ending location, which piece is moved, and whether it made a capture. When a player moves a piece, a Move object will be generated and pushed into the PrevMoveStack stack. In the undo function, we will pop the last move from the Historical stack and reverse that move, which consists of moving the piece from the end location to the starting location. If there was a piece taken in the move that the user wants to undo, the type is retrieved, and the piece's object is placed back in its position.

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

The classes and methods already implemented, such as Piece, Observer, Player, Tile, and Game, do not need to be drastically changed. Major changes pertain only to flexible configuration on the board and changes in game logic according to the rules and flow of the four-handed chess game.

Board Configuration:

The existing Board class will be abstracted to allow configurations for different types of boards. We will have two subclasses: NormalBoard, which represents the classic 8x8 board, and FourHandedBoard, which extends the board on both sides of the conventional 8x8 grid by 3 rows of 8 cells. The initialization logic in FourChessBoard will set up the extended grid properly and place the pieces accordingly.

Piece Configuration:
Four-handed chess requires each player to have their own set of pieces, usually represented with different colors. This requires adding two more sets of chess pieces and adjusting the starting positions. Initialization will be done through the initializeBoard method of the FourHandedBoard subclass, ensuring that each player's pieces are placed correctly at the beginning of the game.

Game Mechanics:
While the movement of pieces follows standard chess rules, the scoring system must be adjusted. Points are awarded for capturing pieces, with different values for each type of piece, and higher scores for checkmating a king. The updateScore method in the Game class will be modified to reflect the new scoring system.

Turn Order:
The Game class will maintain the turn order, which will be clockwise among the four players. This can be managed by iterating over a vector of players.

Game Over Conditions:
The game should terminate when three players have lost, or when one player has accumulated enough points that the remaining player cannot overtake them. The isGameOver method in the Game class will be adapted to detect these conditions.

Player Elimination:
When a player's king is checkmated, they lose, and all their pieces still on the board are removed. This will require updating the game state by removing these pieces and modifying the strategy of the remaining players.

These changes will adapt the program to support four-handed chess, a new and engaging way of playing chess. The significant changes include implementing the FourHandedBoard subclass, updating piece configuration and placement, scoring fields, ensuring proper turn order, and revising end game conditions. This allows the program to function in a four-player game while preserving the core elements that make chess enjoyable.

7, Extra Credit Features

We added an additional level to the computer player (level 5) for our extra credit feature. It is easier to talk about all five levels in this one section.

Level 1 is a completely random move generator. We take in all valid moves and randomly select one and that is the move that the computer performs.

For level 2, we prioritize moves that will capture and put the opponent in check (in that order). To do this, we iterate through all possible valid moves, then save those which have their destination point capture an enemy piece and those which will put the enemy in check. We then randomly select a move going down the priority list.

For level 3, we add a top priority of having our pieces avoid being captured and a lower priority of moving to a non-capturable location. To handle our pieces not being captured, we iterate through all possible valid moves, then select those whose source point can be moved on by the enemy on their next move. From these source points, we find destination points that won't be moved on by the enemy as well, so the piece will move to safety. We follow a similar method for finding moves going to a non-capturable location.

For level 4, we add a feature that gives a computer priority within each priority level instead of randomly selecting moves at that priority level. We determine priority based off the standard points of each piece in a game of chess. We add this priority for avoiding capture, so if there are two pieces that can be captured by one move, the computer will now move the piece with the higher value away to safety instead of randomly selecting one.

For level 5, we introduce the concept of a "move score", which is determined by the value a move can get by capturing and enemy, how much value it will lose by getting a piece captured on the next move, and how much value can be saved by moving a piece away from a capturable location. We determine the values of each subscore in a similar way as described previously. We then sort the vector of moves based on it's move score and select the move with the highest move score.

8, Final Questions

Question 1: What did this project teach you about writing software in teams? If you worked alone what did you learn about writing large programs?

Developing software, whether in a team or alone, taught us several important lessons. Clear communication within the team and frequent collaboration help prevent misunderstandings and ensure everyone is aligned with the project goals. The use of version control systems is fundamental for tracking changes and avoiding code conflicts, which is especially crucial when multiple people are working on the same codebase. Effective task distribution, based on team members' strengths, enhances productivity and the quality of the results.

When working alone, meticulous planning and time management become essential. Breaking down the project into smaller, manageable tasks helps maintain steady progress. Proper documentation and organized code are crucial for tracking and debugging. Rigorous testing and iterative development also play vital roles in ensuring the project's success.

Question 2: How would you go back and do things differently?

If we could start over, we would invest more time in the early stages of design and planning to establish a solid foundation for the project. This would involve creating detailed specifications, design documents, and project timelines. Additionally, establishing comprehensive test suites early on would facilitate faster debugging and smoother development.

This chess application serves as a clear, sensible, and maintainable example of a C++ game. We employed several design patterns, including the Observer Pattern, to ensure separation of concerns and extensibility within the architecture. Our approach emphasized functionality, performance, and future extensibility, laying a strong foundation for continued development and improvement. This project exemplifies our commitment to best practices in software engineering and object-oriented design.

9, Conclusion

This project is an excellent example of a structured and maintainable game developed in C++. We applied design patterns, such as the Observer Pattern, to clearly separate concerns and create an extensible architecture. Our focus on functionality, performance, and future extensibility has laid a strong foundation for further development and potential feature enhancements. This project showcases our commitment to best practices in software engineering and object-oriented design.