Chess Final Project Design Document

Chess project by: Jerry Lin , Pranav Vajpeyi, Alim Dhuka

Overview and Introduction

Our final design and original UML differed somewhat, where most of the changes could be found in the board and controller component of the UML. This was
Due to the fact we underestimated the complexity of a MVC design and the logic behind controlling the game state when first planning. We had some amounts of “redundant” functions and parameters that could be handled by other functions when we actually put the code into practice. We had a lot of trouble when it came to getting the program to make moves due to the complexity of the code and the amount of logic checks that were in place for the program. As such it was hard to debug what went wrong and in what function that was causing the logic error. Looking back we should have started small in scale when it came to compiling and integration. We split up the files among ourselves and tried to compile them all at once which lead to several errors and alot of back and forths before getting a compiling version. A Lot of time was spent debugging mistakes that were “small” but did affect the logic of the code greatly

Our attack remained more or less the same through the course of the project. We made the piece class firsts with their respective moveset logic and distributed the work evenly. Then we moved on to the game flow logic (i.e controller, board, square etc), followed by the text and graphic displays and then finally the AI implementation which took the longest time. For our chess game to function properly we used the observer pattern that was taught in class. Every Time there is a move or a change in game state the observers will update the displays on the status of the board to display the most up to date board state. This was done using a notify function within the parent observer.

Due to time constraints we were unable to implement some of the functions we desired. For instance, in controller.cc we wished for a section that had (cmd == “undo”) for the user to call undo. We implemented a version of undo that worked with the ai but we were unable to get it working in the controller part of the code which was really strange, meaning human players are unable to call undo. Our plan to implement this was to make a stack/vector that contained the board states that were updated each time a valid move was completed and when undo would be called we would try to use the stack to restore the most recent board state. However, it was noticed upon debugging that this would cause a segmentation fault every time we tried to undo due to getName() attempting to return a value of a nullptr. Similarly when we go into setup mode the program is able to add pieces but might segment fault if we try to remove a piece. We are unsure of the exact removal conditions that cause the segment fault, but this is likely related to why we are unable to undo properly as these errors were found late in the development cycle. Moreover we wished to load .pngs of the chess pieces through Xwindow and Graphic display to improve the aesthetic of the X11 interface but encountered issues doing so and decided to stick with the capital and lowercase letter display of pieces. Finally, the idea to implement puzzles was an initial thought that failed to make it to the final version of the project.

Extra Features

Due to time constraints there aren't a lot of notable extra features, but we did implement a save function that converts the state of the board into their corresponding chars onto a text file called board.txt that can be loaded on later when the user wishes to resume their game with another player. For the level 4 ai we implemented Piece-square tables which are essentially 2D arrays of values for a piece on a certain square on the board that determine how much the piece is encouraged to move to certain squares while avoiding others. This adds another aspect of move weighting aside from simply capturing, avoiding capture and checking since there is a lot of importance when it comes to controlling the board and putting your pieces at an advantageous position.

Key Components and Structure

- **Board**: Manages the state of the game board, including the placement and movement of pieces.
  - The chess board is a 2D array where each element of the array is a square class
- **Piece**: Represents the base class for all chess pieces, providing common functionality and attributes.
  - Each piece has a variable pos which is a pointer to a square on the board which, representing its position
- **Square**: Represents individual squares on the chess board, managing piece placement and capturing.
- **Move**: Represents a chess move, including starting and ending positions of each move.
  - The Move class makes use of copy and copy assignment to perform the moving of a piece
- **Player**: Represents the base class for players, both human and AI.
  - Humans have the ability to control pieces on the board
- **AI**: Implements various AI levels for automated gameplay.
  - Inherits from Player and represents an AI player in the game. This class includes common functionalities for different levels of from 1-4

    - **Level 1**
      - Inherits from AI and represents the simplest level of AI. This AI level selects moves randomly.
    - **Level 2**
      - Inherits from AI, prefers capturing moves and checks over other moves
    - **Level 3**
      - Inherits from AI, prefers avoiding capture, capturing moves, and checks.
    - **Level 4**
      - Inherits from AI, prefers being certain positions, prefers avoiding capture, capturing moves, and checks, avoids moves

that would allow enable other pieces open to capture after moving

- **Controller**: Manages game flow, player interactions, and game state updates and manages player inputs to the program.
    - Interacts with Player, TextDisplay, GraphicsDisplay, and Board classes.
- **View**: Base display class for different display types, including updates text and graphics.

Main Implementation Details

Board Class

- **Initialization**: The Board class initializes an 8x8 grid of Square objects. The boardSetup() method places pieces in their starting positions, while clearGame() removes all pieces from the board.
- **Piece Management**: The pieceSetup() method places specific pieces on the board onto squares, and removePiece() handles piece removal and updates the state of the board.
- **Validation and Moves**: The isValid() method checks if a move is valid based on the piece's movement rules and board state. shiftPiece() performs the move, updating the board and notifying the controller.
- **Special Moves**: The overrideMove() and undo() methods handle special moves such as castling and en passant. And allows AI to evaluate a potential move.

Piece Class and Inheritance

- The Piece base class contains common functionality of the pieces and contains any common variables that make up the piece.
- **Storing Moves of Pieces**: Each piece contains a private method "moveSet" which contains all possible moves for the piece and will also have a public virtual method "validMoves" which will assess legal moves depending on the piece's current position.
    - The capturability of the positions are done in Square. Squarehas public methods getCapture and setCapture which take the current player's color as a parameter and determine if the square is capturable by the player's piece
- **EnPassant**: The pawn has a method "enPassant" which will determine if enPassant is a legal move in its current position. It also contains a bool variable "jumped" which will be set to true if the pawn "jumped" on the first move.
- **Castling**: The King will be able to handle King-side and Queen-side castling by validating its own position and the rook's position as well as if there are any pieces in between them, which is implemented in its validMoves method.
- **Validity of King Moves**: Checking if a King is entering a position that places it in check is done using the getCapture and setCapture methods of Square as it indicates if the square that the king is moving to is capturable by a piece of the opposite color.

Player and AI Classes

- The base player class contains a public virtual method that processes and returns the move that the player wants to make.
- **Human Player**: The Human class reads/processes the moves that a human player wants to perform.
- **AI Player**: The AI class and its derived classes (Level_1, Level_2, Level_3, Level_4) implement various AI strategies for automated gameplay. Activate it by simply typing move into stdin. All of the AI uses the C++ random Library and uses mt19937 gen(rd()); In other to randomly generate a number and select that index of the moveList
    - Level 1
        - The generateMove function will randomly select a vector of moves and return a move to be processed on the board state.
    - Level 2
        - The generateMove function will randomly select from a vector of moves that are checks, captures and random moves in that priority and return a move to be processed on the board state.
    - Level 3
        - The generateMove function will select from a vector of moves that are checks, captures, avoiding capture and random moves in that priority pieces are assigned values so even if there for example multiple captures the AI will choose the piece with more value and return a move to be processed on the board state.
    - Level 4
        - The generateMove function will select from a vector of moves that are checks, captures, avoiding capture and random moves in that priority pieces are assigned values so even if there for example multiple captures the AI will choose the piece with more value. However this time the Ai will also take into consideration the pieces position on the board into account for which move to choose and return a move to be processed on the board state.

Controller Class

- **Game Management**: The Controller class handles the overall game flow, including initializing the board, managing player turns, and checking for game end conditions.
    - The “play” method in the controller manages the actual flow of the game and interacts with the player, accepting commands from stdin
- **Observers**: The controller notifies the view (text or graphics) to update the display after each move. Or a coordinate when a piece is set up.
    - The notify methods update displays and game(board) states

View Classes

- The class Initializes a virtual display of the board using a 2D array of chars containing lower case letters for black pieces and capital letters for white pieces or ‘ ‘ and ‘_’ to replicate the square pattern on a board.
- **TextDisplay**: Provides a simple text-based representation of the game board, useful for debugging and console play.
  - The text display is outputted using the print method
- **GraphicsDisplay**: Uses Xwindow to render a graphical representation of the game board, providing a more interactive experience.
  - The update() function renders the display and ensures the board is drawn correctly with pieces displayed in their correct positions

Design Patterns and Object-Oriented Techniques

Inheritance and Polymorphism: The use of base and derived classes allows for flexible and extensible code. There were 2 main parts of the program where this technique was highlighted; in the Piece and the AI classes. New pieces or AI strategies can be added with minimal changes to existing code.

- Having the pieces inherit from a base Piece class allowed the pieces to maintain common core functionality of all pieces while being able to address the specific needs for certain pieces, like the need for enPassant in the Pawn, Castling with King and the move behavior of each piece.
- The overarching AI class contained the core functionality of the AI’s gameplay, while each level incremented its behavior

Encapsulation: Classes manage their own data and expose only necessary methods, improving maintainability and reducing dependencies.

- **Example**: The Board class manages the state of the board and utilizes public methods like shiftPiece() and isValid() for manipulating private date fields and checking the state. This allowed the program to ensure that the chess board was only being manipulated as it would during a real game of chess with real chess rules.

Observer Pattern: The controller acts as an observer, updating the view whenever the game state changes.

- This was mainly shown through the Controller class as it calls the notify() method on the view objects (TextDisplay and GraphicsDisplay) to update the display whenever a move is made.
- This was important in ensuring that any changes to the Board were reported consistently and safely without harming other parts of the program

Factory Method: The pieceSetup() method in the Board class acts as a factory for creating different pieces based on the input character.

- Example: Depending on the character passed ('k' for King, 'q' for Queen, etc.), the method creates the corresponding piece object and places it on the board.

Model View Controller: The Controller in the MVC pattern acts as an intermediary that handles user inputs, updates the Model, and refreshes the View

- In our chess program, the Controller captures user commands, translates them into actions that modify the Model (the Board in this case), and then calls methods on the View (TextDisplay and GraphicsDisplay) to update and display these changes. This design ensures a clear separation of concerns, which allows for low coupling and distribution of tasks.

Cohesion and Coupling

- **High Cohesion**: Each class has a well-defined purpose and encapsulates related behavior. For instance, the Board class strictly manages board state and also takes care of the placement of pieces. The Controller class handles game logic, player interactions, as well as setting up the board. The move class encapsulates information about a move for a piece, including the starting and ending positions of a move and any special considerations like enPassant. We tried to adhere to the doctrine that one class only does one thing as well as responsibility driven design (the Controller does not directly manipulate piece positions but instead delegates this to the Board class)
- **Low Coupling**: Interactions between classes are minimized through well-defined interfaces. The controller interacts with the view through abstract methods, allowing different display implementations without changing the controller logic. The Piece class uses methods to validate moves for different pieces, reducing the need for direct dependencies between piece classes and game logic.

Resilience to Change

As anticipated in DD1, there were several changes in the project specifications, such as rule modifications or additional features. Our biggest changes were the following:

- **Controller Class**: Refactored to include additional methods for saving the board state and initializing displays. This change improves code organization and separation of concerns.
- **Piece Class**: Adjusted methods to ensure proper encapsulation and inheritance hierarchies. Specifically regarding move checks with the King as we had a hard time ensuring the King wasn’t being moved into check and with castling. We had to restructure the King class and edit the validMoves() function to ensure that these components were being checked
- **Display**: We also had to make changes to the notify() and update() methods in the display to ensure that any updates would be properly replicated on the display.

Q&A

1. Standard Opening Moves (not implemented):

To implement standard chess openings:

- **Create Database**: Build a database of common opening moves (we can get this online) and their responses. Key entries by the board state and value with recommended moves. There should be alternate moves if the opening is interrupted
- **Design Database**: Structure the database to include board state, recommended moves, and responses.
- **Integrate with Game**: Modify the AI to use the database for opening moves. Match the board state with stored openings and find the appropriate response.
- **Fallback to Normal Play**: After the opening phase, switch to normal game logic.
- **Test Implementation**: Ensure the openings are played correctly and validate with chess reference sources.

2. Undo Last Move (attempted implementation but were unsuccessful):

To add an undo feature:

- **Track States**: Use a stack to record board states after each move.
- **Push States**: Push the current state onto the stack after each valid move in controller.cc. The stack should resize dynamically.
- **Undo Function**: Implement an ‘undo’ function to revert to the previous state by popping the stack and deep copying the board and all the associated pieces so we make sure the board's states are independent and don't affect each other.
- **UI Integration**: Add a command to trigger undo. Make it easy to use. (can check simply if user typed “undo”)
- **Handle Edge Cases**: Manage cases where the stack is empty, and ensure game consistency.
- **Test Thoroughly**: Check that undo works in various scenarios and the game handles multiple undos without issues experimenting with edge cases.

3. Four-Handed Chess Variant (not implemented):

To adapt the game for four players:

- **Expand Board**: Design a larger or custom board for four players and place pieces accordingly.
- **Adjust Rules**: Define rules for four-handed chess, including turn order and piece interactions. Decide on free-for-all or team play. Would need to add 4 colors for pieces and determine turn order.
- **Update Game Logic**: Modify the game engine to handle four players, including turn order and move validation.
- **Modify UI**: Adjust the display to show the expanded board and additional pieces. Ensure it supports inputs and outputs for four players.

- **Handle Interactions**: Implement logic for player interactions and balance the game. Make sure everything in the original 8x8 board also works in the new 4 player version.

Lessons Learned, Team Development

- **Code Integration**: Ensuring smooth integration of code from different team members requires clear communication and adherence to coding standards.
- **Version Control**: Using version control effectively helps manage changes and resolve conflicts.
- **Constant Live Communication:** Make sure members know what documents you are working on so on commit it does not overwrite another person's code.

Writing Large Programs

- **Modularity**: Breaking down the program into smaller, manageable modules makes development and debugging easier.
- **Documentation**: Maintaining thorough documentation is crucial for understanding complex code and facilitating future modifications.

Improvements for Future Iterations

- **Testing**: Implementing more comprehensive unit tests would help catch bugs early and ensure the reliability of the code. A better way to test the code for practical application would be to use Grandmasters' played games, which use all of the chess pieces and diverse strategies.
- **User Interface**: Enhancing the graphical interface to provide a more interactive and user-friendly experience.
- Better test cases (.in files) should have made more files so

UML Changes

Classes and Their Variables and Functions

1. **Board**
   - Added or Modified:
     - BoardState: Board
     - Board* gamemaster
     - vector<Piece*> whitePieces
     - vector<Piece*> blackPieces
     - Removed moveHistory(): void
     - CheckWin(turn: int) : bool
     - overrideMove(curTurn : Move): void
     - isValid(curTurn : Move, color : int) : Bool
     - undo(curTurn : Move, &previous : Board): void

- Functions added or modified:
    - void setController(Controller* gamemaster)
    - char getPieceName(int x, int y)
    - int getPieceColor(int x, int y)
    - Removed bool isCastle()
    - boardComplete() : bool
    - shiftPiece (curTurn :Move, color :int): bool

2. **Controller**
    - Added or Modified:
        - Added printScore(), saveBoard(), initDisplays()
        - Removed inputW : int, inputB : int ,playerCheck : bool
        - Added whitePlayer: int, blackPlayer: int
        - Added pawnPromote(color: int): char
        - Added setUpPlayers(bw: std::string&): int
        - Remove -inputPlayer(p : string) : int
    - Type Change
        - textDisplay: TextDisplay*
        - graphicDisplay: GraphicsDisplay*
3. **GraphicsDisplay**
    - Added or Modified:
        - void updateBoard(Board* board)
4. **TextDisplay**
    - Added or Modified:
        - void updateBoard(Board* board)
5. **Human**
    - Removed
        - color : char
        - numMoves : num
6. **Player**
    - Removed
        - Name : string
        - white-player : bool
        - black-player : bool
        - board : Board
        - getiswhite() : bool
        - getisBlack() : bool
7. **Ai**
    - Removed
        - Extra numMoves : int
8. **View**
    - Removed
        - print(out : ostream) : void
        - update() : void
        - update(xPos : int, yPos : int): void
9. **Pieces**
    - Added or Modified:
        - pos : Square*
        - board: Board*

        - setBoard(setUp : Board*) : void

10. Square

- Added or Modified:
    - curPiece: Piece*
    - x :int
    - y :int
    - game: Board*
    - getCurrentPiece(): Piece*
    - setPiece(p: Piece*): void
    - removePiece(): void
    - setBoard(control: Board*): void

11. Pawn

- Removed:
    - enPassant : bool

Changes Summary for UML

The UML changes primarily involve significant updates to the `Board` and `Controller` classes, including the addition of variables for game state management, piece tracking, and display management. Several functions related to move history and validation were removed, while new functions for setting controllers, scoring, saving the board, and initializing displays were added. The `GraphicsDisplay` and `TextDisplay` classes were updated to include functions for updating the board state. The `Human`, `Player`, and `AI` classes had various player-related variables and functions removed, while the `Pieces` and `Square` classes saw additions and modifications to variables and functions related to piece management and board interaction. The `Pawn` class had the en passant variable removed.