CS 247 Final Project - Chess

Muneeb Farrukh, Darsh Patel, Sparsh Patel

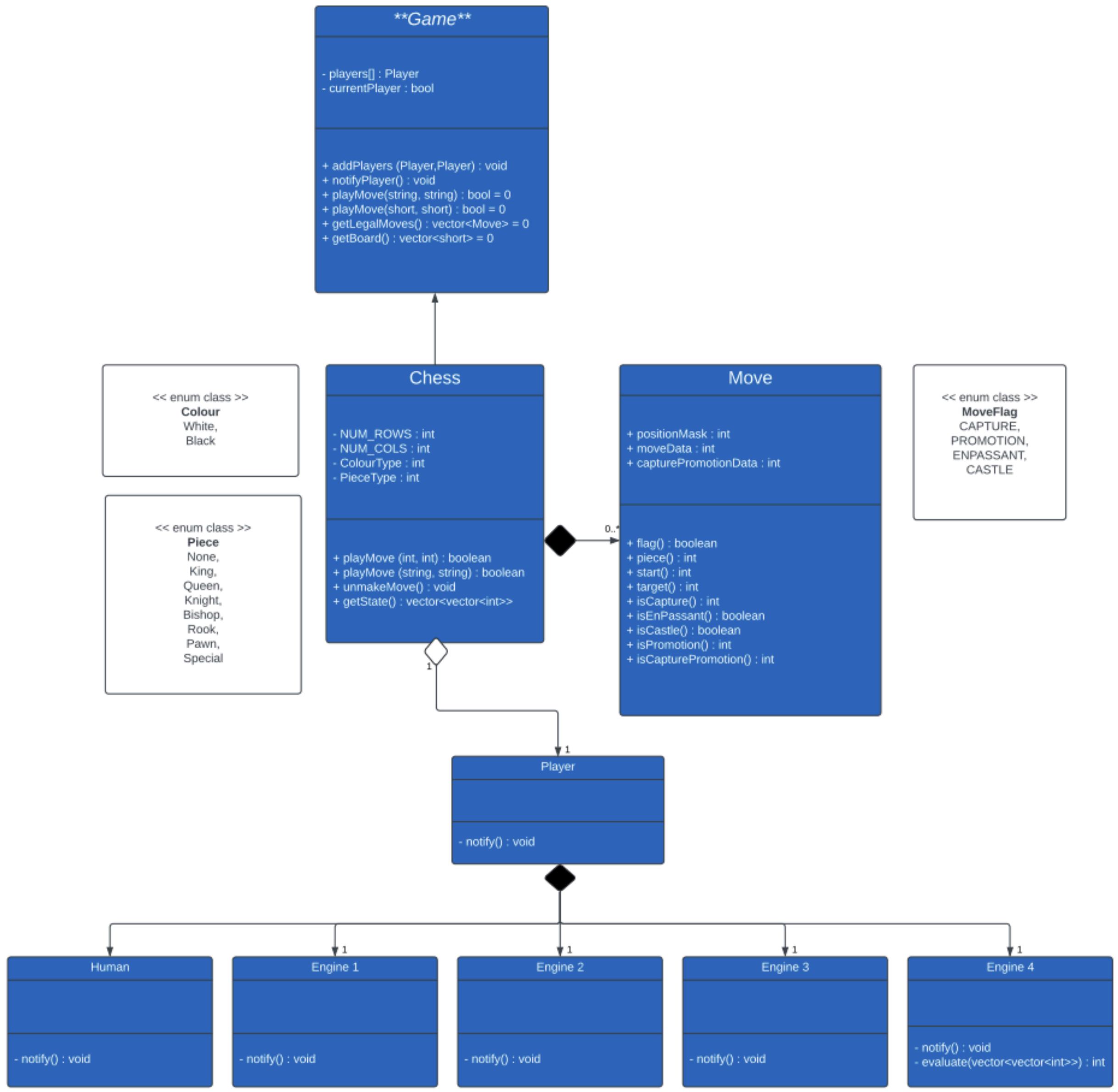
Overview

This project aims to develop a comprehensive chess game application featuring core gameplay mechanics, advanced features, and a user-friendly graphical interface. The key features we have implemented include:

1. **Command Interpreter**
    - **Game Commands**: Implemented basic commands such as `game`, `resign`, and `move`, allowing players to start a new game, concede, and make moves, respectively.
    - **Move Representation**: Implemented the `drawBoard()` function to visually represent the chessboard and display all pieces after each move.
    - **Setup Mode**: Added commands for custom board setup, enabling players to place and remove pieces and set the initial configuration.
2. **Piece Implementation**
    - **Movement Logic**: Coded the movement logic for all six types of chess pieces—King, Queen, Bishop, Rook, Knight, and Pawn—ensuring each piece adheres to the standard rules of chess.
    - **Special Moves**: Implemented special moves including castling, en passant, and pawn promotion.
3. **Advanced Game Features**
    - **Turn Tracking**: Developed a turn tracking system to manage the alternating turns between players.
    - **Move History**: Recorded move history to facilitate undo functionality and provide a replay of moves.
    - **Check and Checkmate Detection**: Implemented logic to detect check and checkmate scenarios and update the board display with relevant messages.
    - **Stalemate Detection**: Added conditions to identify and declare stalemates.
    - **Scoring**: Implemented a scoring system to track and display the final score upon game termination, awarding points for wins and draws.

4. **Bonus Feature**
    - **Undo Feature:** Designed to support undoing the last move, with considerations for implementing an unlimited number of undos.
5. **Graphical Display**
    - **Text-Based Display**: Created a text-based display to show the chessboard state using a grid of characters representing pieces and empty squares.
    - **Graphical Interface**: Developed a visually appealing graphical interface using X11 to enhance the user experience, with pieces represented as letters for simplicity.
6. **Computer Player**
    - **AI Logic**: Implemented multiple levels of AI difficulty:
        - **Level 1**: Random legal moves.
        - **Level 2**: Preference for capturing moves and checks.
        - **Level 3**: Preference for avoiding capture, capturing moves, and checks.
        - **Level 4+**: More sophisticated strategies, incorporating deeper game analysis and opening move sequences.
7. **Testing and Error Handling**
    - **Error Handling**: Ensured the program robustly handles misspelled commands and invalid moves, maintaining stability and user experience.
    - **Testing:** Performed testing for all possible edge cases by generating a thorough testsuite

Updated UML


Game
- players[] : Player
- currentPlayer : bool
+ addPlayers (Player,Player) : void
+ notifyPlayer() : void
+ playMove(string, string) : bool = 0
+ playMove(short, short) : bool = 0
+ getLegalMoves() : vector<Move> = 0
+ getBoard() : vector<short> = 0
<< enum class >> Colour White, Black
<< enum class >> Piece None, King, Queen, Knight, Bishop, Rook, Pawn, Special
Chess
- NUM_ROWS : int
- NUM_COLS : int
- ColourType : int
- PieceType : int
+ playMove (int, int) : boolean
+ playMove (string, string) : boolean
+ unmakeMove() : void
+ getState() : vector<vector<int>>
0..*
Move
+ positionMask : int
+ moveData : int
+ capturePromotionData : int
+ flag() : boolean
+ piece() : int
+ start() : int
+ target() : int
+ isCapture() : int
+ isEnPassant() : boolean
+ isCastle() : boolean
+ isPromotion() : int
+ isCapturePromotion() : int
<< enum class >> MoveFlag CAPTURE, PROMOTION, ENPASSANT, CASTLE
1
1
Player
- notify() : void
Human
- notify() : void
1
Engine 1
- notify() : void
1
Engine 2
- notify() : void
1
Engine 3
- notify() : void
1
Engine 4
- notify() : void
- evaluate(vector<vector<int>>) : int

Changes to UML: Our updated UML was expanded to include a comprehensive `Game` class, which manages the overall game state, players, and current player. The `Engine` class was replaced with a `Player` class, allowing for both human players and different levels of AI players (`Engine1` to `Engine4`). The `Game` class includes methods for adding players, notifying players, playing moves, retrieving legal moves, and getting the board state. This addition provides a more structured and holistic approach to managing the game, encompassing player interactions and game flow. The `Chess` and `Move` classes retained their core attributes and methods, while the enum definitions for `Colour`, `Piece`, and `MoveFlag` remained consistent.

Design

Challenge: Implementing Multi-Level Chess AI

Description: Implementing four levels of chess AI presented several challenges, including creating distinct strategies for each AI level, ensuring real-time responsiveness to game changes, and managing the performance impact of multiple AI observers. The complexity increased as each AI level required unique algorithms, and maintaining a system that could easily adapt to changes or additions was crucial for future updates.

Solution: To address these challenges, we utilized the Observer Pattern with the `Player` class as the subject. Each AI level was implemented as a concrete observer with its specific strategy. The `Game` class managed the registration and notification of these AI observers, allowing them to react to moves and game state changes efficiently. This design ensured that adding or modifying AI levels was straightforward and did not impact the core functionality, thus maintaining performance and ease of maintenance.

Challenge: Handling Complex Move Validations

Description: Validating complex chess moves, such as castling, en passant, and pawn promotions, is intricate due to their unique rules and interactions. Ensuring that all special move conditions were correctly checked without compromising performance was a significant challenge.

Solution: We modularized the validation logic by implementing specialized data structures for each type of move. For `castling` we utilized an array to check if either the King or Rook has moved and whether a castle has already occurred or not. For `en passant`, we utilized a stack to keep a history of previous moves to check if there was a double pawn push on the previous move. This approach allowed us to clearly separate and manage the rules for each move type. By integrating these checks into a unified move validation function, we ensured that each move was validated efficiently and accurately.

Challenge: Managing Game State Consistently

Description: Maintaining and updating the game state consistently across various operations, including moves, captures, and special conditions, required careful tracking of the board, castling rights, and move history.

Solution: We implemented a comprehensive state management system that updates the game state after each move using the `setGameState` method. This method evaluates the current board position, checks for conditions such as checkmate, stalemate, or check, and updates the `gameState` variable accordingly. We utilized bitboards to track which color could capture which squares and leveraged this information to ensure the generation of valid moves. Additionally, we added a bonus `unmakeMove` function, allowing users to undo a move, and we were able to ensure the state was managed correctly when this function was used.

Resilience to Change

1. Modular Design

- **Highly Cohesive Design**: The engine's functionality is divided into distinct modules such as move generation, move handling, and castling. This modular approach ensures that changes to one part of the system (e.g., how moves are generated) do not significantly impact other parts (e.g., move validation).
- **Encapsulation**: Each module (e.g., `genPawnMoves`, `genKingMoves`, `makeMove`) encapsulates its specific functionality. This means that changes to the logic of generating pawn moves are isolated from changes in how moves are handled or how the board state is managed. This allows us to implement new move patterns for certain pieces if required.

2. Flexible Move Handling

- **Move Generation**: The design allows for flexible generation of moves by using generic functions like `generatePseudoLegalMoves()` and `generateLegalMoves()`. These functions can be extended or modified to accommodate new types of moves or special rules without major changes to the core logic.
- **Special Moves**: The handling of special moves (castling, en passant, promotions) is implemented in dedicated functions. This modular approach makes it easier to modify or extend these rules, such as adding new special move types or adjusting existing rules.
- **Attack Mask Updates**: The `getAttackSquares()` function updates attack masks for all pieces. This centralized approach to managing attack information supports changes in attack rules or piece behavior without needing to alter individual piece movement logic.

3. Testing and Debugging

- **Unit Testing**: The design supports comprehensive testing of individual components. Unit tests for move generation, move handling, and special moves can be created to ensure that changes do not introduce regressions..

4. Documentation and Design Patterns

- **Clear Documentation**: Documentation of design choices, functions, and expected behavior helps future developers (or us in the future) understand and implement changes effectively. Clear comments and documentation facilitate easier modifications and extensions.
- **Software Engineering Principles/Design Pattern Usage**: The use of design patterns like Observer Pattern along with our implementation of encapsulation and modularization supports resilience to change. For instance, implementing a strategy pattern for move generation could further enhance flexibility.

Answers to Questions:

Question 1:

Use a tree data structure where each node represents a specific chess position, with the root as the starting position and child nodes as possible legal moves, storing additional information like move frequency and success rates. We would have to populate the tree with standard openings from an external file beforehand, adding each move sequence as a path. During the game, we would check the current position in the tree to choose the next move based on data or fallback to the standard engine's logic if none is found**.**

Question 2:

We would use a stack to keep track of moves. Each entry in the stack would represent a move and contain the necessary information to undo it (which piece was moved, original and new positions, any piece captured, etc). To undo the last move, we would pop the top entry from the stack. Then we would reverse the move by restoring the piece to its original position and reinstate any captured piece. We would also update the game state accordingly (e.g., change the turn). As long as there are entries in the stack, continue to pop and reverse moves. This allows an unlimited number of undos until the stack is empty.

Question 3:

To adapt the program for four-handed chess, we would redesign the board layout to accommodate a 16x16 board with four sets of pieces, and then we would update our `drawBoard()` function. We would also need to adjust the game rules to accommodate four players, including turn order, endgame conditions, possible alliances, and victory conditions. Move generation and validation logic would remain similar, we would just have to ensure it is updated for the larger board. Some of the special moves like castling and en passant would need to be modified. Both the text and GUI would need to be changed to display four player sections and handle inputs and game status for each player. The turn management would also require a system to track and switch turns among the four players, including handling skips and possible alliances.

Extra Credit Features

Undo Feature:

We implemented an undo feature for our chess game, allowing players to revert their last move. The `unmakeMove` function restores the game state to its previous position by reversing the last move made. This includes handling standard moves, special moves like castling, en passant captures, and pawn promotions.

Challenges Faced:

1. **Move Reversal Logic:** Ensuring that each type of move (standard, castling, en passant, promotion) is correctly reversed required meticulous attention to the specific rules governing each move.
2. **Maintaining State Integrity**: It was essential to accurately restore not only the positions of the pieces but also other aspects of the game state, such as castling rights and en passant availability.
3. **Edge Cases:** Handling edge cases like capture promotions and ensuring the game state reflects these complex scenarios added further complexity.

Solutions to the Challenges:

1. **Detailed Move Tracking:** We ensured that each move was tracked with sufficient detail to allow for precise reversal. This included storing information about the piece type, start and target positions, and any special attributes of the move.
2. **Comprehensive Reversal Logic:** We implemented specific logic for each type of move:
    - Standard Moves: Simply move the piece back to its original position.
    - Castling: Move the rook back to its original position alongside the king.
    - En Passant: Restore the captured pawn to its original position.
    - Promotions: Change the promoted piece back to a pawn.
3. **State Restoration:** We carefully restored other aspects of the game state, such as castling rights, which might have been affected by the move. We also ensured that the state was consistent and accurately reflected the game's rules.
4. **Testing and Validation:** Extensive testing was conducted to ensure that the `unmakeMove` function worked correctly in all scenarios, including edge cases. This helped identify and fix any potential issues in the implementation.

Final Questions

1. What lessons did this project teach you about developing software in teams?

   We've learned that catering to each team member's strengths and preferences is crucial for a smooth and efficient project completion. Before starting, we met to discuss how to delegate responsibilities, primarily based on individual strengths and weaknesses. We also considered each team member's interests to ensure everyone worked on something they enjoyed, which boosted morale and productivity. Additionally, this approach helped us complete the project on time. We also discovered that clear communication is vital for team success, as it ensures alignment and addresses issues promptly.

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

   If we had the chance to start over, we would have focused on documenting the code more thoroughly and spending more time investigating edge cases before implementation. Better documentation would have made it easier for team members to understand and maintain the code, facilitating smoother collaboration. Additionally, dedicating more time to exploring potential edge cases upfront would have allowed us to anticipate and address complex scenarios more effectively, leading to a more robust and reliable final product.