7. Design a Chess Game

Step 1: Outline Use Cases and Constraints

Use Cases

- **Two-Player Online Chess Game**: The system should allow two online players to play a game of chess.
- **Rule Enforcement**: All rules of international chess should be followed.
- **Random Side Assignment**: Each player is randomly assigned a side, either black or white.
- **Turn-Based Gameplay**: Players alternate turns, with white always making the first move.
- **Move Restrictions**: Players cannot cancel or roll back their moves.
- **Game Log Maintenance**: The system should keep a record of all moves made by both players.
- **Game Conclusion**: The game can end via checkmate, stalemate, resignation, or forfeit.

Constraints and Assumptions

- The system will support real-time, two-player online play.
- Only standard chess rules will be implemented.
- The initial version will not include AI-based opponents or puzzle modes.
- The system will maintain move logs for replay or review purposes.
- Network latency should be minimized to ensure smooth gameplay.

Step 2: High-Level Design

- **Client-Server Architecture**: The chess game will have a client-side UI and a backend server to process moves and enforce rules.
- **Database for Game States**: A database will store active and completed games, including move history.

- **Turn-Based Communication**: The server will ensure correct turn order and validate each move before updating the game state.
- **Authentication & Player Management**: Users must register and log in to play.
- **Admin Panel**: Admins will have the ability to manage users (ban/modify accounts).

Step 3: Core Components

Use Case Diagram

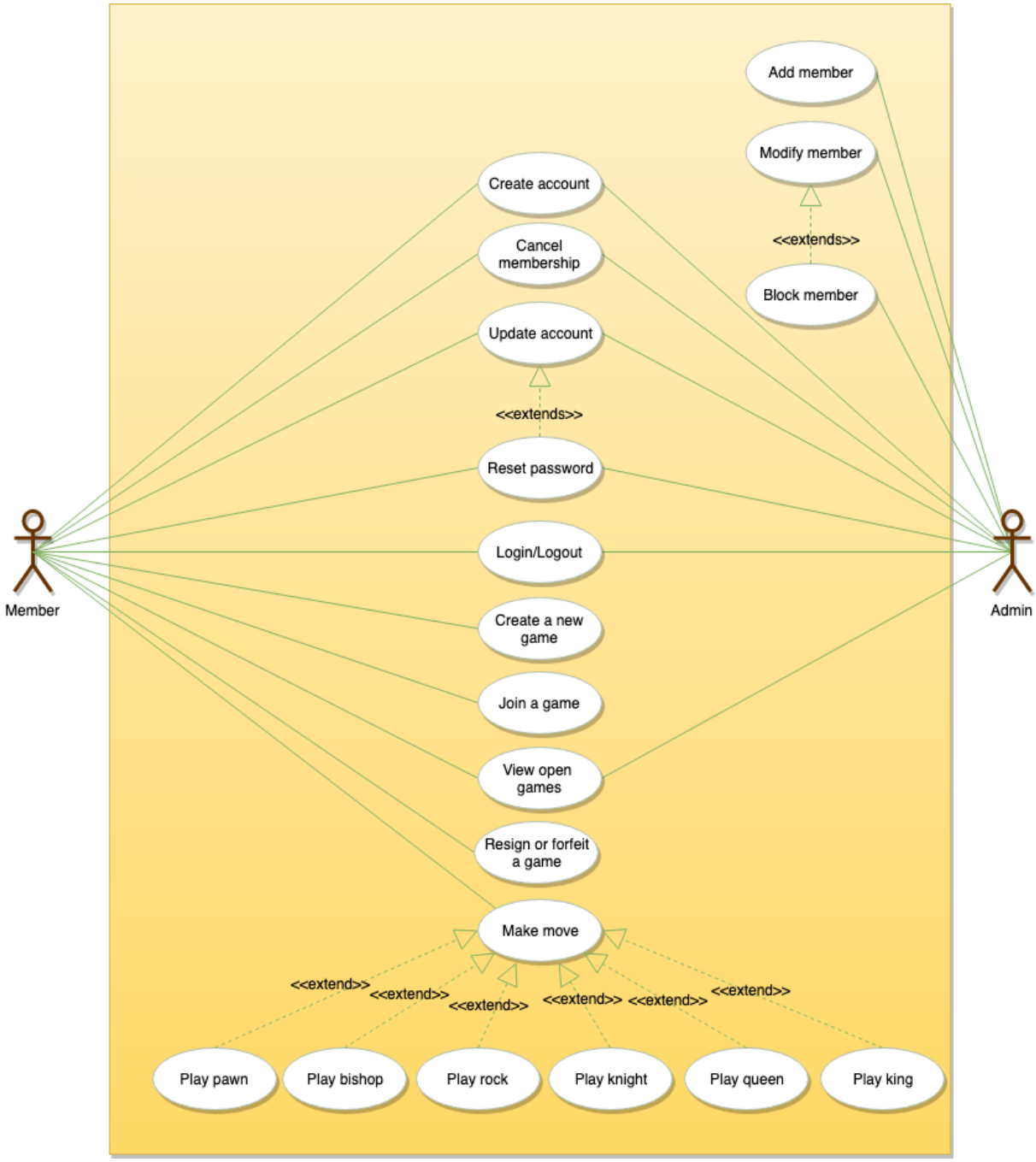
We have two actors in our system:

- **Player**: A registered account in the system, who will play the game. The player will play chess moves.
- **Admin**: To ban/modify players.

Here are the top use cases for chess:

- **Player moves a piece**: To make a valid move of any chess piece.
- **Resign or forfeit a game**: A player resigns from/forfeits the game.
- **Register new account/Cancel membership**: To add a new member or cancel an existing member.
- **Update game log**: To add a move to the game log.

Here is the use case diagram of our Chess Game:

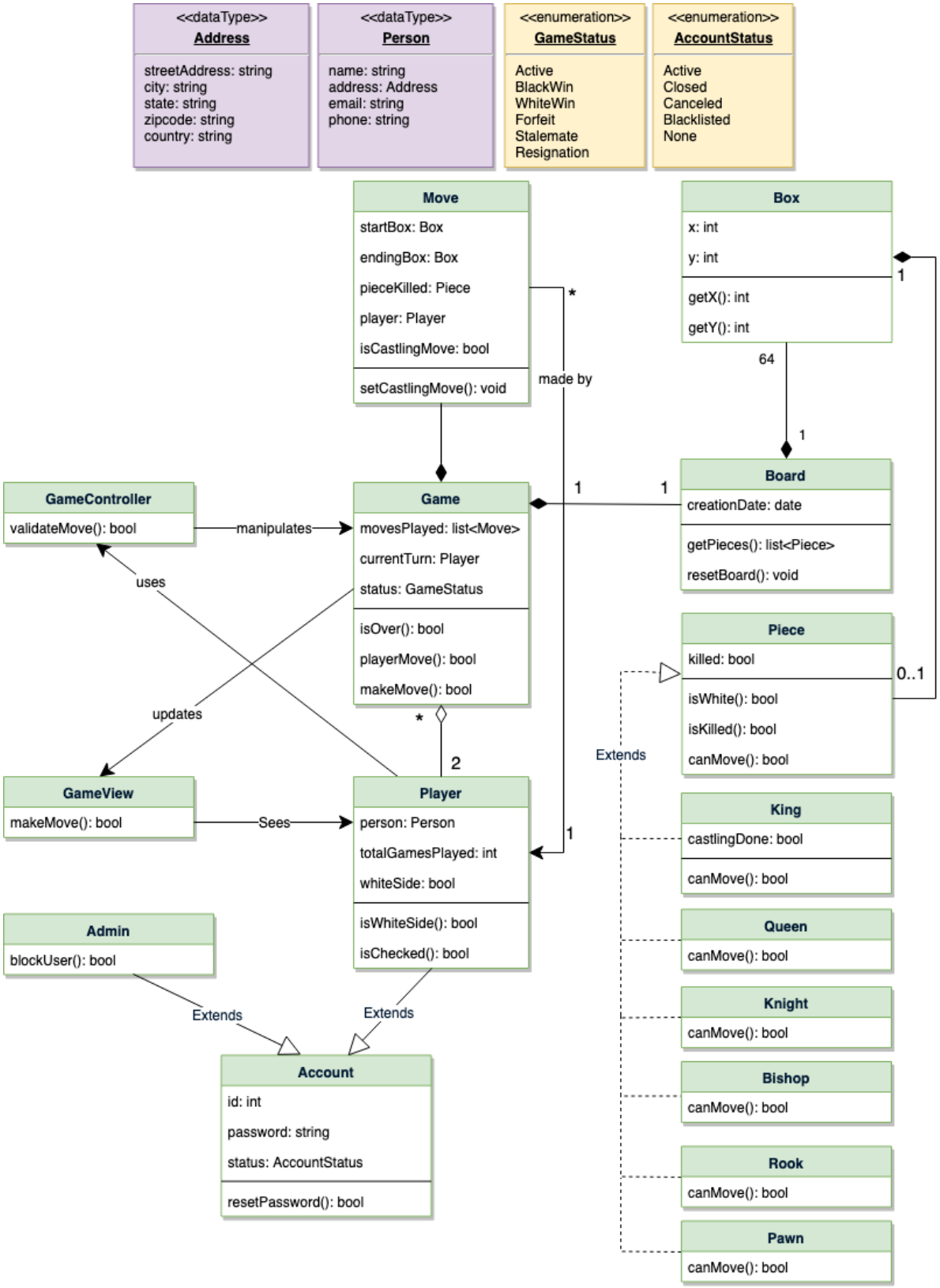


Use Case Diagram for Chess

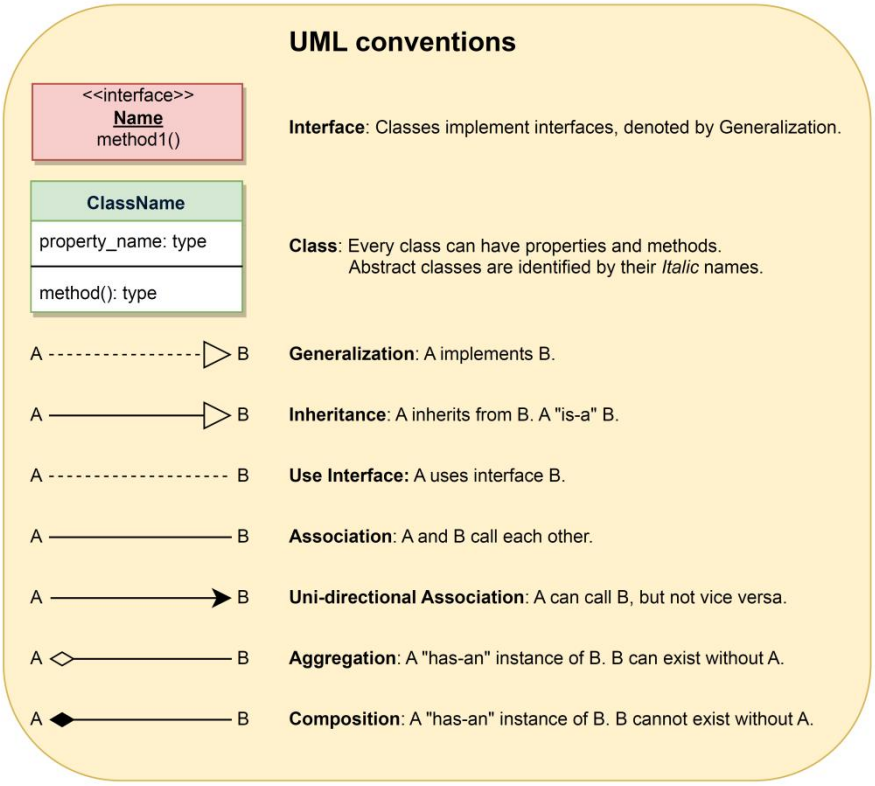
Class Diagram

Here are the main classes for chess:

- **Player**: Player class represents one of the participants playing the game. It keeps track of which side (black or white) the player is playing.
- **Account**: We'll have two types of accounts in the system: one will be a player, and the other will be an admin.
- **Game**: This class controls the flow of a game. It keeps track of all the game moves, which player has the current turn, and the final result of the game.
- **Box**: A box represents one block of the 8x8 grid and an optional piece.
- **Board**: Board is an 8x8 set of boxes containing all active chess pieces.
- **Piece**: The basic building block of the system, every piece will be placed on a box. This class contains the color the piece represents and the status of the piece (that is, if the piece is currently in play or not). This would be an abstract class and all game pieces will extend it.
- **Move**: Represents a game move, containing the starting and ending box. The Move class will also keep track of the player who made the move, if it is a castling move, or if the move resulted in the capture of a piece.
- **GameController**: Player class uses GameController to make moves.
- **GameView**: Game class updates the GameView to show changes to the players.



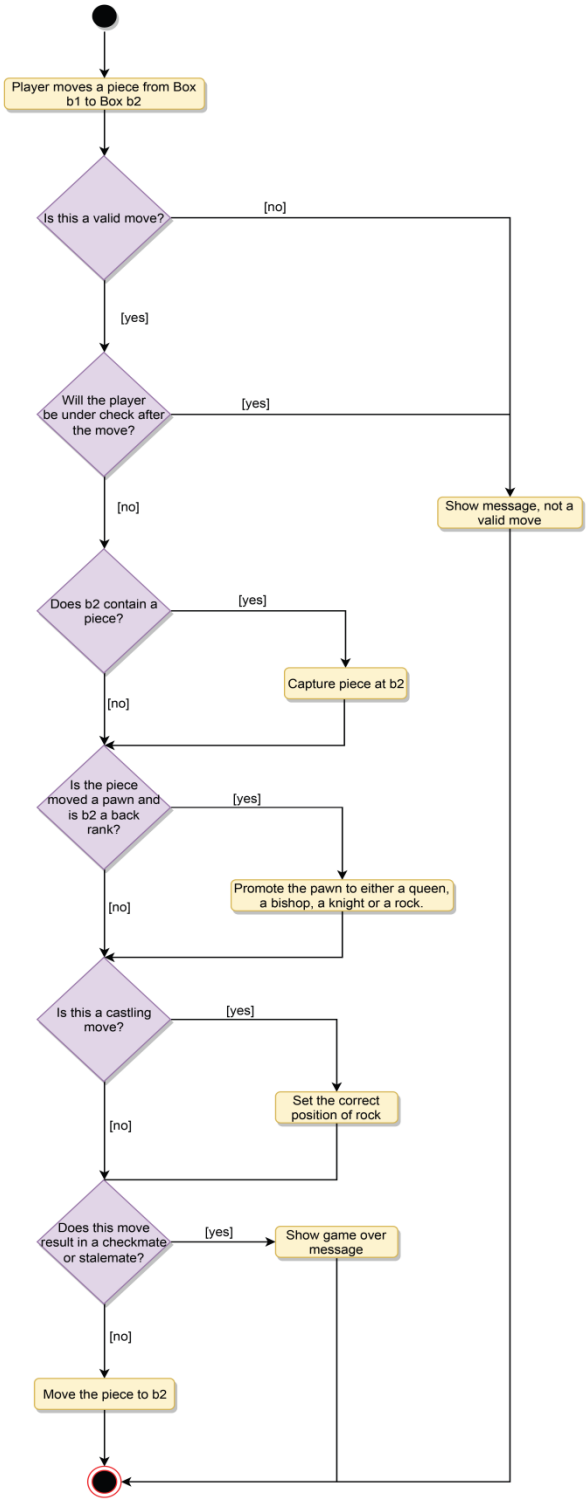
Class Diagram for Chess



UML for Chess

Activity Diagram

Make move: Any Player can perform this activity. Here are the set of steps to make a move:



Activity Diagram for Chess

Code:

Enum Definitions

```
enum Color {
   WHITE, BLACK;
}
```

Position Class

```
class Position {
   int row, col;
   public Position(int row, int col) {
      this.row = row;
      this.col = col;
   }
}
```

Abstract Piece Class

```
abstract class Piece {
   protected Color color;
   protected Position position;

   public Piece(Color color, Position position) {
      this.color = color;
      this.position = position;
   }

   public abstract boolean isValidMove(Position newPosition, Board board);
}
```

Concrete Piece Classes

```
class King extends Piece {
   public King(Color color, Position position) {
      super(color, position);
   }
   @Override
   public boolean isValidMove(Position newPosition, Board board) {
      int rowDiff = Math.abs(newPosition.row - position.row);
      int colDiff = Math.abs(newPosition.col - position.col);
      return rowDiff <= 1 && colDiff <= 1;
   }
```

```
}

class Queen extends Piece {
    public Queen(Color color, Position position) {
        super(color, position);
    }
    @Override
    public boolean isValidMove(Position newPosition, Board board) {
        return (position.row == newPosition.row || position.col == newPosition.col ||
                Math.abs(position.row - newPosition.row) == Math.abs(position.col - newPosition.col));
    }
}
```

Board Class

```
class Board {
    private Piece[][] board;

    public Board() {
        board = new Piece[8][8];
        setupBoard();
    }

    private void setupBoard() {
        board[0][4] = new King(Color.BLACK, new Position(0, 4));
        board[7][4] = new King(Color.WHITE, new Position(7, 4));
        board[0][3] = new Queen(Color.BLACK, new Position(0, 3));
        board[7][3] = new Queen(Color.WHITE, new Position(7, 3));
        // Setup other pieces...
    }
}
```

Player Class

```
class Player {
    private String name;
    private Color color;

    public Player(String name, Color color) {
        this.name = name;
        this.color = color;
    }
}
```

Game Class

```
class Game {
   private Board board;
   private Player[] players;
   private Player currentTurn;

   public Game(Player player1, Player player2) {
      this.board = new Board();
      this.players = new Player[]{player1, player2};
      this.currentTurn = player1;
   }

   public void start() {
      System.out.println("Game started between " + players[0].name + " and " + players[1].name);
   }
}
```

Running the Game

```
public class ChessGame {
   public static void main(String[] args) {
      Player p1 = new Player("Alice", Color.WHITE);
      Player p2 = new Player("Bob", Color.BLACK);

      Game game = new Game(p1, p2);
      game.start();
   }
}
```

Summary

- Designed HLD & LLD with class and use case diagrams.
- Implemented core Java classes with OOP principles.
- Next Steps: Add checkmate logic, piece moves, and UI for full implementation.

This design ensures a scalable and maintainable chess game.