|  |  |  |  |
| --- | --- | --- | --- |
| **Name:** | Ali Rafay | **Registration No:** | SP24-BCS-014 |
| **Name:** | M. Bilal Sarwar | **Registration No:** | SP24-BCS-078 |
| **Name:** | M. Zayn-Ul-Hassan | **Registration No:** | SP24-BCS-099 |
| **Name:** | Umer Rashid | **Registration No:** | SP24-BCS-096 |
| **Course Title:** | Object Oriented Programming | **Instructor’s Name:** | Sir Zaheer Ahmed Gondal |

**Chess Engine**

**Documentation**

**Position Class:**

The Position class represents a specific location on the chessboard. Each position is defined by an x-coordinate (column) and a y-coordinate (row). The coordinates use a 0-based system, where (0, 0) represents the lower-left corner of the board and (7, 7) represents the top-right corner. This class is central to tracking the locations of chess pieces and interacting with the board during gameplay.

**Attributes:**

1. **private int x**:  
   Represents the column index of the position, where 0 corresponds to column "a" and 7 corresponds to column "h".
2. **private int y**:  
   Represents the row index of the position, where 0 corresponds to row "1" and 7 corresponds to row "8".

**Constructors:**

1. **Position(int x, int y)**:  
   Initializes a Position with specific x and y coordinates.
2. **Position(Position position)**:  
   Creates a new Position object by copying the x and y coordinates from an existing Position.
3. **Position(MouseEvent event)**:  
   Converts mouse click coordinates (in pixels) into a chessboard position based on a grid-square size of 50x50 pixels. The y-coordinate is adjusted to account for the inverted direction (top to bottom vs. bottom to top).
4. **Position(String position)**:  
   Parses a chess notation string like "a1" or "h8" to determine the corresponding x and y coordinates. Validates that the string is in the correct format and within board boundaries.

**Methods:**

1. **getX()**:  
   Returns the x-coordinate (column index) of the position.
2. **getY()**:  
   Returns the y-coordinate (row index) of the position.
3. **insideBoard()**:  
   Returns true if the position lies within the chessboard boundaries (0 ≤ x, y < 8), otherwise returns false.
4. **add(Position position)**:  
   Adds the x and y coordinates of the current position with another position, returning a new Position object with the summed coordinates.
5. **toString()**:  
   Converts the position to chess notation (e.g., "a1", "b2"). This is done by converting the x-coordinate into its corresponding column letter and the y-coordinate into its corresponding row number.
6. **hashCode()**:  
   Generates a hash code for the position based on its x and y coordinates. This is useful for storing positions in hash-based collections but is currently not used.
7. **equals(Object obj)**:  
   Compares two Position objects based on their hash codes, determining if they represent the same location on the board.

**Player Class:**

The Player class represents a player in a chess game, managing details such as the player's color, direction of movement, turn status, and castling rights. It provides utility methods to manipulate and retrieve this information, enabling effective gameplay control and logic.

**Attributes:**

1. **private boolean white**:  
   A boolean value indicating the player's color. true if the player is white, false if the player is black.
2. **private boolean hasTurn**:  
   A boolean value indicating whether it is currently the player's turn. Defaults to false.
3. **private int dir**:  
   The direction in which the player’s pieces move. 1 indicates upward movement (white pieces), and -1 indicates downward movement (black pieces).
4. **private boolean canCastleKingSide**:  
   Indicates whether the player has the right to castle on the king’s side. Defaults to true.
5. **private boolean canCastleQueenSide**:  
   Indicates whether the player has the right to castle on the queen’s side. Defaults to true.

**Constructors:**

1. **Player(PieceColor color)**:  
   Creates a new player with the specified color (WHITE or BLACK). Initializes the white attribute based on the color and sets the direction (dir) accordingly:
   * White player: white = true and dir = 1.
   * Black player: white = false and dir = -1.

**Methods:**

1. **toggleHasTurn()**:  
   Toggles the player's turn status. If the player currently has the turn, this method sets hasTurn to false and vice versa.
2. **hasTurn()**:  
   Returns true if it is the player’s turn, otherwise false.
3. **toString()**:  
   Returns the player’s color as a string: "White" for white players, "Black" for black players.
4. **toChar()**:  
   Returns the player's color as a character: 'w' for white players, 'b' for black players.
5. **getDir()**:  
   Returns the player’s direction: 1 for white players, -1 for black players.
6. **isWhite()**:  
   Returns true if the player is white, otherwise false.
7. **setWhite(boolean isWhite)**:  
   Sets the player's color based on the provided value. Used to modify the player's color dynamically.
8. **canCastleKingSide()**:  
   Returns true if the player can castle on the king’s side, otherwise false.
9. **canCastleQueenSide()**:  
   Returns true if the player can castle on the queen’s side, otherwise false.
10. **setCastling(boolean value, boolean queenSide)**:  
    Updates the player's castling rights.
    * If queenSide is true, updates the canCastleQueenSide attribute.
    * If queenSide is false, updates the canCastleKingSide attribute.
11. **disableCastling()**:  
    Disables all castling rights for the player by setting both canCastleKingSide and canCastleQueenSide to false.
12. **getCastlingRights()**:  
    Returns the player’s castling rights as a string:
    * "k" if king-side castling is allowed.
    * "q" if queen-side castling is allowed.
    * "kq" if both castling rights are allowed.
    * An empty string "" if no castling rights exist.
    * The string is converted to uppercase for white players.

**Piece Class:**

The Piece class is an abstract representation of a chess piece, serving as a base class for all specific chess pieces (e.g., King, Queen, Pawn). It provides common functionality for pieces such as moving, determining legal moves, tracking ownership and position, and representing the piece on the board. The class facilitates interaction between the chessboard, players, and specific pieces.

**Attributes:**

1. **protected Position pos**:  
   Represents the current position of the piece on the chessboard.
2. **protected ChessBoard board**:  
   A reference to the chessboard on which the piece resides, allowing interaction with other pieces and positions.
3. **protected Player owner**:  
   The player who owns the piece (either white or black).
4. **protected String name**:  
   The name of the piece, used for identification (e.g., "King", "Queen").
5. **protected int moveCount**:  
   Tracks the number of times the piece has moved during the game. This is important for special moves like castling or pawn promotion.

**Constructors:**

1. **Piece(Position position, ChessBoard board, Player owner, String name)**:  
   Initializes a piece with a specific position, board, owner, and name. If the piece is not already at the specified position, it is automatically placed on the board.

**Methods:**

1. **Static Method:**
   * **placePiece(Player player, Position pos, ChessBoard board, char c)**:  
     Places a specific piece on the board based on the character c. For uppercase characters, the piece belongs to the white player; for lowercase, it belongs to the black player. Creates an instance of the appropriate subclass (King, Queen, etc.).
2. **Getter Methods:**
   * **getX()**: Returns the x-coordinate of the piece.
   * **getY()**: Returns the y-coordinate of the piece.
   * **getPos()**: Returns a copy of the current position of the piece.
   * **getMoveCount()**: Returns the number of moves the piece has made.
   * **getOwner()**: Returns the player who owns the piece.
   * **isWhite()**: Returns true if the piece belongs to the white player, otherwise false.
3. **Setter Methods:**
   * **setPos(Position pos)**: Sets the piece's position on the board, with validation to ensure the position is within the board's boundaries.
4. **Movement and Validation:**
   * **move(Position to)**: Moves the piece to the specified position. Throws an exception if the move is not valid.
   * **isValidMove(Position to)**: Checks whether the specified position is a valid move for the piece, based on its movement rules.
   * **addMoveCount()**: Increments the piece's move count after a successful move.
5. **Legal Moves and Threats:**
   * **getLegalMoves()**: Abstract method to calculate all legal moves for the piece, implemented by subclasses.
   * **threatening(Position position)**: Abstract method to determine if the piece is threatening a given position, implemented by subclasses.
   * **messesUpcheck(Position to)**: Checks if moving the piece to the specified position would put the king in check. Temporarily modifies the board state to simulate the move.
6. **Representation:**
   * **toString()**: Returns the name of the piece.
   * **toChar()**: Returns a character representation of the piece:
     + Uppercase for white pieces.
     + Lowercase for black pieces.
     + Specific mappings: 'K' for King, 'Q' for Queen, etc.

**Pawn Class:**

The Pawn class represents a pawn piece in chess and inherits from the Piece class. It encapsulates the unique behavior and movement rules of pawns, such as moving one step forward, capturing diagonally, making an initial two-step move, and performing en passant captures.

**Attributes:**

1. **private boolean hasMoved**:  
   Tracks whether the pawn has moved from its initial position. This is used to determine if the pawn can make a two-step forward move.
2. **private boolean hasMadeAnPassant**:  
   Indicates whether the pawn has performed an en passant capture during the game.

**Constructor:**

1. **Pawn(Position position, ChessBoard board, Player owner)**:  
   Creates a new pawn at a specified position on the board, assigns it to the given owner (player), and sets its initial state. If the pawn is not in its starting row (rank 2 for white, rank 7 for black), the hasMoved attribute is set to true.

**Methods:**

1. **boolean isValidMove(Position to)**:  
   Checks whether the move to the specified position is valid.
   * If the x-coordinate changes, it checks if the move is a valid capture.
   * Otherwise, it checks if the move is a valid straight-forward move.
2. **Collection<Position> getLegalMoves()**:  
   Returns all possible legal moves for the pawn, considering its unique movement rules:
   * One step forward if the destination square is empty.
   * Two steps forward on the first move if both squares are empty.
   * Capturing one step diagonally forward if an opponent's piece is present or if the move is an en passant.
3. **void move(Position to)**:  
   Moves the pawn to the specified position if the move is legal.
   * Checks if the move is an en passant and sets hasMadeAnPassant accordingly.
   * Updates the pawn's position and sets hasMoved to true.
4. **void setHasMadeAnPassant(boolean hasMadeAnPassant)**:  
   Sets whether the pawn has made an en passant.
5. **boolean getHasMadeEnPassant()**:  
   Returns true if the pawn has made an en passant; otherwise, false.
6. **boolean threatening(Position position)**:  
   Checks if the pawn is threatening the given position by moving diagonally one step forward.
7. **boolean isValidStraightMove(Position to)**:  
   Validates if the pawn's move straight forward to the specified position is valid:
   * Ensures the x-coordinate does not change.
   * Ensures the destination square is empty.
   * Allows a two-step forward move only if the pawn hasn't moved and both squares are empty.
8. **boolean isValidCaptureMove(Position to)**:  
   Validates if the move to the specified position is a valid capture:
   * Ensures the destination is one step diagonally forward.
   * Allows capturing opponent pieces or performing en passant.
9. **boolean moveIsEnPassant(Position to)**:  
   Checks whether the move to the specified position is an en passant:
   * Ensures the move is one step diagonally forward to an empty square.
   * Verifies that the captured piece (directly behind the target square) is an opponent pawn that just made a two-step forward move.

**Knight Class:**

The Knight class represents a knight piece in chess and extends the Piece class. It encapsulates the unique movement rules of the knight, which include moving in an "L" shape: two steps in one direction and one step in a perpendicular direction. The knight is also unique in its ability to jump over other pieces.

**Attributes:**

1. **private final static int[] dX**:  
   Represents the possible changes in the x-coordinate for the knight's moves. Each index corresponds to a move direction.
2. **private final static int[] dY**:  
   Represents the possible changes in the y-coordinate for the knight's moves. Each index corresponds to the respective move direction in dX.

**Constructor:**

1. **Knight(Position position, ChessBoard board, Player owner)**:  
   Initializes a knight at a specified position, on a given board, and assigns it to the specified player. Calls the Piece superclass constructor to set the initial state.

**Methods:**

1. **boolean isValidMove(Position to)**:  
   Checks if the move to the specified position is valid by determining if the position is present in the knight's legal moves.
2. **Collection<Position> getLegalMoves()**:  
   Calculates and returns all possible legal moves for the knight:
   * Iterates through the possible moves defined by the dX and dY arrays.
   * For each potential move:
     + Creates a new position by adding the respective dX and dY values to the current position.
     + Checks if moving to the position puts the king in check using messesUpcheck.
     + Validates the position using validOnBoard.
   * Adds valid positions to the legalMoves collection.
3. **boolean threatening(Position position)**:  
   Checks if the knight is threatening the specified position:
   * Iterates through all possible moves.
   * Compares each resulting position with the specified position.
   * Returns true if the position matches any valid move.
4. **boolean validOnBoard(Position to)**:  
   Determines if the specified position is valid for the knight to move to:
   * Ensures the position is inside the board boundaries.
   * Allows the move if the destination is empty or occupied by an opponent's piece.
   * Disallows the move if the destination is occupied by a piece belonging to the same player.

**King Class:**

The King class represents the king piece in chess and extends the Piece class. It encapsulates the unique movement and game-specific rules of the king, including single-step movement in all directions, castling, and avoiding moves that put the king in check.

**Attributes:**

The King class doesn't define additional attributes beyond those inherited from Piece. However, its methods implement king-specific behavior, such as checking for threats and castling.

**Constructor:**

1. **King(Position position, ChessBoard board, Player owner)**:  
   Initializes the king at a specified position, on a given board, and assigns it to the specified player. The superclass constructor is called to set common piece attributes.

**Methods:**

1. **boolean inCheck()**:  
   Determines if the king is currently in check by calling positionIsInCheck for its current position.
2. **private boolean positionIsInCheck(Position position)**:  
   Checks if the specified position is under threat from any opponent's piece. Iterates through all pieces on the board, checking if each one threatens the position.
3. **private boolean canCastleQueen()**:  
   Validates if the king can perform a queenside castle:
   * Ensures the player still has queenside castling rights.
   * Checks that the king is not in check and does not move through or land in a square under attack.
   * Confirms that there are no pieces between the king and the queenside rook.
4. **private boolean canCastleKing()**:  
   Validates if the king can perform a kingside castle:
   * Similar to canCastleQueen(), but for the kingside rook.
   * Checks the relevant squares between the king and rook.
5. **private Collection<Position> getCastlingMoves()**:  
   Returns a collection of castling moves:
   * Adds the kingside and/or queenside castling destination to the collection if the respective castling is valid.
6. **Collection<Position> getLegalMoves()**:  
   Overrides the Piece method to calculate all legal moves for the king:
   * Includes one-step moves in all directions (up, down, left, right, and diagonals).
   * Filters out moves that place the king in check or are outside the board.
   * Includes valid castling moves by calling getCastlingMoves.
7. **protected boolean threatening(Position position)**:  
   Determines if the king threatens the specified position by checking all adjacent squares.
8. **void move(Position to)**:  
   Overrides the Piece method to handle king-specific movement:
   * Throws an exception if the move is illegal.
   * Handles castling by moving the respective rook to the correct position.
   * Moves the king to the target position.

**LinearPiece Class:**

The LinearPiece class is an abstract subclass of Piece that defines the behavior of chess pieces capable of moving in straight lines for any length. These movements include vertical, horizontal, and diagonal directions, depending on the piece type. Specific subclasses like Rook, Bishop, and Queen inherit from LinearPiece and define their unique movement directions.

**Attributes:**

1. **private Collection<Position> legalDirections**:  
   A collection of directional vectors that represent the possible directions this piece can move in. For example:
   * A rook would have horizontal and vertical directions.
   * A bishop would have diagonal directions.
   * A queen would have all eight linear directions.

**Constructor:**

1. **LinearPiece(Position position, ChessBoard board, Player owner, String name)**:  
   Initializes a linear piece with its position, the chessboard it belongs to, its owner (player), and its name. This constructor calls the superclass (Piece) constructor to set these values.

**Methods:**

1. **getLegalMoves()**:  
   Overrides the getLegalMoves() method from the Piece class. This method calculates all possible moves for the piece based on its movement directions (legalDirections).
   * Iterates over each direction vector in legalDirections.
   * Continuously adds the direction vector to the current position to compute all valid positions in that direction.
   * Stops when it encounters:
     + A position outside the board.
     + A position occupied by a piece of the same owner.
     + A position where moving the piece would put its king in check.
   * Adds valid positions (unoccupied or occupied by opponent pieces) to the legalMoves collection.
2. **threatening(Position position)**:  
   Checks whether this piece is threatening a specific position on the board:
   * Iterates over each direction vector in legalDirections.
   * Simulates movement along each direction until it either:
     + Reaches the specified position (returns true if it does).
     + Encounters an obstacle (another piece or the edge of the board).
   * Returns false if the specified position is not threatened by the piece.
3. **setDirections(List<Position> directions)**:  
   Sets the possible directions this piece can move in.
   * Converts the input directions list into a new ArrayList and assigns it to legalDirections.
   * This method is typically used by subclasses (Rook, Bishop, Queen) to define their unique movement directions.

**Bishop Class:**

The Bishop class represents the bishop piece in chess and extends the LinearPiece class. It specializes the behavior of the LinearPiece by defining movement directions specific to the bishop: diagonal lines in all four diagonal directions.

**Attributes:**

The Bishop class does not introduce new attributes. It inherits all attributes from the LinearPiece class, which itself inherits from the Piece class. Key inherited attributes include:

* **pos**: The current position of the bishop.
* **board**: The chessboard the bishop is on.
* **owner**: The player who owns the bishop.

**Constructor:**

1. **Bishop(Position position, ChessBoard board, Player owner)**:  
   Creates a new bishop at a specific position, on a given board, and assigns it to the specified player.
   * Calls the constructor of the LinearPiece superclass to initialize the bishop with common attributes.
   * Defines the specific movement directions for the bishop:
     + new Position(1, 1): Up-right diagonal.
     + new Position(1, -1): Down-right diagonal.
     + new Position(-1, 1): Up-left diagonal.
     + new Position(-1, -1): Down-left diagonal.
   * These directions are passed to the setDirections method of the LinearPiece class, which configures the bishop's movement capabilities.

**Methods:**

The Bishop class does not define new methods. Instead, it inherits and uses methods from the LinearPiece and Piece classes to handle its functionality. Key inherited methods include:

1. **getLegalMoves()** (inherited from LinearPiece):  
   Calculates all legal moves for the bishop by iterating through its diagonal movement directions.
   * Moves are valid if they:
     + Remain within the board boundaries.
     + Are not blocked by other pieces along the path.
     + Do not place the bishop's king in check.
   * The bishop can capture opponent pieces but cannot move through them.
2. **threatening(Position position)** (inherited from LinearPiece):  
   Checks if the bishop is threatening a specific position by tracing along its diagonal directions.
3. **move(Position to)** (inherited from Piece):  
   Moves the bishop to the specified position if the move is legal.
4. **isValidMove(Position to)** (inherited from Piece):  
   Verifies if a move to the given position is valid.

**Rook Class:**

The Rook class represents the rook piece in chess and extends the LinearPiece class. It specializes the behavior of LinearPiece by defining movement directions specific to the rook: straight lines horizontally and vertically.

**Attributes:**

The Rook class does not introduce any new attributes. It inherits all attributes from the LinearPiece and Piece classes, including:

* **pos**: The rook's current position.
* **board**: The chessboard the rook is on.
* **owner**: The player who owns the rook.

**Constructor:**

1. **Rook(Position position, ChessBoard board, Player owner)**:  
   Creates a new rook at a specific position on the given chessboard and assigns it to the specified player.
   * Calls the constructor of the LinearPiece superclass to initialize the rook with shared attributes.
   * Configures the rook's movement directions by calling the setDirections method with the following directions:
     + new Position(0, 1): Up.
     + new Position(0, -1): Down.
     + new Position(1, 0): Right.
     + new Position(-1, 0): Left.

These directions define the rook's ability to move along straight lines horizontally and vertically.

**Methods:**

The Rook class does not define new methods. It relies on the inherited methods from LinearPiece and Piece to handle its functionality. Key methods include:

1. **getLegalMoves()** (inherited from LinearPiece):  
   Calculates all legal moves for the rook by iterating through its defined movement directions.
   * Moves are valid if:
     + They remain within the board boundaries.
     + They are not blocked by other pieces along the path.
     + They do not place the rook's king in check.
   * The rook can capture opponent pieces but cannot move through them.
2. **threatening(Position position)** (inherited from LinearPiece):  
   Checks if the rook threatens a specific position by tracing along its horizontal and vertical movement paths.
3. **move(Position to)** (inherited from Piece):  
   Moves the rook to the specified position if the move is valid.
4. **isValidMove(Position to)** (inherited from Piece):  
   Verifies if a move to the given position is valid.

**Queen Class:**

The Queen class represents the queen piece in chess and extends the LinearPiece class. It combines the movement capabilities of both the rook (horizontal and vertical movement) and the bishop (diagonal movement), allowing it to move in a straight line in any direction.

**Attributes:**

The Queen class does not introduce new attributes. It inherits all attributes from the LinearPiece and Piece classes, including:

* **pos**: The queen's current position on the chessboard.
* **board**: The chessboard the queen is part of.
* **owner**: The player who owns the queen.

**Constructor:**

1. **Queen(Position position, ChessBoard board, Player owner)**:  
   Creates a new queen at the specified position, assigns it to the specified player, and places it on the board.
   * Calls the LinearPiece superclass constructor to initialize the shared attributes and name the piece "Queen."
   * Configures the queen's movement directions using the setDirections method, enabling movement in all eight possible straight-line directions:
     + Vertical: new Position(0, 1) (up), new Position(0, -1) (down).
     + Horizontal: new Position(1, 0) (right), new Position(-1, 0) (left).
     + Diagonal: new Position(1, 1) (up-right), new Position(1, -1) (down-right), new Position(-1, 1) (up-left), new Position(-1, -1) (down-left).

**Methods:**

The Queen class does not define new methods. It uses the methods inherited from LinearPiece and Piece to handle its functionality. Key inherited methods include:

1. **getLegalMoves()** (inherited from LinearPiece):  
   Calculates all legal moves for the queen by iterating through its movement directions.
   * Moves are valid if:
     + They stay within the board boundaries.
     + They are not blocked by other pieces along the path.
     + They do not place the queen's king in check.
   * The queen can capture opponent pieces but cannot move through them.
2. **threatening(Position position)** (inherited from LinearPiece):  
   Checks if the queen is threatening a specific position by tracing along its movement paths.
3. **move(Position to)** (inherited from Piece):  
   Moves the queen to the specified position if the move is legal.
4. **isValidMove(Position to)** (inherited from Piece):  
   Verifies whether moving to the specified position is valid based on chess rules.

**PieceColor Enum:**

The PieceColor enum represents the two possible colors of players in chess: white and black. It also associates a directional value with each color, which is used to determine the direction of movement for pieces owned by the player of that color.

**Attributes:**

1. **WHITE(1)**:  
   Represents the white color. Pieces belonging to the white player move in the "up" direction on the board. The direction is represented by the value 1.
2. **BLACK(-1)**:  
   Represents the black color. Pieces belonging to the black player move in the "down" direction on the board. The direction is represented by the value -1.
3. **private final int dir**:  
   A private field storing the directional value associated with the color.
   * 1 for upward movement (white).
   * -1 for downward movement (black).

**Constructor:**

1. **private PieceColor(int dir)**:
   * A private constructor for the enum, called automatically for each constant (WHITE and BLACK) when the enum is loaded.
   * Assigns the directional value (dir) to the respective color.

**Methods:**

1. **public int getDir()**:
   * A getter method to retrieve the directional value associated with the color.
   * Returns 1 for WHITE and -1 for BLACK.

**ChessBoardIterator Class:**

The ChessBoardIterator class is an implementation of the Iterator interface that allows for the traversal of a ChessBoard object, returning each non-empty position (i.e., positions occupied by chess pieces) sequentially. It provides an easy way to iterate through the chessboard without directly accessing the board positions, making it useful for tasks like checking or processing each piece on the board.

**Attributes:**

1. **board**:
   * A reference to the ChessBoard object that the iterator is traversing.
   * It is passed into the constructor and is used to retrieve positions and check for chess pieces.
2. **x**:
   * Represents the current column on the board (from 0 to 7).
   * Initially set to 0 and updated as the iterator moves through the board.
3. **y**:
   * Represents the current row on the board (from 0 to 7).
   * Initially set to 0 and updated as the iterator moves through the board.

**Constructor:**

1. **ChessBoardIterator(ChessBoard board)**:
   * This constructor initializes the iterator with a specific ChessBoard object, allowing the iteration to start from the top-left corner of the board (0, 0).

**Methods:**

1. **hasNext()**:

This method checks if there are any remaining non-empty positions (i.e., positions with a chess piece) on the board to iterate over.

1. **next()**:

This method returns the next piece from the chessboard and advances the iterator:

**FENParser Class:**

The FENParser class is a utility class designed to handle the parsing, generation, and saving of FEN (Forsyth-Edwards Notation) strings for chess games. It includes methods for reading, writing, validating, and generating FEN strings, making it easier to manage game states in chess. The class operates with the ChessBoard, Piece, Player, and Position objects to represent and manipulate the state of the chessboard.

**Attributes:**

1. **FEN\_EXTENSION**:
   * A constant that stores the file extension for FEN files, which is "fen".
2. **DEFAULT\_STRING**:
   * A constant that provides the default FEN string for a standard chess game setup. This string represents the initial state of a chess game with all pieces in their starting positions.

**Methods:**

1. **readFENFromFile(File file)**:
   * Reads a FEN string from a given file.
2. **getBoardFromFEN(String input)**:
   * Parses a FEN string and converts it into a ChessBoard object, which represents the current state of the chess game.
3. **getBoardFromDefaultFEN()**:
   * Returns a default ChessBoard object initialized with the DEFAULT\_STRING. This method handles the default chessboard setup.
4. **legalFischerRandom(String FENString)**:
   * Validates if a FEN string represents a legal Fischer Random Chess setup (a variant of chess where pieces are randomly shuffled on the back ranks).
   * Returns true if the FEN string is valid for Fischer Random Chess, false otherwise.
5. **generateFischerRandomFEN()**:
   * Generates a FEN string for a random Fischer Random Chess setup. It shuffles the back rank pieces and checks for legal placement.
   * Returns the generated FEN string.
6. **generateFEN(ChessBoard board)**:
   * Converts a ChessBoard object to a corresponding FEN string representing the current state of the game.
   * The FEN string is generated by traversing the chessboard and representing each piece and its position, as well as the game state like castling rights and en passant.
7. **saveToFile(String fenString, File file)**:
   * Saves the provided FEN string to a specified file.
   * Creates the parent directories if they don't exist and writes the FEN string to the file using a BufferedWriter.
8. **getFileFromChooser()**:
   * Opens a file chooser dialog for the user to select a file.
   * Returns the selected file or null if no file is selected.

**IllegalFENException Exception**:

The IllegalFENException class is a custom exception used to indicate errors related to invalid FEN (Forsyth-Edwards Notation) strings in the chess engine. The class extends the Exception class, which allows it to function as a standard Java exception that can be thrown and caught in try-catch blocks.

**Constructor:**

1. **IllegalFENException(String string)**:
   * This constructor initializes the exception with a custom error message passed as a String parameter.
   * The super(string) call invokes the constructor of the Exception class, passing the error message to be displayed when the exception is thrown.

**PopUp Class:**

The PopUp class is a utility class for creating and displaying pop-up windows in a JavaFX-based application. It allows you to create a customizable window with a title, the option to add multiple nodes (such as text, buttons, or other UI components), and control whether or not the window is closable.

**Attributes:**

* **nodes**:  
  A Collection<Node> that stores the nodes (UI elements) to be displayed inside the pop-up window. Nodes can include buttons, labels, text fields, etc.
* **closeable**:  
  A boolean that determines if the pop-up window is closable by the user. If set to true, the window will have a "Close" button. If set to false, the window cannot be closed by the user and will not have a close button.
* **windowTitle**:  
  A string that stores the title of the pop-up window.

**Constructor:**

1. **PopUp(String windowTitle, boolean closeable)**:
   * This constructor initializes the PopUp object with a given title (windowTitle) and a closeable flag to control whether the pop-up should have a close button.
   * It also initializes the nodes attribute as an empty ArrayList and assigns the provided title and closable status.

**Methods:**

1. **addNode(Node node)**:
   * This method allows adding a node (a UI component) to the list of nodes to be displayed in the pop-up. This can be any type of JavaFX node (e.g., labels, buttons, images, etc.).
2. **display()**:
   * This method creates and displays the pop-up window.
   * It creates a new Stage object (which represents a window in JavaFX), sets the window to be modal (using initModality(Modality.APPLICATION\_MODAL)), and gives it the title specified when the PopUp object was created.
   * If the closeable attribute is true, it adds a close button to the window. If closeable is false, it disables the ability to close the window by consuming the close request event (e.consume()).
   * A VBox layout is used to organize the nodes vertically, and the scene is set with this layout. The window is then displayed using showAndWait(), which ensures that the user interacts with the pop-up before returning to the main application.

**ChessBoard Class:**

The ChessBoard class is a central part of the chess engine, responsible for managing the chessboard state, handling piece movements, managing the game rules, and providing game status updates. Below is an explanation of its variables and methods:

**Variables:**

* **board**:  
  A 2D array (ArrayList<ArrayList<Piece>>) representing the chessboard, where each cell contains a Piece or null if the square is empty.
* **moves**:  
  An ArrayList<String> that tracks all moves made in the game, represented as strings (e.g., "e2e4"). These moves are used to display the game history.
* **selectedPiece**:  
  A Piece representing the currently selected piece by the player, which can be moved to a new position.
* **lastMovedPiece**:  
  The last Piece that was moved. This is important for certain game mechanics like castling and en passant.
* **white**:  
  The Player object representing the white player.
* **black**:  
  The Player object representing the black player.
* **turn**:  
  The Player whose turn it is to make a move.
* **halfMoves**:  
  An integer tracking the number of half-moves (plies) made, where one ply is a single move made by either player.
* **fullMoves**:  
  An integer tracking the number of full moves in the game (a full move consists of a move from both players).
* **gameFinished**:  
  A boolean flag indicating whether the game has ended, either by checkmate, stalemate, or draw.
* **gameMessage**:  
  A string that stores a message describing the game's current state, such as "Checkmate", "Stalemate", or "Draw".
* **upgradablePawn**:  
  A Pawn object representing a pawn that has reached the opponent’s back rank and can be promoted to another piece.

**Constructors:**

* **ChessBoard()**:  
  A default constructor that initializes a chessboard with two new players (white and black).
* **ChessBoard(Player white, Player black)**:  
  Initializes a chessboard with the specified white and black players. The board is set up as an 8x8 grid of null values, and the white player starts the game.

**Methods:**

* **getPosition(Position position)**:  
  Returns the Piece located at the specified position on the board. If the position is out of bounds, it returns null.
* **setPosition(Position position, Piece piece)**:  
  Sets the Piece at the given position on the board. It checks whether the piece already occupies the position and updates accordingly.
* **move(Piece piece, Position to, boolean isCastle)**:  
  Moves a piece to a new position. It handles special cases like castling and en passant. It updates the piece's position, adds the move to the history, and switches turns. If the move is a pawn reaching the last rank, it triggers the pawn promotion logic.
* **move(Piece piece, Position to)**:  
  A simplified version of the move() method without the castling flag, used for normal piece moves.
* **inCheck(Player player)**:  
  Returns true if the specified player is in check (i.e., their king is under threat of capture).
* **getMoves()**:  
  Returns a string representation of all moves made in the game, formatted in a readable manner.
* **iterator()**:  
  Returns an iterator over the pieces on the board, allowing for iteration through all pieces using a for-each loop.
* **getLastPieceMoved()**:  
  Returns the last piece that was moved.
* **toString()**:  
  Returns a string representation of the chessboard in FEN (Forsyth-Edwards Notation) format.
* **setTurn(PieceColor color)**:  
  Sets the current player's turn based on the provided color (WHITE or BLACK).
* **setLastPieceMoved(Piece piece)**:  
  Sets the last moved piece.
* **disableCastling()**:  
  Disables castling for both players, typically invoked when the king or rook has moved.
* **getCastlingRights()**:  
  Returns a string representing the castling rights of both players (e.g., "KQkq" indicates both players can castle on both sides).
* **getFEN()**:  
  Returns the current board state as a FEN string, representing the game state in a standardized notation.
* **getPlayerTurn()**:  
  Returns the player whose turn it is to move.
* **setHalfMoves(int moves)**:  
  Sets the number of half-moves (plies) that have been played so far.
* **getHalfMoves()**:  
  Returns the current number of half-moves.
* **setFullMoves(int moves)**:  
  Sets the number of full moves (two plies) made so far.
* **getFullMoves()**:  
  Returns the number of full moves made in the game.
* **getGameFinished()**:  
  Returns true if the game is over.
* **getGameMessage()**:  
  Returns a string describing the game status, such as "Checkmate", "Stalemate", or "Draw".
* **setSelectedPiece(Piece piece)**:  
  Sets the currently selected piece for movement.
* **getSelectedPiece()**:  
  Returns the currently selected piece.
* **getUpgradablePawn()**:  
  Returns the pawn that is eligible for promotion (i.e., a pawn that has reached the last rank).
* **handleCastlingDisabling(Piece piece, Position originalPos)**:  
  Disables castling for the player if the king or rook has moved.
* **handleEnPassantMove(Piece piece)**:  
  Handles an en passant move, capturing the opposing pawn if necessary.
* **setPromotePawn(Pawn piece)**:  
  Sets a pawn that has reached the last rank for promotion.
* **promotePawn(Pawn piece, Piece upgrade)**:  
  Promotes a pawn to the given piece (e.g., Queen, Rook, etc.).
* **handleHalfMove(Piece piece)**:  
  Updates the half-move counter. If it's black’s turn, the full move counter is incremented.
* **handleFullMove(Piece piece, boolean pieceWasCaptured)**:  
  Handles the full move counter, resetting it if a pawn or a piece was captured.
* **checkGameFinished()**:  
  Checks if the game has ended and updates the game message accordingly.
* **inDraw()**:  
  Checks if the game is in a draw situation based on the number of half-moves.
* **inStalemate(Player player)**:  
  Checks if the specified player is in stalemate.
* **inCheckmate(Player player)**:  
  Checks if the specified player is in checkmate.
* **addMove(String string)**:  
  Adds a move to the list of moves made in the game.

**StockFish Class:**

The StockFish class interacts with the Stockfish chess engine, which is a powerful open-source chess engine used for analyzing and playing chess. The class provides methods to start the engine, set a position using FEN (Forsyth-Edwards Notation), retrieve the best move based on a given depth, and stop the engine.

**Variables:**

* **process**:  
  This is an instance of the Process class that represents the running Stockfish engine. The process is started through the ProcessBuilder in the startEngine() method.
* **writer**:  
  A BufferedWriter used to send commands to the Stockfish engine. It writes to the engine's input stream.
* **reader**:  
  A BufferedReader used to read the responses from the Stockfish engine, including the best move suggested by the engine.

**Methods:**

* **startEngine(String path)**:  
  Starts the Stockfish engine by invoking it from the specified path.
  + It uses the ProcessBuilder class to start a new process running Stockfish.
  + It redirects the error stream and connects input/output streams to communicate with the engine.
  + Path parameter: The path to the Stockfish executable file.
  + Returnstrue if the engine starts successfully, otherwise false.
* **setPositionUsingFEN(String fen)**:  
  Sets the position of the chess game using the FEN string. It sends the position fen <FEN> command to the Stockfish engine.
* **sendCommand(String command)**:  
  Sends a command to the Stockfish engine via the BufferedWriter.
  + The command is written to the input stream and immediately flushed to ensure it is sent to the engine.:
  + Command Paramter: The command to send to Stockfish.
* **getBestMove(int depth)**:  
  Requests Stockfish to calculate the best move for the current position at a specified depth. It sends the go depth <depth> command to Stockfish.
  + This method returns the best move that Stockfish computes.
  + Depth Paramter: The depth (number of moves ahead) for which Stockfish should calculate the best move.
  + Returns a string representing the best move, or the entire output from Stockfish if there are issues.
* **readOutput()**:  
  Reads the output from the Stockfish engine. It looks for a line starting with bestmove to identify the best move in the response.
  + Returns a string containing the best move and other information from Stockfish.
* **stopEngine()**:  
  Stops the Stockfish engine by sending the quit command. It waits for the process to terminate after sending the stop command.

**StockFishAPI Class:**

The StockFishAPI class is designed to serve as an interface between a chess game application and the Stockfish chess engine. It allows the application to interact with Stockfish, set a position on the board using a FEN string, retrieve the best move based on the engine's analysis, and stop the engine when it's no longer needed.

**Variables:**

* **api**:  
  An instance of the StockFish class that manages the interaction with the Stockfish engine. It allows for communication with Stockfish, such as setting a position, retrieving the best move, and stopping the engine.
* **depth**:  
  A static integer variable representing the depth (number of moves ahead) for which Stockfish will calculate the best move. The higher the depth, the more accurate (but slower) the move calculation.

**Methods:**

* **StockFishAPI()**:  
  This constructor initializes the StockFish instance (api) and starts the Stockfish engine based on the operating system (Windows, Mac, or unknown OS).
  + It detects the operating system using System.getProperty("os.name"), then chooses the appropriate Stockfish executable path based on the OS:
    - For **Windows**, it uses the stockfish-windows-x86-64-avx2 version.
    - For **Mac**, it uses the stockfish-macos-x86-64 version.
    - If the OS is unknown, it prints a message and does not start the engine.
* **getAIMove(String fen)**:  
  This method interacts with Stockfish to get the best move for a given position.
  + It first sets the position on the board using the setPositionUsingFEN() method of the StockFish class.
  + Then, it requests the best move by calling getBestMove(depth) and parsing the result to extract the best move. The move is extracted by splitting the response string at "bestmove " and taking the first word after it.
* **stopEngine()**:  
  This method stops the Stockfish engine by calling the stopEngine() method of the StockFish class, which sends the quit command to Stockfish and terminates the process.

**ChessGameController:**

The ChessGameController class is responsible for managing the chess game scene. It handles game setup, gameplay logic, UI updates, and interaction with the Stockfish API for AI-generated moves.

**Attributes**:

* isFirstTime: Flag to track if it's the first time loading the game.
* moves: TextArea to display the move history.
* grid and backgroundBoard: GridPane objects for the chessboard and its background.
* player1time and player2time: Labels to show the players' timers.
* chessboard: The ChessBoard object representing the current state of the game.
* FENString: Stores the FEN string to initialize the game state.
* hasShownGameOver: Flag to track if the game over screen has been shown.
* gridBackgroundColor: Stores the background color for each grid tile.
* allPiecesImg: Holds the image of all chess pieces.
* imgColIdx: Maps piece characters to column indices for the piece images.

**Constructor**:

* Initializes the game with either a default or custom FEN string.

**Methods:**

* **initialize()**: Initializes the game grid and calls the method to restart the game.
* **toGamePanel()**: Switches to the game panel screen.
* **handleRestartGame()**: Resets the board based on the FEN string. If a custom FEN is available, it is used; otherwise, the default FEN is used. It starts a new game, including handling AI moves if it’s not the player's turn.
* **handleGridPaneClicked()**: Handles the click event on the chessboard. If no piece is selected, it selects the piece; otherwise, it attempts to move the selected piece.
* **drawBoard()**: Updates the chessboard UI by placing pieces, regenerating the background, coloring tiles, and updating the moves list.
* **toTitleScreen()**: Switches to the title screen.
* **exportGame()**: Opens a pop-up window to export the current game’s FEN string and save it to a file.
* **getCurrentPlayerFullName()**: Determines the current player (White or Black) based on the FEN string.
* **positionPressed()**: Handles piece selection and movement. If a piece is selected and the move is valid, the piece is moved; if not, it deselects the piece.
* **checkPawnPromotion()**: Checks if a pawn has reached the other side of the board and should be promoted.
* **promotePawn()**: Displays a pop-up to allow the user to promote a pawn to a different piece (Queen, Rook, Bishop, Knight).
* **cropPieceImage()**: Crops the image of a specific piece from the combined image and returns an ImageView object for displaying the piece.
* **placePieces()**: Places all pieces on the board by clearing the board and adding each piece at the correct position.
* **regenerateBackgroundBoard()**: Regenerates the background grid of the board.
* **colorBackgrounds()**: Colors the background of each tile based on the chessboard’s layout.
* **updateMoves()**: Updates the move history displayed in the moves TextArea.
* **getAIMove()**: Fetches the best move for the AI using the Stockfish API by passing the current FEN string.
* **applyAIMove()**: Applies the AI’s move to the board, including handling special moves like castling and promotion.
* **handlePromotion()**: Promotes a pawn to a chosen piece (Queen, Rook, Bishop, Knight).
* **setLegalMovesBackground()**: Highlights legal moves for a selected piece.
* **showGameOver()**: Displays a pop-up when the game is over, showing the result (Checkmate, Stalemate, Draw).

This controller orchestrates the game’s flow, manages UI updates, processes player and AI moves, and handles game state changes (e.g., checkmate, stalemate). It leverages the StockFishAPI to allow for AI play and integrates with the ChessBoard model to represent the state of the game.

**UpdatePawnController Class:**

The UpgradePawnController class handles the promotion of a pawn in the chess game. It provides an interface for the user to choose the piece they want the pawn to promote to, which can be a queen, rook, bishop, or knight.

**Attributes:**

* **queen**: An ImageView representing the queen image.
* **rook**: An ImageView representing the rook image.
* **bishop**: An ImageView representing the bishop image.
* **knight**: An ImageView representing the knight image.
* **upgradeChoice**: A static string variable that stores the player's selection of the piece they want the pawn to be promoted to (queen, rook, bishop, or knight).

**Methods:**

1. **initialize(URL location, ResourceBundle resources)**: This method is automatically called after the FXML is loaded. It sets up mouse click event handlers for each ImageView (queen, rook, bishop, knight). When an image is clicked, the choose method is called with the appropriate piece type as a string.
2. **choose(String pieceType)**: This method is invoked when a piece image is clicked. It sets the static variable upgradeChoice to the piece type selected by the user. After the selection, it closes the current window (using the Stage object) to complete the promotion process.
3. **getUpgradeChoice()**: A static method that returns the value stored in the upgradeChoice variable. This value indicates which piece the user chose for the pawn promotion.

**SceneSwitcher Abstract Class:**

The SceneSwitcher class is an abstract class used for handling scene transitions in a JavaFX-based chess application. It provides a method to insert a new scene or pane into a specified AnchorPane (a container for UI components) within the main application window.

**Attributes:**

* **baseAnchor**: An AnchorPane that serves as the root container for the scene being switched. This is where new scenes or panes are inserted.

**Methods:**

* **insertPane(String fxmlFileName, AnchorPane parent, SceneSwitcher controller)**:
  + **Purpose**: This method loads a new FXML file (a UI layout file) and inserts it into the parent AnchorPane. It clears the existing children in the parent before adding the new scene.
  + **Parameters**:
    - fxmlFileName: The name of the FXML file to load.
    - parent: The AnchorPane into which the new scene is inserted.
    - controller: The controller associated with the FXML file (although it's commented out in the current implementation).
  + **Implementation**:
    - It clears the existing content of parent (parent.getChildren().clear()).
    - It loads the specified FXML file using FXMLLoader.
    - The loaded Parent (the scene layout) is added to the parent AnchorPane.

**TitleScreenController Class:**

The TitleScreenController class is a JavaFX controller for the title screen of the chess application. It manages user interactions on the title screen, such as navigating to the login, registration, or guest play screens, as well as exiting the application.

**Attributes:**

* **loginButton**: A button that, when clicked, triggers the transition to the login screen.
* **registerButton**: A button that, when clicked, triggers the transition to the registration screen.
* **playAsAGuestButton**: A button that, when clicked, allows the user to play as a guest. (Although this button is present, its functionality is not fully defined in the current class).

**Methods:**

* **initialize(URL url, ResourceBundle resourceBundle)**:

This method is required by the Initializable interface but is empty in this class, as no specific initialization is needed for the title screen.

* **loginScreen()**:

This method is called when the "Login" button is clicked. It uses the insertPane method (inherited from SceneSwitcher) to replace the current scene with the login screen (LoginScreen.fxml), and sets the controller for the login screen (LoginScreenController).

* **registerScreen()**:

This method is called when the "Register" button is clicked. It uses the insertPane method to replace the current scene with the registration screen (RegistrationScreen.fxml), and sets the controller for the registration screen (RegistrationScreenController).

* **exit()**:

This method is called when the "Exit" button is clicked. It closes the application by calling System.exit(0).

**RegistrationScreenController:**

The RegistrationScreenController class manages the user interface for the registration screen in the chess application. It handles user input for registering a new username and password, ensuring that the username is unique and not already taken, and saving the credentials to a file.

**Attributes:**

* **username**: A string that stores the user's input for the username.
* **password**: A string that stores the user's input for the password.
* **usernameTakenText**: A label that is shown when the entered username is already taken.
* **registerUsername**: A text field for the user to enter their desired username.
* **registerPassword**: A text field for the user to enter their desired password.
* **emptyFieldsText**: A label that is shown if the user tries to register with empty fields.

**Methods:**

1. **initialize(URL url, ResourceBundle resourceBundle)**:
   * This method initializes the screen, setting both usernameTakenText and emptyFieldsText labels to be invisible when the screen is first loaded.
2. **register()**:
   * This method is triggered when the user submits the registration form.
   * It first checks if the username and password fields are empty. If either field is empty, it makes the emptyFieldsText label visible and stops the registration process.
   * It then checks if the provided username already exists in the file userdata.txt by scanning the file. If the username is found, it displays the usernameTakenText label, clears the input fields, and exits the method.
   * If the username is not found, it appends the new username and password to the userdata.txt file and then redirects the user to the login screen using the insertPane method, loading the LoginScreen.fxml.
3. **loginScreen()** (redundant):
   * This method is not necessary because the insertPane method is already used in the register() method for navigation. It’s supposed to redirect to the login screen.
4. **toTitleScreen()** (redundant):
   * Similar to loginScreen(), this method is redundant. It serves to navigate to the title screen but is not necessary because the title screen is managed elsewhere.

**LoginScreenController Class:**

The LoginScreenController class handles the user interface for the login screen of the chess application. It allows the user to log in with a username and password, validates the credentials against stored data, and handles error messages for incorrect or empty fields.

**Attributes:**

* **username**: Stores the entered username for login.
* **password**: Stores the entered password for login.
* **wrongPasswordText**: A label that is shown if the entered password is incorrect.
* **emptyFieldsText**: A label that is shown if either the username or password field is empty.
* **loginUsername**: A text field for the user to enter their username.
* **loginPassword**: A text field for the user to enter their password.

**Methods:**

1. **initialize(URL url, ResourceBundle resourceBundle)**:
   * Initializes the screen and sets the error labels (wrongPasswordText and emptyFieldsText) to be invisible.
2. **login()**:
   * This method is triggered when the user submits the login form.
   * It first checks if the username and password fields are empty. If either is empty, the emptyFieldsText label is made visible, and the method exits.
   * It then checks the userdata.txt file for the entered username and password using a Scanner. If a match is found, it closes the scanner and navigates to the PlayerScreen.fxml using the insertPane method.
   * If no match is found, the wrongPasswordText label is made visible, and the password field is cleared.
3. **registerScreen()** (redundant):
   * This method is unnecessary since the insertPane method is already used elsewhere in the application for navigation. It redirects to the RegistrationScreen.fxml, but this logic is already handled in the registration workflow.
4. **toTitleScreen()** (redundant):
   * Similar to registerScreen(), this method is redundant and navigates to the TitleScreen.fxml. The title screen navigation is managed elsewhere, and this method is not needed.

**PlayerScreenController Class:**

The PlayerScreenController class manages the user interface for the player selection screen in the chess application. It handles navigation to various other scenes, including the game panel, difficulty level screen, and title screen, depending on whether the user selects a single-player or multi-player mode.

**Attributes:**

* **isSinglePlayer**: A static boolean that tracks whether the game mode is single-player or multi-player. It's set to true when the player chooses the single-player mode and false for multi-player mode.

**Methods:**

1. **initialize(URL url, ResourceBundle resourceBundle)**:
   * This method is called when the scene is initialized but doesn't perform any actions in this case.
2. **toGamePanel()**:
   * Navigates to the game panel screen (GamePanel.fxml), setting isSinglePlayer to false to indicate multi-player mode.
3. **toDifficultyLevelScreen()**:
   * Navigates to the difficulty level screen (DifficultyLevelScreen.fxml), setting isSinglePlayer to true to indicate single-player mode.
4. **toTitleScreen()**:
   * Navigates back to the title screen (TitleScreen.fxml), allowing the user to return to the main menu.
5. **exit()**:
   * Closes the application by calling System.exit(0).

**DifficultyLevelScreenController Class:**

The DifficultyLevelScreenController class manages the difficulty level screen of the chess application, providing options to the user to select different difficulty levels for the AI opponent. It uses the StockFishAPI to set the AI's difficulty level, which is determined by the number of moves it considers before making a move.

**Attributes:**

* No additional instance attributes are defined in this class.

**Methods:**

1. **initialize(URL url, ResourceBundle resourceBundle)**:
   * This method is required as part of the Initializable interface, but it is not used in this class. It ensures the class is correctly initialized when the scene is loaded.
2. **easyMode()**:
   * Sets the difficulty level to "easy" by adjusting the StockFishAPI.depth to 1. This means the AI will look at 1 future move before selecting its next move.
   * Then, it navigates to the game panel (GamePanel.fxml), where the game will be played.
3. **mediumMode()**:
   * Sets the difficulty level to "medium" by adjusting the StockFishAPI.depth to 3, meaning the AI will consider 3 future moves.
   * It then switches to the game panel.
4. **hardMode()**:
   * Sets the difficulty level to "hard" by adjusting the StockFishAPI.depth to 6, meaning the AI will look at 6 future moves.
   * It then navigates to the game panel.
5. **toPlayersScreen()**:
   * Navigates back to the players screen (PlayersScreen.fxml) when the user presses the back button.

**GamePanelController Class:**

The GamePanelController class manages the game panel screen of the chess application, handling user interactions such as starting a new game, loading a game from a FEN string, and navigating between different scenes. It allows the user to either play a new game or load a previously saved game from a FEN string.

**Attributes:**

1. **customFEN**: A static string variable that holds the FEN string for a previously saved game. This is used to load the game state.
2. **UI Elements**: Various UI elements for handling the game state and user interactions, including:
   * **loadGameText**, **loadGameButton**, **loadGameCross**, **loadGameField**, **loadGameBox**: UI elements related to the game loading functionality.

**Methods:**

1. **initialize(URL location, ResourceBundle resources)**:
   * Initializes the UI elements by making the load game box and related components invisible initially.
2. **playChessDefault()**:
   * Starts a new game using the default FEN string by navigating to the ChessBoard.fxml scene.
3. **loadGameFromString()**:
   * Makes the load game UI components visible when the user chooses to load a game, allowing them to enter a FEN string.
4. **playChess()**:
   * Retrieves the FEN string entered by the user and stores it in the customFEN variable.
   * Navigates to the chessboard scene and loads the game based on the provided FEN string.
5. **exitGame()**:
   * Exits the application when the user presses the exit button.
6. **toPlayersScreen()**:
   * Navigates back to the player selection screen. If the user selected single-player mode, it goes to the DifficultyLevelScreen.fxml, otherwise, it goes to the PlayersScreen.fxml.
7. **cross()**:
   * Closes the load game input box by making the relevant UI components invisible when the user presses the cross button.