Laboratoire 8 - Chess

Leonard Cseres, Aladin Iseni

9 janvier 2025

Table des matières

1 Introduction

L'objectif de ce laboratoire est de développer un jeu d'échecs fonctionnel respectant les règles de base. Le projet inclut les fonctionnalités suivantes: déplacements des pièces, coups spéciaux (roque, prise en passant, promotion des pions) et gestion des états de jeu (par exemple, échec). Les objectifs bonus consistent à implémenter la détection de l'échec et mat, du pat et l'impossibilité de mater.

Pour simplifier le développement, les éléments suivants nous ont été fournis:

- **Enums:** `PieceType` pour les types de pièces et `PlayerColor` pour les couleurs des joueurs.
- **Interfaces:** `ChessController` et `ChessView` pour la gestion du jeu et de l'interface utilisateur.
- **Vues pré-construites:** Une vue graphique (`GUIView`) et une vue en mode texte (`ConsoleView`).

L'implémentation se concentre sur un nouveau package `engine` qui encapsule la logique du jeu tout en exploitant les interfaces fournies pour l'interaction.

Le rendu est composé de 2 versions:

- Sans modifications au package `chess` fourni
- Avec modifications au package `chess` fourni (c.f. Extensions)

---

2 Conception et Architecture

Notre approche respecte les principes de conception orientée objet, en garantissant l'encapsulation, la réutilisabilité et la modularité. Le package `engine` contient les classes et la logique pour la gestion du jeu, le suivi de l'état de l'échiquier et la génération des mouvements.

2.1 Structure

Comme mentionné précédemment, notre implémentation se situe dans le package `engine`.

- `app` ..... point d'entrée de l'application
- `chess` ..... *code founi*
- `engine` ..... notre implémentation
  - `board` ..... logique de l'échiquier
  - `generator` ..... génération de mouvements possibles pour les pièces
  - `move` ..... gestion des différents mouvements
  - `piece` ..... logique des pièces d'échec

2.2 Composants Clés

Composant	Description
`ChessEngine`	Gère le déroulement du jeu et communique avec le contrôleur de l'échiquier.
`ChessBoardController`	Expose l'échiquier en contrôlant la vue (`ChessView`).
`ChessBoard`	Représente l'échiquier, suit les pièces et valide les états du jeu.
`ChessBoardInitializer`	Met en place l'échiquier.
`ChessBoardReader` / `ChessBoardWriter`	Interface de lecture/écriture de l'échiquier.
`ChessPiece`	Classe abstraite définissant le comportement commun à toutes les pièces, étendue par des sous-classes spécifiques.
`MoveGenerator`	Classe abstraite responsable de la génération des mouvements possibles pour les pièces.
`ChessMove`	Représente un type de mouvement aux échecs.

2.3 Détails de Conception

Séparation `ChessBoard` et `ChessBoardContoller`

Nous avons découplé la logique de l'échiquier avec la mise à jour de la vue (`ChessView`) afin de pouvoir cloner l'échiquier sans être lié à la vue.

Cela nous permet d'exécuter des mouvements sur la classe `ChessBoardContoller` pour mettre à jour l'interface et exécuter des mouvements sur la classe `ChessBoard` sans mettre à jour l'interface.

Séparation `ChessBoard` et `ChessBoardStateValidator`

Nous avons découplé l'état de l'échiquier avec la vérification de différents états de jeu. Cela permet de donner une seule responsabilité aux classes.

Interfaces `ChessBoardReader` et `ChessBoardWriter`

Pour encapsuler l'échiquier, deux interfaces limitent l'accès de `ChessBoard`.

Par exemple, une pièce d'échec est uniquement intéressée à lire l'état de l'échiquier pour générer différents types de mouvements. Alors qu'un mouvement (`ChessMove`) doit pouvoir modifier l'état de l'échiquier.

Classe Abstraite `PromotableChessPiece`

Le rôle de cette classe est d'implémenter l'interface `UserChoice` fournie afin d'avoir une représentation textuelle pour l'utilisateur de l'interface de la pièce lors d'une promotion.

2.4 Diagramme UML

Le diagramme UML fournit une vue d'ensemble de la structure et des relations du système. Les éléments grisés représentent le code que nous avons utilisé, mais que nous n'avons pas implémenté nous-mêmes.

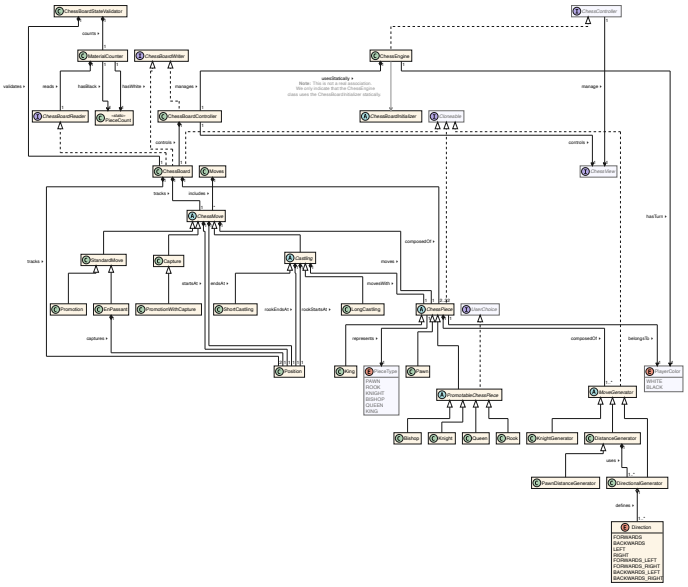

Figure 1: Schéma UML (Vue simplifiée)

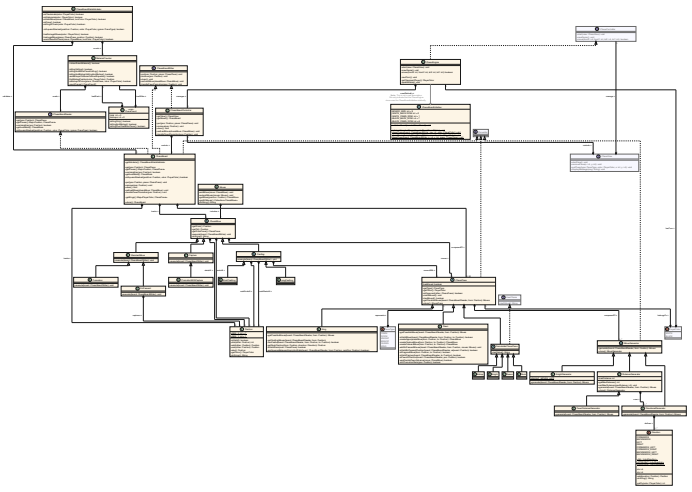

Figure 2: Schéma UML (Vue détaillée)

3 Caractéristiques Principales

Notre jeu implémente les fonctionnalités suivantes pour une partie de joueur contre joueur:

- Les mouvements de base des pièces (pion, tour, cavalier, fou, reine, roi).
- La capture des pièces adverses.
- Le petit et le grand roque
- La prise en passant
- La promotion des pions en avançant
- La promotion des pions en capturant
- La détection et l'affichage de:
  - L'échec
  - L'échec et mat
  - Le pat
  - L'impossibilité de mater

3.1 Règles Spéciales

- **Roque:** Vérifie que le roi et la tour concernés n'ont pas bougé, que le chemin est libre et que les cases traversées ne sont pas attaquées.
- **Prise en passant:** Implémente la capture d'un pion adjacent qui a avancé de deux cases à son premier mouvement.
- **Promotion de pions:** Demande au joueur de choisir un type de promotion (tour, cavalier, fou ou dame).

---

4 Tests Effectués

Tests effectués	**Résultat**
Mettre le roi blanc en échec où le seule mouvement possible est l'attaque de la pièce blanche par une pièce noire	Vrai
En Passant est uniquement pratiquable lorsque le pion adverse avance de deux cases	Vrai
En passant est praticable uniquement au tour suivant et pas 2 tours après	Vrai
Le roque est uniquement praticable si le roi et la tour en question n'ont pas bougé	Vrai
Le roque est pratiquable uniquement si les cases sur lesquelles passe le roi ne sont pas attaquées	Vrai
Les pions peuvent avancer de deux cases uniquement lors de leur premier déplacement	Vrai
Chaque pièce avance dans la bonne direction	Vrai
Uniquement les chevaux peuvent sauter des pièces	Vrai
Les pièces ne peuvent pas découvrir un échec	Vrai
Le roi ne peut pas se mettre en échec	Vrai
Lorsque le roi est en échec, uniquement les mouvements de défenses sont praticables	Vrai

Tests effectués	Résultat
Une pièce ne peut que capturer les pièces d'une autre couleur	Vrai
Un pion peut être promu en reine, fou, chevalier ou tour	Vrai
Un message Check s'affiche lorsque le roi est en échec et Checkmate lorsque quelqu'un a gagné	Vrai
Un message Draw s'affiche lorsqu'il n'est plus possible de faire un checkmate avec le matériel restant	Vrai

4.1 Défense par l'attaque

Les images suivantes montrent que le joueur blanc est obligé d'attaque le fou en H4 avec le cavalier en F3 afin de défendre son roi.

Figure 3: Le roi est bloqué

Figure 4: Le cavalier peut défendre le roi

4.2 Checkmate

Cette image montre que notre jeu est capable de détecter un échec et mat.

Figure 5: Checkmate

4.3 Stalemate

Cette image montre que notre jeu est capable de détecter un pat.

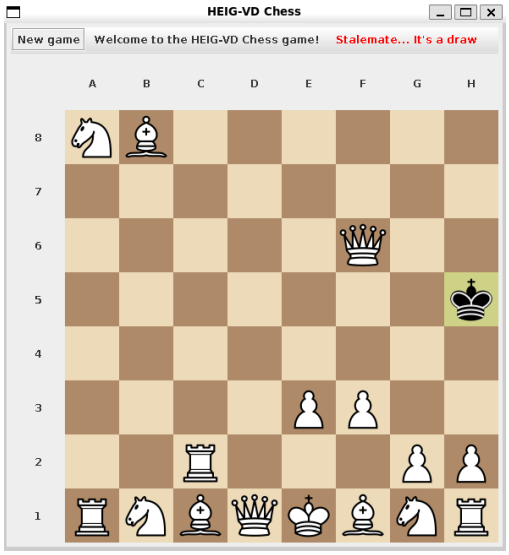

Figure 6: Stalemate

4.4 Draw

Nous observons sur l'image suivante le message d'égalité dû au manque de pièces pour effectuer un échec et mat.

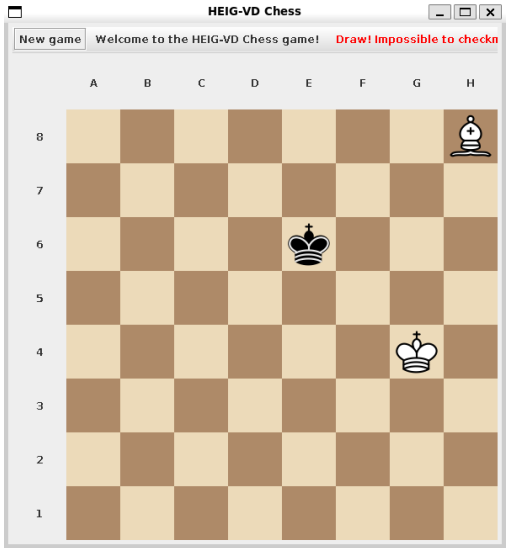

Figure 7: Draw

4.5 En Passant

Sur les deux images ci-dessous, nous pouvons observer que notre jeu propose l'attaque En Passant et permet de l'exécuter.

Figure 8: Le pion a la possibilité de capturer En Passant

Figure 9: Le pion capture En Passant

4.6 Castling

Les trois images suivantes montrent qu'il n'est pas possible d'effectuer un castling si les cases du passage du roi sont attaquées.

Figure 10: Le roi est bloqué car la case à côté est attaquée

Figure 11: Le roi est bloqué car il est attaqué

Figure 12: Le roi peut effectuer un roque

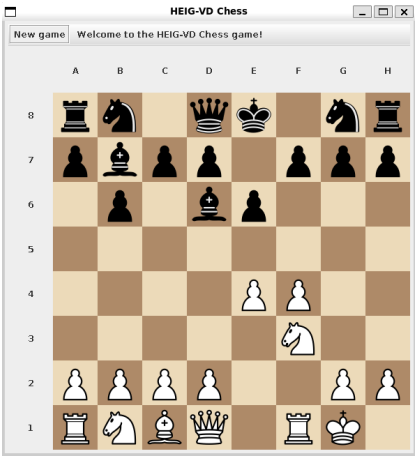

Figure 13: Le roi a effectué le roque

4.7 Promotion

Ci-dessous, nous observons qu'il est possible de promouvoir un pion en reine, tour, fou ou cavalier à l'aide d'une fenêtre de sélection.

Figure 14: Le pion est prêt à être promu

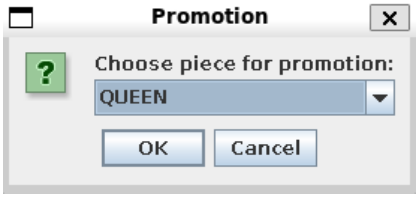

Figure 15: Le joueur peut choisir la promotion du pion

Figure 16: Chaque promotion

---

5 Extensions

5.1 Génération des Mouvements

La hiérarchie `MoveGenerator` encapsule la logique de génération des mouvements:

- **`DirectionalGenerator`:** Pour les mouvements directionnels.
- **`DistanceGenerator`:** Gère les mouvements avec des portées variables, comme les pions.
- **`KnightGenerator`:** Pour les mouvements en L propres aux cavaliers.
- **`PawnDistanceGenerator`:** Spécialise `DistanceGenerator` pour autoriser le mouvement de 2 cases quand le pion n'as pas encore bougé, puis 1 seule case.

Au point de vue conception, la classe `DistanceGenerator` peut utiliser une liste de `DirectionalGenerator` pour obtenir les mouvements d'une distance maximum avec de la directionnalité.

Par exemple, le roi, peut se déplacer dans tous les sens, mais d'une seule case à la fois.

5.2 Gestion des États de Jeu

La logique de détection des états de jeu est encapsulée dans la classe `ChessBoard`. Chaque cas est vérifié de la manière suivante:

- **Échec et Mat (Checkmate):** Vérifie si le roi est en échec et qu'il n'a aucun mouvement légal disponible.
- **Pat (Stalemate):** Vérifie que le roi n'est pas en échec et qu'aucun mouvement légal n'est disponible.
- **Impossibilité de mater (Draw):** Vérifie qu'il n'y a plus de matériel nécessaire pour mater. C'est-à-dire, qu'il vérifie si une des 4 situations

suivantes est vraie:
- Les deux joueurs n'ont plus que leur roi.
- L'un des joueurs n'a plus que son roi et l'autre joueur n'a plus que son roi et un fou.
- L'un des joueurs n'a plus que son roi et l'autre joueur n'a plus que son roi et un cavalier.
- Chaque joueur a un roi et un fou, mais les deux fous sont sur des cases de la même couleur.

5.3 Package `chess`

Nous avons aussi apporté quelques modifications au package `chess` fourni. Comme mentionné dans l'introduction, nous avons fourni deux versions qui incluent/excluent ces modifications.

Changements Esthétiques

Nous avons modifié les images des pions ainsi que les couleurs de l'échiquier. Les images sont sous licence AGPL-3.0 et proviennent du projet Lichess[1].

Soulignement des Mouvements Possibles

Lorsqu'un utilisateur clique sur une pièce, les cases où la pièce peut bouger sont surlignées. Cette fonctionnalité nous a aidé à déboguer les mouvements de pièces.

Pour implémenter cela, nous avons rajouté la méthode suivante dans `ChessView`:

```
/**
 * Highlights with a green dot the given positions
 *
 * @param pos the list of positions
 */
void highlightPositions(List<Position> pos);
```

Et nous avons rajouté la méthode suivante dans `ChessController`:

```
/**
 * Called when the user selects a piece on the board
 *
 * @param x the piece x position
 * @param y the piece y position
 */
void select(int x, int y);
```

Cela nous permet de réagir aux évènements 'select' afin de souligner les mouvements possibles.

[1] https://github.com/lichess-org/lila/tree/master/public/piece/cburnett

6 Conclusion

Ce projet a renforcé les principes de programmation orientée objet tout en abordant des règles et interactions complexes. Les défis ont inclus:

- Garantir l'encapsulation tout en gérant les comportements variés des pièces.
- Traiter les cas limites dans les coups spéciaux et les conditions de fin de jeu.

Améliorations futures possibles:

- Ajouter une IA pour un mode solo.
- Proposer des suggestions de mouvements pour améliorer l'expérience utilisateur.

A Annexes

A.1 Listing Java

c.f. page suivante.

Folder engine

33 printable files

engine/ChessEngine.java
engine/board/ChessBoard.java
engine/board/ChessBoardController.java
engine/board/ChessBoardInitializer.java
engine/board/ChessBoardReader.java
engine/board/ChessBoardStateValidator.java
engine/board/ChessBoardWriter.java
engine/board/MaterialCounter.java
engine/generator/Direction.java
engine/generator/DirectionalGenerator.java
engine/generator/DistanceGenerator.java
engine/generator/KnightGenerator.java
engine/generator/MoveGenerator.java
engine/generator/PawnDistanceGenerator.java
engine/move/Capture.java
engine/move/Castling.java
engine/move/ChessMove.java
engine/move/EnPassant.java
engine/move/LongCastling.java
engine/move/Moves.java
engine/move/Promotion.java
engine/move/PromotionWithCapture.java
engine/move/ShortCastling.java
engine/move/StandardMove.java
engine/piece/Bishop.java
engine/piece/ChessPiece.java
engine/piece/King.java
engine/piece/Knight.java
engine/piece/Pawn.java
engine/piece/Position.java
engine/piece/PromotableChessPiece.java
engine/piece/Queen.java
engine/piece/Rook.java

engine/ChessEngine.java

```java
package engine;

import chess.ChessController;
import chess.ChessView;
import chess.PlayerColor;
import engine.board.ChessBoard;
import engine.board.ChessBoardController;
import engine.board.ChessBoardInitializer;
import engine.move.ChessMove;
import engine.move.Moves;
import engine.piece.Position;

/**
 * Main engine class responsible for managing the chess game logic, turns, and
 * interactions with the view.
 * Implements the {@link ChessController} interface.
```

```
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class ChessEngine implements ChessController {
    private ChessBoardController controller;
    private PlayerColor turnColor;

    /**
     * Starts the chess game, initializes the board, and starts the view.
     *
     * @param view the {@link ChessView} used for displaying the game
     */
    @Override
    public void start(ChessView view) {
        controller = new ChessBoardController(view);
        newGame();
    }

    /**
     * Resets the game state to start a new game.
     *
     * @throws IllegalStateException if the ChessEngine was not initialized properly
     */
    @Override
    public void newGame() {
        turnColor = PlayerColor.WHITE;
        if (controller == null) {
            throw new IllegalStateException("Call ChessEngine.start() before starting a new game");
        }
        ChessBoardInitializer.initializeBoard(controller);
    }

    /**
     * Attempts to make a move on the chessboard from the given coordinates.
     *
     * @param fromX the starting X-coordinate
     * @param fromY the starting Y-coordinate
     * @param toX   the destination X-coordinate
     * @param toY   the destination Y-coordinate
     * @return true if the move is successful, false otherwise
     */
    @Override
    public boolean move(int fromX, int fromY, int toX, int toY) {
        Position from = new Position(fromX, fromY);
        Position to = new Position(toX, toY);
        assert from.isValid() : "From position is invalid";
        assert to.isValid() : "To position is invalid";
        assert controller.getBoard().containsKey(from) : "From position is invalid";

        ChessBoard board = controller.getBoard();
        Moves moves = board.get(from).getPossibleMoves(board, from);
        ChessMove move = moves.getMove(to);
        if (!board.getValidator().isValidMove(move, turnColor)) {
            return false;
        }
        move.execute(controller);
        nextTurn();
        updateState();
        return true;
    }

    /**
     * Switches to the next player's turn.
     */
```

```
    private void nextTurn() {
        turnColor = getOpponentPlayer();
    }

    /**
     * Determines the color of the opposing player.
     *
     * @return the color of the opposing player
     */
    private PlayerColor getOpponentPlayer() {
        return turnColor == PlayerColor.WHITE ? PlayerColor.BLACK : PlayerColor.WHITE;
    }

    /**
     * Displays a message to the view if an event occurred
     */
    private void updateState() {
        ChessBoard board = controller.getBoard();
        String event;
        if (board.getValidator().isCheckmate(turnColor)) {
            event = "Checkmate! " + getOpponentPlayer() + " won!";
        } else if (board.getValidator().isStalemate(turnColor)) {
            event = "Stalemate... It's a draw";
        } else if (board.getValidator().isDraw()) {
            event = "Draw! Impossible to checkmate";
        } else if (board.getValidator().isKingInCheck(turnColor)) {
            event = "Check!";
        } else {
            event = null;
        }

        if (event != null) {
            controller.getView().displayMessage(event);
        }
    }
}
```

engine/board/ChessBoard.java

```
package engine.board;

import chess.PieceType;
import chess.PlayerColor;
import engine.move.ChessMove;
import engine.piece.ChessPiece;
import engine.piece.Position;
import engine.piece.Queen;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

/**
 * Represents the chessboard, managing the state of the game, including pieces
 * and positions.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class ChessBoard implements ChessBoardReader, ChessBoardWriter, Cloneable {
    private Map<Position, ChessPiece> pieces = new HashMap<>();
    private ChessMove lastMove = null;
    private Map<PlayerColor, Position> kings = new HashMap<>();
```

```java
/**
 * Creates a new chessboard state validator and returns it
 *
 * @return the chessboard state validator
 */
public ChessBoardStateValidator getValidator() {
    return new ChessBoardStateValidator(clone());
}

/**
 * Retrieves the chess piece at the specified position.
 *
 * @param pos the position on the chessboard
 * @return the {@link ChessPiece} at the specified position, or null if empty
 */
@Override
public ChessPiece get(Position pos) {
    return pieces.get(pos);
}

/**
 * Checks if a given position contains a chess piece.
 *
 * @param pos the position to check
 * @return true if a piece exists at the given position, false otherwise
 */
@Override
public boolean containsKey(Position pos) {
    return pieces.containsKey(pos);
}

/**
 * Retrieves the last move that was made on the chessboard.
 *
 * @return the last move that was made on the chessboard
 */
@Override
public ChessMove getLastMove() {
    return lastMove;
}

/**
 * Sets the last move that was made on the chessboard.
 *
 * @param chessMove the last move that was made
 */
@Override
public void setLastMove(ChessMove chessMove) {
    lastMove = chessMove;
}

/**
 * Places a chess piece at the specified position on the board.
 * Updates the view and tracks the position of kings.
 *
 * @param pos   the position to place the piece
 * @param piece the {@link ChessPiece} to place
 */
@Override
public void put(Position pos, ChessPiece piece) {
    pieces.put(pos, piece);
    if (piece.getType() == PieceType.KING) {
        kings.put(piece.getColor(), pos);
    }
```

```
}

/**
 * Removes a chess piece from the specified position.
 *
 * @param pos the position to remove the piece from
 * @throws IllegalStateException if no piece exit at the position
 */
@Override
public void remove(Position pos) {
    if (pieces.get(pos) == null) {
        throw new IllegalStateException("No piece exits at " + pos);
    }
    pieces.remove(pos);
}

/**
 * Clears all pieces from the chessboard.
 */
@Override
public void clear() {
    pieces.clear();
}

/**
 * Handles pawn promotion at the given position.
 * Defaults to a queen.
 *
 * @param pos the position of the pawn being promoted
 */
@Override
public void handlePawnPromotion(Position pos) {
    put(pos, new Queen(get(pos).getColor()));
}

/**
 * Checks if the square at the given position is attacked by any piece of the
 * given color.
 *
 * @param position the position to check
 * @param color    the color of the attacking pieces
 * @param ignore   the piece type to ignore, can be set to null to check all
 *                 piece types
 * @return true if the square is attacked, false otherwise
 */
@Override
public boolean isSquareAttacked(Position position, PlayerColor color, PieceType ignore) {
    return new ChessBoardStateValidator(this).isSquareAttacked(position, color, ignore);
}

/**
 * Get all the chessboard pieces
 *
 * @return a map of the positions its piece
 */
@Override
public Map<Position, ChessPiece> getPieces() {
    return Collections.unmodifiableMap(pieces);
}

/**
 * Gets the kings mapped by player color
 *
 * @return the kings
 */
```

```
    Map<PlayerColor, Position> getKings() {
        return kings;
    }

    /**
     * Creates a deep clone of this chessboard, including all pieces.
     *
     * @return a new {@link ChessBoard} instance identical to this one
     * @throws AssertionError if the clone failed. We assert it won't happen
     */
    @Override
    public ChessBoard clone() {
        try {
            ChessBoard clonedBoard = (ChessBoard) super.clone();
            // Deep copy the pieces map
            clonedBoard.pieces = new HashMap<>();
            for (Map.Entry<Position, ChessPiece> entry : pieces.entrySet()) {
                clonedBoard.pieces.put(entry.getKey(), entry.getValue().clone());
            }
            // Deep copy the kings map
            clonedBoard.kings = new HashMap<>(kings);
            return clonedBoard;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError("Cloning failed", e);
        }
    }
}
```

engine/board/ChessBoardController.java

```
package engine.board;

import chess.ChessView;
import chess.PlayerColor;
import engine.move.ChessMove;
import engine.piece.Bishop;
import engine.piece.ChessPiece;
import engine.piece.Knight;
import engine.piece.Position;
import engine.piece.PromotableChessPiece;
import engine.piece.Queen;
import engine.piece.Rook;

/**
 * Wraps the ChessBoard, implementing the ChessBoardWrite interface such that
 * it can interact with the ChessView in conjunction with the ChessBoard.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class ChessBoardController implements ChessBoardWriter {
    private final ChessBoard board = new ChessBoard();
    private final ChessView view;

    /**
     * Instantiates the ChessBoardController.
     *
     * @param view the ChessView
     */
    public ChessBoardController(ChessView view) {
        this.view = view;
        this.view.startView();
    }
```

```java
/**
 * Gets the associated ChessView.
 *
 * @return the ChessView
 */
public ChessView getView() {
    return view;
}

/**
 * Gets the associated ChessBoard.
 *
 * @return the ChessBoard
 */
public ChessBoard getBoard() {
    return board;
}

/**
 * Places a chess piece at the specified position on the board.
 * Updates the view and tracks the position of kings.
 *
 * @param pos   the position to place the piece
 * @param piece the {@link ChessPiece} to place
 */
@Override
public void put(Position pos, ChessPiece piece) {
    board.put(pos, piece);
    view.putPiece(piece.getType(), piece.getColor(), pos.x(), pos.y());
}

/**
 * Removes a chess piece from the specified position.
 *
 * @param pos the position to remove the piece from
 * @throws IllegalStateException if no piece exit at the position
 */
@Override
public void remove(Position pos) {
    board.remove(pos);
    view.removePiece(pos.x(), pos.y());
}

/**
 * Clears all pieces from the chessboard.
 */
@Override
public void clear() {
    for (Position pos : board.getPieces().keySet()) {
        view.removePiece(pos.x(), pos.y());
    }
    board.clear();
}

/**
 * Sets the last move that was made on the chessboard.
 *
 * @param chessMove the last move that was made
 */
@Override
public void setLastMove(ChessMove chessMove) {
    board.setLastMove(chessMove);
}
```

```
    /**
     * Handles pawn promotion at the given position.
     * Prompts the user though the ChessView.
     *
     * @param pos the position of the pawn being promoted
     */
    @Override
    public void handlePawnPromotion(Position pos) {
        PlayerColor color = board.get(pos).getColor();
        PromotableChessPiece chosen = view.askUser(
                "Promotion",
                "Choose piece for promotion:",
                new Queen(color), new Rook(color), new Bishop(color), new Knight(color));
        put(pos, chosen);

    }
}
```

engine/board/ChessBoardInitializer.java

```
package engine.board;

import chess.PlayerColor;
import engine.piece.*;

/**
 * Utility class for initializing a chessboard with different piece
 * configurations.
 * Provides methods for standard chess setup and supports custom board
 * arrangements.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public abstract class ChessBoardInitializer {
    private static final int BOARD_SIZE = 8;
    private static final int WHITE_BACK_ROW = 0;
    private static final int WHITE_PAWN_ROW = 1;
    private static final int BLACK_BACK_ROW = 7;
    private static final int BLACK_PAWN_ROW = 6;

    /**
     * Initializes the chessboard with the standard chess piece configuration.
     *
     * @param board the board to initialize
     */
    public static void initializeBoard(ChessBoardWriter board) {
        board.clear();
        initializeStandardGame(board);
    }

    /**
     * Sets up the standard chess game configuration.
     *
     * @param board the board to initialize
     */
    private static void initializeStandardGame(ChessBoardWriter board) {
        placePieces(board, WHITE_BACK_ROW, PlayerColor.WHITE);
        placePieces(board, BLACK_BACK_ROW, PlayerColor.BLACK);
        placePawns(board, WHITE_PAWN_ROW, PlayerColor.WHITE);
        placePawns(board, BLACK_PAWN_ROW, PlayerColor.BLACK);
    }
```

```
    /**
     * Places all pieces for one player's back row according to standard chess
     * rules.
     *
     * @param board the board to place pieces on
     * @param row   the row number to place pieces
     * @param color the color of the pieces to place
     */
    private static void placePieces(ChessBoardWriter board, int row, PlayerColor color) {
        int col = 0;
        placePiece(board, col++, row, new Rook(color));
        placePiece(board, col++, row, new Knight(color));
        placePiece(board, col++, row, new Bishop(color));
        placePiece(board, col++, row, new Queen(color));
        placePiece(board, col++, row, new King(color));
        placePiece(board, col++, row, new Bishop(color));
        placePiece(board, col++, row, new Knight(color));
        placePiece(board, col, row, new Rook(color));
    }

    /**
     * Places pawns for one player's row.
     *
     * @param board the board to place pawns on
     * @param row   the row number to place pawns
     * @param color the color of the pawns to place
     */
    private static void placePawns(ChessBoardWriter board, int row, PlayerColor color) {
        for (int col = 0; col < BOARD_SIZE; col++) {
            placePiece(board, col, row, new Pawn(color));
        }
    }

    /**
     * Places a single piece on the board at the specified position.
     *
     * @param board the board to place the piece on
     * @param x     the x-coordinate
     * @param y     the y-coordinate
     * @param piece the piece to place
     */
    private static void placePiece(ChessBoardWriter board, int x, int y, ChessPiece piece) {
        board.put(new Position(x, y), piece);
    }
}
```

engine/board/ChessBoardReader.java

```
package engine.board;

import chess.PieceType;
import chess.PlayerColor;
import engine.move.ChessMove;
import engine.piece.ChessPiece;
import engine.piece.Position;

import java.util.Map;

/**
 * Read-only interface for the ChessBoard.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
```

```
 */
public interface ChessBoardReader {
    /**
     * Retrieves the chess piece located at the specified position.
     *
     * @param pos the position on the chessboard
     * @return the {@link ChessPiece} at the given position, or null if no piece is
     * present
     */
    ChessPiece get(Position pos);

    /**
     * Get all the chessboard pieces
     *
     * @return a map of the positions its piece
     */
    Map<Position, ChessPiece> getPieces();

    /**
     * Checks if the specified position contains a chess piece.
     *
     * @param pos the position on the chessboard
     * @return true if a piece is present at the given position, false otherwise
     */
    boolean containsKey(Position pos);

    /**
     * Retrieves the last move that was made on the chessboard.
     *
     * @return the last move that was made on the chessboard
     */
    ChessMove getLastMove();

    /**
     * Checks if the square at the given position is attacked by any piece of the
     * given color.
     *
     * @param position the position to check
     * @param color    the color of the attacking pieces
     * @param ignore   the piece type to ignore, can be set to null to check all
     *                 piece types
     * @return true if the square is attacked, false otherwise
     */
    boolean isSquareAttacked(Position position, PlayerColor color, PieceType ignore);
}
```

engine/board/ChessBoardStateValidator.java

```
package engine.board;

import chess.PieceType;
import chess.PlayerColor;
import engine.move.ChessMove;
import engine.move.Moves;
import engine.piece.ChessPiece;
import engine.piece.Position;

/**
 * Validates chess game states including checkmate, stalemate, draws, and move
 * validity.
 * Separates game state validation logic from board management.
 *
 * @author Leonard Cseres
```

```java
 * @author Aladin Iseni
 */
public final class ChessBoardStateValidator {
    private final ChessBoard board;
    private final MaterialCounter materialCounter;

    /**
     * Creates a new ChessBoardStateValidator to validate game states and moves for
     * the provided board.
     *
     * @param board the chess board to validate
     */
    public ChessBoardStateValidator(ChessBoard board) {
        this.board = board;
        this.materialCounter = new MaterialCounter(board);
    }

    /**
     * Checks if the player of the given color is in checkmate.
     *
     * @param color the color of the player to check
     * @return true if the player is in checkmate, false otherwise
     */
    public boolean isCheckmate(PlayerColor color) {
        return isKingInCheck(color) && hasNoLegalMoves(color);
    }

    /**
     * Checks if the player of the given color is in stalemate.
     *
     * @param color the color of the player to check
     * @return true if the player is in stalemate, false otherwise
     */
    public boolean isStalemate(PlayerColor color) {
        return !isKingInCheck(color) && hasNoLegalMoves(color);
    }

    /**
     * Validates if a move is legal considering check conditions.
     *
     * @param move      the move to validate
     * @param turnColor the color of the player making the move
     * @return true if the move is valid, false otherwise
     */
    public boolean isValidMove(ChessMove move, PlayerColor turnColor) {
        if (move == null || move.getFromPiece().getColor() != turnColor) {
            return false;
        }
        return !wouldResultInCheck(move, turnColor);
    }

    /**
     * Checks if the game is a draw based on insufficient material.
     * Handles scenarios: K vs K, K+B vs K, K+N vs K, and K+B vs K+B (same colored
     * squares)
     *
     * @return true if the game is a draw due to insufficient material
     */
    public boolean isDraw() {
        return materialCounter.isInsufficientMaterial();
    }

    /**
     * Checks if the king of the given color is in check.
     *
```

```
     * @param kingColor the color of the king to check
     * @return true if the king is in check, false otherwise
     */
    public boolean isKingInCheck(PlayerColor kingColor) {
        Position kingPosition = board.getKings().get(kingColor);
        return isSquareAttacked(kingPosition, kingColor, null);
    }

    /**
     * Checks if the square at the given position is attacked by any piece of the
     * given color.
     *
     * @param position the position to check
     * @param color    the color of the attacking pieces
     * @param ignore   the piece type to ignore, can be set to null to check all
     *                 piece types
     * @return true if the square is attacked, false otherwise
     */
    boolean isSquareAttacked(Position position, PlayerColor color, PieceType ignore) {
        return board.getPieces().entrySet().stream()
                .filter(entry -> {
                    ChessPiece piece = entry.getValue();
                    return piece.getColor() != color &&
                            (ignore == null || ignore != piece.getType());
                }).anyMatch(entry -> {
                    Moves possibleMoves = entry.getValue().getPossibleMoves(board, entry.getKey());
                    return possibleMoves.getMove(position) != null;
                });
    }

    /**
     * Determines if the current board state results in the given color having no
     * legal moves.
     *
     * @param color the color of the player to check
     * @return true if the player has no legal moves, false otherwise
     */
    private boolean hasNoLegalMoves(PlayerColor color) {
        return board.getPieces().entrySet().stream()
                .filter(entry -> entry.getValue().getColor() == color)
                .noneMatch(entry -> hasLegalMove(entry.getValue(), entry.getKey()));
    }

    /**
     * Determines if the given piece at the specified position has any legal moves.
     *
     * @param piece    the chess piece to check
     * @param position the position of the chess piece
     * @return true if the piece has at least one legal move, false otherwise
     */
    private boolean hasLegalMove(ChessPiece piece, Position position) {
        Moves possibleMoves = piece.getPossibleMoves(board, position);
        return possibleMoves.getAllMoves().stream()
                .anyMatch(move -> !wouldResultInCheck(move, piece.getColor()));
    }

    /**
     * Simulates a move on a cloned board to determine if it results in the king
     * being in check.
     *
     * @param move      the move to simulate
     * @param turnColor the color of the player making the move
     * @return true if the simulated move results in the king being in check, false
     * otherwise
     */
```

```
    private boolean wouldResultInCheck(ChessMove move, PlayerColor turnColor) {
        ChessBoard testBoard = board.clone();
        move.execute(testBoard);
        return testBoard.getValidator().isKingInCheck(turnColor);
    }
}
```

engine/board/ChessBoardWriter.java

```
package engine.board;

import engine.move.ChessMove;
import engine.piece.ChessPiece;
import engine.piece.Position;

/**
 * Write-only interface for the ChessBoard.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public interface ChessBoardWriter {
    /**
     * Places a chess piece at the specified position on the board.
     * Updates the view and tracks the position of kings.
     *
     * @param pos   the position to place the piece
     * @param piece the {@link ChessPiece} to place
     */
    void put(Position pos, ChessPiece piece);

    /**
     * Removes a chess piece from the specified position.
     *
     * @param pos the position to remove the piece from
     * @throws IllegalStateException if no piece exit at the position
     */
    void remove(Position pos);

    /**
     * Clears all pieces from the chessboard.
     */
    void clear();

    /**
     * Sets the last move that was made on the chessboard.
     *
     * @param chessMove the last move that was made
     */
    void setLastMove(ChessMove chessMove);

    /**
     * Handles pawn promotion at the given position.
     *
     * @param pos the position of the pawn being promoted
     */
    void handlePawnPromotion(Position pos);
}
```

engine/board/MaterialCounter.java

```java
package engine.board;

import chess.PieceType;
import chess.PlayerColor;
import engine.piece.ChessPiece;
import engine.piece.Position;

import java.util.Map;

/**
 * Helper class to handle material counting and insufficient material detection.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
final class MaterialCounter {
    private final ChessBoardReader board;
    private final PieceCount whiteCount;
    private final PieceCount blackCount;

    /**
     * Constructs a MaterialCounter to analyze the material on the given chess board.
     *
     * @param board the chess board to analyze
     */
    MaterialCounter(ChessBoardReader board) {
        this.board = board;
        PieceCount[] counts = countPieces();
        this.whiteCount = counts[0];
        this.blackCount = counts[1];
    }

    /**
     * Checks if the game is a draw due to insufficient material.
     * Considers scenarios such as King vs King, King and minor piece vs King,
     * and King with a bishop vs King with a bishop on same-colored squares.
     *
     * @return true if the game is a draw due to insufficient material, false otherwise
     */
    boolean isInsufficientMaterial() {
        return isKingVsKing() || isKingAndMinorPieceVsKing() || isKingAndBishopVsKingAndBishop();
    }

    /**
     * Checks if the board represents a King vs King scenario.
     *
     * @return true if only kings remain, false otherwise
     */
    private boolean isKingVsKing() {
        return whiteCount.total == 1 && blackCount.total == 1;
    }

    /**
     * Checks if the board represents a King and one minor piece vs King scenario.
     *
     * @return true if one side has a king and a single minor piece, and the other side has only a king
     */
    private boolean isKingAndMinorPieceVsKing() {
        return (whiteCount.isKingPlusOneMinorPiece() && blackCount.isKingOnly()) ||
                (blackCount.isKingPlusOneMinorPiece() && whiteCount.isKingOnly());
    }

    /**
     * Checks if the board represents a King and Bishop vs King and Bishop scenario
     * where the bishops are on same-colored squares.
```

```
     *
     * @return true if both sides have a King and Bishop, and the bishops are on the same color
     */
    private boolean isKingAndBishopVsKingAndBishop() {
        return whiteCount.isKingAndBishop() && blackCount.isKingAndBishop() &&
                areBishopsOnSameColoredSquares();
    }

    /**
     * Determines if the bishops on both sides are on the same-colored squares.
     *
     * @return true if bishops are on the same color, false otherwise
     */
    private boolean areBishopsOnSameColoredSquares() {
        Position whiteBishop = findBishopPosition(PlayerColor.WHITE);
        Position blackBishop = findBishopPosition(PlayerColor.BLACK);
        return whiteBishop != null && blackBishop != null &&
                whiteBishop.getColor() == blackBishop.getColor();
    }

    /**
     * Finds the position of a bishop for the given player color.
     *
     * @param color the color of the player to find the bishop for
     * @return the position of the bishop, or null if none exists
     */
    private Position findBishopPosition(PlayerColor color) {
        return board.getPieces().entrySet().stream()
                .filter(entry -> isBishopOfColor(entry.getValue(), color))
                .map(Map.Entry::getKey)
                .findFirst()
                .orElse(null);
    }

    /**
     * Checks if the given piece is a bishop of the specified color.
     *
     * @param piece the chess piece to check
     * @param color the color to match
     * @return true if the piece is a bishop of the specified color, false otherwise
     */
    private boolean isBishopOfColor(ChessPiece piece, PlayerColor color) {
        return piece.getType() == PieceType.BISHOP && piece.getColor() == color;
    }

    /**
     * Counts the total pieces and bishops for both players on the board.
     *
     * @return an array containing the piece counts for white [0] and black [1]
     */
    private PieceCount[] countPieces() {
        PieceCount white = new PieceCount();
        PieceCount black = new PieceCount();

        board.getPieces().values().forEach(piece -> {
            PieceCount count = (piece.getColor() == PlayerColor.WHITE) ? white : black;
            count.total++;
            if (piece.getType() == PieceType.BISHOP) {
                count.bishops++;
            }
        });

        return new PieceCount[]{white, black};
    }
```

```
    /**
     * Helper class to track piece counts for each player.
     */
    private static final class PieceCount {
        private int total = 0;
        private int bishops = 0;

        /**
         * Checks if only a king remains.
         *
         * @return true if only a king is present, false otherwise
         */
        private boolean isKingOnly() {
            return total == 1;
        }

        /**
         * Checks if the player has exactly a king and a single bishop.
         *
         * @return true if the player has a king and a bishop, false otherwise
         */
        private boolean isKingAndBishop() {
            return total == 2 && bishops == 1;
        }

        /**
         * Checks if the player has a king and one minor piece (bishop or knight).
         *
         * @return true if the player has a king and one minor piece, false otherwise
         */
        private boolean isKingPlusOneMinorPiece() {
            return total == 2;
        }
    }
}
```

engine/generator/Direction.java

```
package engine.generator;

import chess.PlayerColor;
import engine.piece.Position;

import java.util.List;

/**
 * Enum representing the possible directions a chess piece can move on the
 * board.
 * Each direction is represented by a pair of x and y changes (dx, dy).
 * The directions include vertical, horizontal, and diagonal movements.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public enum Direction {
    FORWARDS(0, 1),
    BACKWARDS(0, -1),
    LEFT(-1, 0),
    RIGHT(1, 0),
    FORWARDS_LEFT(-1, 1),
    FORWARDS_RIGHT(1, 1),
    BACKWARDS_LEFT(-1, -1),
    BACKWARDS_RIGHT(1, -1);
```

```java
    public static final List<Direction> ALL = List.of(Direction.FORWARDS, Direction.BACKWARDS, Direction.LEFT,
            Direction.RIGHT, Direction.FORWARDS_LEFT, Direction.FORWARDS_RIGHT, Direction.BACKWARDS_LEFT,
            Direction.BACKWARDS_RIGHT);
    public static final List<Direction> STRAIGHT = List.of(Direction.FORWARDS, Direction.BACKWARDS, Direction.LE
            Direction.RIGHT);
    public static final List<Direction> DIAGONAL = List.of(Direction.FORWARDS_LEFT, Direction.FORWARDS_RIGHT,
            Direction.BACKWARDS_LEFT, Direction.BACKWARDS_RIGHT);

    private final int dx;
    private final int dy;

    /**
     * Constructor for the Direction enum, defining the change in position (dx, dy).
     *
     * @param dx the change in x-coordinate (horizontal movement)
     * @param dy the change in y-coordinate (vertical movement)
     * @throws IllegalArgumentException if the provided arguments are invalid
     */
    Direction(int dx, int dy) {
        if (dx < -1 || dx > 1) {
            throw new IllegalArgumentException("dx must be between -1 and 1");
        }
        if (dy < -1 || dy > 1) {
            throw new IllegalArgumentException("dy must be between -1 and 1");
        }
        this.dx = dx;
        this.dy = dy;
    }

    /**
     * Adjusts the vertical movement (dy) based on the player's color.
     *
     * @param color the color of the player (used to determine direction)
     * @return the adjusted vertical movement (dy) based on the color
     */
    private int getDy(PlayerColor color) {
        return color == PlayerColor.WHITE ? dy : -dy;
    }

    /**
     * Calculates a new position by applying this direction to the given position,
     * taking the piece color into account.
     *
     * @param position the current position of the piece
     * @param color    the color of the piece (used to adjust direction)
     * @return the new position after applying the direction
     */
    public Position add(Position position, PlayerColor color) {
        return position.add(new Position(dx, getDy(color)));
    }

    /**
     * Returns a string representation of the direction, including the dx and dy
     * values.
     *
     * @return a string representation of the direction
     */
    @Override
    public String toString() {
        return getClass().getSimpleName() + " (" + dx + ", " + dy + ")";
    }
}
```

engine/generator/DirectionalGenerator.java

```java
package engine.generator;

import engine.board.ChessBoardReader;
import engine.move.Capture;
import engine.move.Moves;
import engine.move.StandardMove;
import engine.piece.ChessPiece;
import engine.piece.Position;

import java.util.List;

/**
 * Generates possible moves for pieces that move in specific directions.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public class DirectionalGenerator extends MoveGenerator {
    private final List<Direction> dirs;

    /**
     * Constructs a DirectionalGenerator with specified directions
     * capability.
     *
     * @param dirs the directions the piece can move in
     */
    public DirectionalGenerator(Direction... dirs) {
        this(List.of(dirs));
    }

    /**
     * Constructs a DirectionalGenerator with specified directions
     * capability.
     *
     * @param dirs the directions the piece can move in
     */
    public DirectionalGenerator(List<Direction> dirs) {
        this.dirs = dirs;
    }

    /**
     * Generates all possible moves at a specified position at given directions
     *
     * @param board the current state of the chessboard
     * @param from  the position of the piece on the board
     * @return a collection of possible moves
     */
    @Override
    public Moves generate(ChessBoardReader board, Position from) {
        Moves possibleMoves = new Moves();
        ChessPiece piece = board.get(from);

        for (Direction dir : dirs) {
            Position current = from;

            while (true) {
                current = dir.add(current, piece.getColor());

                if (!current.isValid()) {
                    break;
                }
```

```
                if (board.containsKey(current)) {
                    // If there's a piece at the current position
                    ChessPiece otherPiece = board.get(current);
                    if (otherPiece.isOpponent(piece)) {
                        possibleMoves.addMove(new Capture(from, current, piece));
                    }
                    break; // Stop further exploration the piece cannot jump
                } else {
                    // Add the move if the square is empty
                    possibleMoves.addMove(new StandardMove(from, current, piece));
                }
            }
        }

        return possibleMoves;
    }
}
```

engine/generator/DistanceGenerator.java

```
package engine.generator;

import engine.board.ChessBoardReader;
import engine.move.ChessMove;
import engine.move.Moves;
import engine.piece.Position;

import java.util.List;

/**
 * Generates possible moves for pieces that have a maximum distance they can
 * move.
 * Supports a collection of DirectionalGenerators to generate moves in multiple
 * directions.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public class DistanceGenerator extends MoveGenerator {
    private int maxDistance;
    private final List<DirectionalGenerator> directionalGenerators;

    /**
     * Constructs a DistanceGenerator with the specified maximum distance and
     * directional generators.
     *
     * @param maxDistance           the maximum distance the piece can move
     * @param directionalGenerators the generators that handle the piece's movement
     *                              in different directions
     */
    public DistanceGenerator(int maxDistance, DirectionalGenerator... directionalGenerators) {
        this.maxDistance = maxDistance;
        this.directionalGenerators = List.of(directionalGenerators);
    }

    /**
     * Gets the maximum distance the piece can move.
     *
     * @return the maximum distance
     */
    public int getMaxDistance() {
        return maxDistance;
    }
```

```java
    /**
     * Sets the maximum distance the piece can move.
     *
     * @param maxDistance the maximum distance
     */
    public void setMaxDistance(int maxDistance) {
        this.maxDistance = maxDistance;
    }

    /**
     * Generates all possible moves at a specified position given a max distance and
     * directions
     *
     * @param board the current state of the chessboard
     * @param from  the position of the piece on the board
     * @return a collection of possible moves
     */
    @Override
    public Moves generate(ChessBoardReader board, Position from) {
        Moves possibleMoves = new Moves();

        // Generate moves using all directional generators
        for (DirectionalGenerator gen : directionalGenerators) {
            Moves generatedMoves = gen.generate(board, from);

            // Filter moves that exceed the maximum distance
            for (ChessMove move : generatedMoves.getAllMoves()) {
                if (from.dist(move.getTo()) <= maxDistance) {
                    possibleMoves.addMove(move);
                }
            }
        }

        return possibleMoves;
    }

    /**
     * Creates a deep clone of the move generator
     *
     * @return a cloned instance of the move generator
     * @throws CloneNotSupportedException if the cloning process fails
     */
    @Override
    public DistanceGenerator clone() throws CloneNotSupportedException {
        DistanceGenerator dg = (DistanceGenerator) super.clone();
        dg.maxDistance = maxDistance;
        return dg;
    }
}
```

engine/generator/KnightGenerator.java

```java
package engine.generator;

import engine.board.ChessBoardReader;
import engine.move.Capture;
import engine.move.Moves;
import engine.move.StandardMove;
import engine.piece.ChessPiece;
import engine.piece.Position;

/**
```

```java
 * Generates possible moves for a knight piece on the chessboard.
 * The knight moves in an "L" shape: two squares in one direction and one square
 * perpendicular to that.
 * It can jump over other pieces.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class KnightGenerator extends MoveGenerator {
    // Possible moves for a knight (8 directions)
    private static final int[][] KNIGHT_MOVES = {
            {2, 1}, {2, -1}, {-2, 1}, {-2, -1},
            {1, 2}, {1, -2}, {-1, 2}, {-1, -2}
    };

    /**
     * Generates all possible moves the knight at a specified position
     *
     * @param board the current state of the chessboard
     * @param from  the position of the piece on the board
     * @return a collection of possible moves
     */
    @Override
    public Moves generate(ChessBoardReader board, Position from) {
        Moves moves = new Moves();
        ChessPiece piece = board.get(from);

        // Evaluate each possible knight move
        for (int[] move : KNIGHT_MOVES) {
            Position to = from.add(new Position(move[0], move[1]));

            // If the move is valid and the destination is either empty or occupied by an
            // opponent
            if (to.isValid()) {
                if (!board.containsKey(to)) {
                    moves.addMove(new StandardMove(from, to, piece));
                } else if (board.get(to).isOpponent(piece)) {
                    moves.addMove(new Capture(from, to, piece));
                }
            }
        }

        return moves;
    }
}
```

engine/generator/MoveGenerator.java

```java
package engine.generator;

import engine.board.ChessBoardReader;
import engine.move.Moves;
import engine.piece.Position;

/**
 * Abstract class for generating possible moves for a chess piece.
 * Implementations of this class will define how to generate moves
 * for specific types of chess pieces.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public abstract class MoveGenerator implements Cloneable {
```

```
    /**
     * Generates all possible moves for a given piece from a specific position.
     *
     * @param board the current state of the chessboard
     * @param from  the position of the piece on the board
     * @return a collection of possible moves
     */
    public abstract Moves generate(ChessBoardReader board, Position from);

    /**
     * Creates a deep clone of the move generator
     *
     * @return a cloned instance of the move generator
     * @throws CloneNotSupportedException if the cloning process fails
     */
    @Override
    public MoveGenerator clone() throws CloneNotSupportedException {
        return (MoveGenerator) super.clone();
    }
}
```

engine/generator/PawnDistanceGenerator.java

```
package engine.generator;

import engine.board.ChessBoardReader;
import engine.move.Moves;
import engine.piece.ChessPiece;
import engine.piece.Position;

/**
 * Generates possible moves for a pawn piece on the chessboard.
 * The pawn can move one or two squares forward on its first move, and one
 * square forward thereafter.
 * It does not consider diagonal captures, which are handled in the
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class PawnDistanceGenerator extends DistanceGenerator {

    public PawnDistanceGenerator() {
        super(2, new DirectionalGenerator(Direction.FORWARDS));
    }

    @Override
    public Moves generate(ChessBoardReader board, Position from) {
        ChessPiece piece = board.get(from);
        // If the pawn has moved, restrict its maximum distance to 1
        if (piece.hasMoved() && getMaxDistance() == 2) {
            setMaxDistance(1);
        }
        return super.generate(board, from);
    }
}
```

engine/move/Capture.java

```
package engine.move;
```

```java
import engine.board.ChessBoardWriter;
import engine.piece.ChessPiece;
import engine.piece.Position;

/**
 * Represents a capture move in chess, where a piece moves to a position
 * occupied by an opponent's piece.
 * The opponent's piece is removed from the board, and the moving piece replaces
 * it.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public class Capture extends ChessMove {

    /**
     * Constructs a Capture move with the specified starting and destination
     * positions.
     *
     * @param from      the starting position of the move
     * @param to        the destination position of the move (where the opponent's piece
     *                  will be captured)
     * @param fromPiece the starting position chess piece
     */
    public Capture(Position from, Position to, ChessPiece fromPiece) {
        super(from, to, fromPiece);
    }

    /**
     * Executes the capture move on the provided chess board.
     * The piece is moved from the starting position to the destination position,
     * and the opponent's piece at the destination is removed from the board.
     *
     * @param board the chessboard on which the move is executed
     */
    @Override
    public void execute(ChessBoardWriter board) {
        super.execute(board);
        board.remove(from);
        fromPiece.markMoved();
        board.remove(to);
        board.put(to, fromPiece);
    }
}
```

engine/move/Castling.java

```java
package engine.move;

import engine.board.ChessBoardWriter;
import engine.piece.ChessPiece;
import engine.piece.Position;

/**
 * Abstract base class representing a castling move in chess.
 * Provides common functionality for both long (queenside) and short (kingside)
 * castling.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
abstract class Castling extends ChessMove {
    private final Position fromRook;
```

```
    private final Position toRook;
    private final ChessPiece rook;

    /**
     * Constructs a Castling move with the specified starting and destination
     * positions for the king.
     *
     * @param from     the starting position of the king
     * @param to       the destination position of the king
     * @param king     the starting position king
     * @param fromRook the starting position of the rook
     * @param toRook   the destination position of the rook
     * @param rook     the starting position rook
     */
    protected Castling(Position from, Position to, ChessPiece king, Position fromRook, Position toRook,
            ChessPiece rook) {
        super(from, to, king);
        this.fromRook = fromRook;
        this.toRook = toRook;
        this.rook = rook.clone();
    }

    /**
     * Executes the castling move on the provided chessboard.
     *
     * @param board the chessboard on which the move is executed
     */
    @Override
    public void execute(ChessBoardWriter board) {
        super.execute(board);
        ChessPiece king = fromPiece;

        board.remove(from);
        board.remove(fromRook);

        king.markMoved();
        rook.markMoved();

        board.put(to, king);
        board.put(toRook, rook);
    }
}
```

engine/move/ChessMove.java

```
package engine.move;

import engine.board.ChessBoardWriter;
import engine.piece.ChessPiece;
import engine.piece.Position;

/**
 * Represents a chess move from one position to another.
 * This is an abstract class that can be extended for specific move types such
 * as regular moves, captures, etc.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public abstract class ChessMove {
    protected final Position from;
    protected final Position to;
    protected final ChessPiece fromPiece;
```

```java
    /**
     * Constructs a ChessMove with the specified starting and ending positions.
     *
     * @param from      the starting position of the move
     * @param to        the destination position of the move
     * @param fromPiece the starting position chess piece
     */
    public ChessMove(Position from, Position to, ChessPiece fromPiece) {
        this.from = from;
        this.to = to;
        this.fromPiece = fromPiece.clone();
    }

    /**
     * Gets the starting position of the move.
     *
     * @return the starting position
     */
    public Position getFrom() {
        return from;
    }

    /**
     * Gets the destination position of the move.
     *
     * @return the destination position
     */
    public Position getTo() {
        return to;
    }

    /**
     * Gets the starting position chess piece.
     *
     * @return the starting position chess piece
     */
    public ChessPiece getFromPiece() {
        return fromPiece;
    }

    /**
     * Executes the move on the given chess board.
     * This method must be overridden by subclasses to define the specific behavior
     * of the move.
     *
     * @param board the chessboard on which the move is executed
     */
    public void execute(ChessBoardWriter board) {
        board.setLastMove(this);
    }

    /**
     * Returns a string representation of the move, including the class name and the
     * positions involved.
     *
     * @return a string representation of the move
     */
    @Override
    public String toString() {
        return getClass().getSimpleName() + " (" + from + " -> " + to + ")";
    }
}
```

engine/move/EnPassant.java

```java
package engine.move;

import engine.board.ChessBoardWriter;
import engine.piece.ChessPiece;
import engine.piece.Position;

/**
 * Represents an En Passant move in chess, a special pawn capture that occurs
 * when a pawn moves two squares forward from its starting position, and an
 * opposing pawn on an adjacent file captures it as if it had only moved one
 * square.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class EnPassant extends StandardMove {
    private final Position capturePawnPosition;

    /**
     * Constructs an En Passant move with the specified starting and destination
     * positions.
     *
     * @param from                the starting position of the capturing pawn
     * @param to                  the destination position where the capturing pawn
     *                            moves to
     * @param fromPiece           the starting position chess piece
     * @param capturePawnPosition the position of the captured pawn
     */
    public EnPassant(Position from, Position to, ChessPiece fromPiece, Position capturePawnPosition) {
        super(from, to, fromPiece);
        this.capturePawnPosition = capturePawnPosition;
    }

    /**
     * Executes the En Passant move on the provided chessboard. The capturing pawn
     * is moved to the destination square, and the captured pawn (which is bypassed
     * in the move) is removed from the board.
     *
     * @param board the chessboard on which the move is executed
     */
    @Override
    public void execute(ChessBoardWriter board) {
        super.execute(board);
        board.remove(capturePawnPosition);
    }
}
```

engine/move/LongCastling.java

```java
package engine.move;

import engine.piece.ChessPiece;
import engine.piece.Position;

/**
 * Represents a long (queenside) castling move in chess.
 * In this move, the king moves two squares towards the queenside rook,
 * and the rook moves three squares towards the center.
 *
 * @author Leonard Cseres
```

```
 * @author Aladin Iseni
 */
public final class LongCastling extends Castling {
    /**
     * Constructs a long castling move.
     *
     * @param from     the starting position of the king
     * @param to       the destination position of the king
     * @param king     the starting position king
     * @param fromRook the starting position of the rook
     * @param rook     the starting position rook
     */
    public LongCastling(Position from, Position to, ChessPiece king, Position fromRook, ChessPiece rook) {
        super(from, to, king, fromRook, fromRook.add(new Position(3, 0)), rook);
    }
}
```

engine/move/Moves.java

```
package engine.move;

import engine.piece.Position;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

/**
 * Represents a collection of chess moves, storing them in a map with the
 * destination position as the key.
 * Provides methods for adding, extending, and retrieving moves.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class Moves {
    private final Map<Position, ChessMove> movesMap;

    /**
     * Constructs an empty Moves object to hold chess moves.
     */
    public Moves() {
        movesMap = new HashMap<>();
    }

    /**
     * Adds a move to the collection of moves.
     *
     * @param move the chess move to be added
     */
    public void addMove(ChessMove move) {
        movesMap.put(move.getTo(), move);
    }

    /**
     * Extends the current collection of moves by adding all moves from another
     * Moves object.
     *
     * @param moves the Moves object whose moves should be added
     */
    public void extendMoves(Moves moves) {
        for (ChessMove move : moves.getAllMoves()) {
            this.addMove(move);
```

```
        }
    }

    /**
     * Retrieves a move based on its destination position.
     *
     * @param to the destination position of the move
     * @return the chess move associated with the destination position, or null if
     * no such move exists
     */
    public ChessMove getMove(Position to) {
        return movesMap.get(to);
    }

    /**
     * Retrieves all moves in the collection.
     *
     * @return a collection of all chess moves
     */
    public Collection<ChessMove> getAllMoves() {
        return movesMap.values();
    }

    /**
     * Returns a string representation of all moves in the collection.
     *
     * @return a string representing all moves
     */
    @Override
    public String toString() {
        return movesMap.values().toString();
    }
}
```

engine/move/Promotion.java

```
package engine.move;

import engine.board.ChessBoardWriter;
import engine.piece.ChessPiece;
import engine.piece.Position;

/**
 * Represents a promotion move in chess, where a pawn reaches the last rank and
 * is promoted
 * to a more powerful piece (Queen, Rook, Bishop, or Knight).
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class Promotion extends StandardMove {

    /**
     * Constructs a Promotion move with the specified starting and destination
     * positions.
     *
     * @param from the starting position of the pawn
     * @param to   the destination position where the pawn will be promoted
     * @param pawn the starting position pawn
     */
    public Promotion(Position from, Position to, ChessPiece pawn) {
        super(from, to, pawn);
    }
```

```
    /**
     * Executes the promotion move on the provided chess board. The pawn is moved
     * from its starting position
     * to the destination, and the pawn is promoted to a new piece (Queen, Rook,
     * Bishop, or Knight).
     *
     * @param board the chessboard on which the move is executed
     */
    @Override
    public void execute(ChessBoardWriter board) {
        super.execute(board);
        board.handlePawnPromotion(to);
    }
}
```

engine/move/PromotionWithCapture.java

```
package engine.move;

import engine.board.ChessBoardWriter;
import engine.piece.ChessPiece;
import engine.piece.Position;

/**
 * Represents a promotion move with capture in chess, where a pawn captures an
 * opponent's piece
 * and is promoted to a more powerful piece (Queen, Rook, Bishop, or Knight).
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class PromotionWithCapture extends Capture {

    /**
     * Constructs a PromotionWithCapture move with the specified starting and
     * destination positions.
     *
     * @param from the starting position of the pawn
     * @param to   the destination position where the pawn will capture and be
     *             promoted
     * @param pawn the starting position pawn
     */
    public PromotionWithCapture(Position from, Position to, ChessPiece pawn) {
        super(from, to, pawn);
    }

    /**
     * Executes the promotion with capture move on the provided chess board. The
     * pawn captures an opponent's
     * piece at the destination position, and then the pawn is promoted to a new
     * piece (Queen, Rook, Bishop, or Knight).
     *
     * @param board the chessboard on which the move is executed
     */
    @Override
    public void execute(ChessBoardWriter board) {
        super.execute(board);
        board.handlePawnPromotion(to);
    }
}
```

engine/move/ShortCastling.java

```java
package engine.move;

import engine.piece.ChessPiece;
import engine.piece.Position;

/**
 * Represents a short (kingside) castling move in chess.
 * In this move, the king moves two squares towards the kingside rook,
 * and the rook moves two squares towards the center.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class ShortCastling extends Castling {
    /**
     * Constructs a short castling move.
     *
     * @param from     the starting position of the king
     * @param to       the destination position of the king
     * @param king     the starting position king
     * @param fromRook the starting position of the rook
     * @param rook     the starting position rook
     */
    public ShortCastling(Position from, Position to, ChessPiece king, Position fromRook, ChessPiece rook) {
        super(from, to, king, fromRook, fromRook.sub(new Position(2, 0)), rook);
    }
}
```

engine/move/StandardMove.java

```java
package engine.move;

import engine.board.ChessBoardWriter;
import engine.piece.ChessPiece;
import engine.piece.Position;

/**
 * Represents a standard move in chess, where a piece is moved from one position
 * to another.
 * The piece is removed from the starting position and placed at the destination
 * position.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public class StandardMove extends ChessMove {

    /**
     * Constructs a StandardMove with the specified starting and ending positions.
     *
     * @param from      the starting position of the move
     * @param to        the destination position of the move
     * @param fromPiece the starting position chess piece
     */
    public StandardMove(Position from, Position to, ChessPiece fromPiece) {
        super(from, to, fromPiece);
    }

    /**
     * Executes the move on the provided chess board.
```

```
     * The piece is moved from the starting position to the destination position,
     * and it is marked as moved.
     *
     * @param board the chessboard on which the move is executed
     */
    @Override
    public void execute(ChessBoardWriter board) {
        super.execute(board);
        board.remove(from);
        fromPiece.markMoved();
        board.put(to, fromPiece);
    }
}
```

engine/piece/Bishop.java

```
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
import engine.generator.Direction;
import engine.generator.DirectionalGenerator;

/**
 * Represents the Bishop chess piece.
 * The Bishop can move diagonally any number of squares in any diagonal
 * direction.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class Bishop extends PromotableChessPiece {

    /**
     * Constructs a Bishop chess piece with the specified color.
     * Uses a {@link DirectionalGenerator} limited to diagonal movements.
     *
     * @param color the color of the Bishop
     */
    public Bishop(PlayerColor color) {
        super(PieceType.BISHOP, color, new DirectionalGenerator(Direction.DIAGONAL));
    }
}
```

engine/piece/ChessPiece.java

```
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
import engine.board.ChessBoardReader;
import engine.generator.MoveGenerator;
import engine.move.Moves;

import java.util.ArrayList;
import java.util.List;

/**
 * Represents a chess piece with associated type, color, and movement
 * generators.
```

```java
 * Provides functionality to track movement and generate possible moves.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public abstract class ChessPiece implements Cloneable {
    protected final PieceType type;
    protected final PlayerColor color;
    private List<MoveGenerator> generators;
    private boolean hasMoved = false;

    /**
     * Constructs a chess piece with specified type, color, and movement generators.
     *
     * @param type       the type of the chess piece
     * @param color      the color of the chess piece
     * @param generators the movement generators defining how the piece moves
     */
    public ChessPiece(PieceType type, PlayerColor color, MoveGenerator... generators) {
        this.type = type;
        this.color = color;
        this.generators = List.of(generators);
    }

    /**
     * Gets the type of the chess piece.
     *
     * @return the {@link PieceType} of the chess piece
     */
    public PieceType getType() {
        return type;
    }

    /**
     * Gets the color of the chess piece.
     *
     * @return the {@link PlayerColor} of the chess piece
     */
    public PlayerColor getColor() {
        return color;
    }

    /**
     * Determines if another chess piece is an opponent.
     *
     * @param other the other chess piece
     * @return true if the other piece is an opponent, false otherwise
     */
    public boolean isOpponent(ChessPiece other) {
        return color != other.color;
    }

    /**
     * Marks the piece has moved.
     */
    public void markMoved() {
        this.hasMoved = true;
    }

    /**
     * Checks if the piece has moved at least once.
     *
     * @return true if the piece has moved, false otherwise
     */
    public boolean hasMoved() {
```

```
        return hasMoved;
    }

    /**
     * Generates all possible moves for the chess piece from a given position on the
     * board.
     *
     * @param board the current state of the chessboard
     * @param from  the position of the piece on the chessboard
     * @return a {@link Moves} object containing all possible moves
     */
    public Moves getPossibleMoves(ChessBoardReader board, Position from) {
        Moves moves = new Moves();
        for (MoveGenerator gen : generators) {
            moves.extendMoves(gen.generate(board, from));
        }
        return moves;
    }

    /**
     * Creates a deep clone of the chess piece, preserving its movement state.
     *
     * @return a cloned instance of the chess piece
     * @throws AssertionError if the clone failed. We assert it won't happen
     */
    @Override
    public ChessPiece clone() {
        try {
            ChessPiece clonedPiece = (ChessPiece) super.clone();
            clonedPiece.hasMoved = this.hasMoved;

            List<MoveGenerator> clonedGenerators = new ArrayList<>();
            for (MoveGenerator generator : this.generators) {
                clonedGenerators.add(generator.clone());
            }
            clonedPiece.generators = clonedGenerators;

            return clonedPiece;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError("Cloning failed", e);
        }
    }
}
```

engine/piece/King.java

```
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
import engine.board.ChessBoardReader;
import engine.generator.Direction;
import engine.generator.DirectionalGenerator;
import engine.generator.DistanceGenerator;
import engine.move.LongCastling;
import engine.move.Moves;
import engine.move.ShortCastling;

/**
 * Represents the King chess piece.
 * The King can move one square in any direction, as defined by the movement
 * rules.
 *
```

```java
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class King extends ChessPiece {

    /**
     * Constructs a King chess piece with the specified color.
     * Uses a {@link DistanceGenerator} limited to one square and all directions.
     *
     * @param color the color of the King
     */
    public King(PlayerColor color) {
        super(PieceType.KING, color, new DistanceGenerator(1, new DirectionalGenerator(Direction.ALL)));
    }

    /**
     * Gets all possible moves for the King from the given position.
     * Handles regular moves and castling moves.
     *
     * @param board the chess board
     * @param from  the starting position of the King
     * @return a {@link Moves} object containing all valid moves for the King
     */
    @Override
    public Moves getPossibleMoves(ChessBoardReader board, Position from) {
        Moves moves = super.getPossibleMoves(board, from);
        moves.extendMoves(getCastlingMoves(board, from));

        return moves;
    }

    /**
     * Calculates the possible castling moves for the king from the given position.
     *
     * @param board the chessboard used to evaluate castling conditions
     * @param from  the current position of the king
     * @return a Moves object containing valid castling moves, or empty if
     *         no castling is possible
     */
    private Moves getCastlingMoves(ChessBoardReader board, Position from) {
        Moves castlingMoves = new Moves();

        Position shortCastlingPosition = new Position(from.x() + 2, from.y());
        if (canCastle(board, from, shortCastlingPosition)) {
            Position rookPosition = getRookPosition(from, Direction.RIGHT);
            ChessPiece rook = board.get(rookPosition);
            castlingMoves.addMove(new ShortCastling(from, shortCastlingPosition, this, rookPosition, rook));
        }

        Position longCastlingPosition = new Position(from.x() - 2, from.y());
        if (canCastle(board, from, longCastlingPosition)) {
            Position rookPosition = getRookPosition(from, Direction.LEFT);
            ChessPiece rook = board.get(rookPosition);
            castlingMoves.addMove(new LongCastling(from, longCastlingPosition, this, rookPosition, rook));
        }
        return castlingMoves;
    }

    /**
     * Determines if the King can castle with the Rook at the given positions.
     * The King and Rook must not have moved, the squares between them must be
     * empty and not attacked, and the King must not currently be in check.
     *
     * @param board the chess board
     * @param from  the position of the King
```

```
     * @param to    the target position for the King (castling destination)
     * @return true if the King can castle, false otherwise
     */
    private boolean canCastle(ChessBoardReader board, Position from, Position to) {
        if (hasMoved()) {
            return false;
        }

        Direction direction = to.x() > from.x() ? Direction.RIGHT : Direction.LEFT;
        Position rookPosition = getRookPosition(from, direction);
        ChessPiece rook = board.get(rookPosition);

        return isValidRook(rook) &&
                areSquaresBetweenEmptyAndSafe(board, from, rookPosition, direction) &&
                !board.isSquareAttacked(from, color, null);
    }

    /**
     * Calculates the position of the Rook based on the King's position and the
     * castling direction.
     *
     * @param from      the starting position of the King
     * @param direction the direction of castling (RIGHT or LEFT)
     * @return the position of the Rook involved in castling
     */
    private Position getRookPosition(Position from, Direction direction) {
        return direction == Direction.RIGHT
                ? new Position(Position.MAX_X, from.y())
                : new Position(0, from.y());
    }

    /**
     * Checks if the Rook at the given position is valid for castling.
     * The Rook must exist, must not have moved, and must be of type ROOK.
     *
     * @param rook the chess piece at the Rook's position
     * @return true if the Rook is valid for castling, false otherwise
     */
    private boolean isValidRook(ChessPiece rook) {
        return rook != null && !rook.hasMoved() && rook.getType() == PieceType.ROOK;
    }

    /**
     * Checks if the squares between the King and the Rook are both empty and not
     * attacked.
     *
     * @param board     the chess board
     * @param from      the position of the King
     * @param rookPos   the position of the Rook
     * @param direction the direction of castling (RIGHT or LEFT)
     * @return true if the squares between are empty and safe, false otherwise
     */
    private boolean areSquaresBetweenEmptyAndSafe(ChessBoardReader board, Position from, Position rookPos,
            Direction direction) {
        Position current = direction.add(from, color);
        while (!current.equals(rookPos)) {
            if (board.containsKey(current) || board.isSquareAttacked(current, color, PieceType.KING)) {
                return false;
            }
            current = direction.add(current, color);
        }
        return true;
    }
}
```

engine/piece/Knight.java

```java
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
import engine.generator.KnightGenerator;

/**
 * Represents the Knight chess piece.
 * The Knight moves in an "L" shape: two squares in one direction and then one
 * square perpendicular, or vice versa.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class Knight extends PromotableChessPiece {

    /**
     * Constructs a Knight chess piece with the specified color.
     * Uses a {@link KnightGenerator} to define its movement pattern.
     *
     * @param color the color of the Knight
     */
    public Knight(PlayerColor color) {
        super(PieceType.KNIGHT, color, new KnightGenerator());
    }
}
```

engine/piece/Pawn.java

```java
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
import engine.board.ChessBoardReader;
import engine.generator.Direction;
import engine.generator.DirectionalGenerator;
import engine.generator.DistanceGenerator;
import engine.generator.PawnDistanceGenerator;
import engine.move.Capture;
import engine.move.ChessMove;
import engine.move.EnPassant;
import engine.move.Moves;
import engine.move.Promotion;
import engine.move.PromotionWithCapture;
import engine.move.StandardMove;

/**
 * Represents the Pawn chess piece.
 * The Pawn can move one or two squares forward, but captures diagonally.
 * It also has the option to promote upon reaching the opposite end of the
 * board.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class Pawn extends ChessPiece {

    /**
     * Constructs a Pawn chess piece with the specified color.
     * Uses a {@link PawnDistanceGenerator} for forward movement and a
```

```java
     * {@link DirectionalGenerator}
     * for diagonal captures.
     *
     * @param color the color of the Pawn
     */
    public Pawn(PlayerColor color) {
        super(PieceType.PAWN, color, new PawnDistanceGenerator(), new DistanceGenerator(1,
                new DirectionalGenerator(Direction.FORWARDS_LEFT, Direction.FORWARDS_RIGHT)));
    }

    /**
     * Gets all possible moves for the Pawn from the given position.
     * Handles regular moves, captures, and promotions (including promotion with
     * capture).
     *
     * @param board the chess board
     * @param from  the starting position of the Pawn
     * @return a {@link Moves} object containing all valid moves for the Pawn
     */
    @Override
    public Moves getPossibleMoves(ChessBoardReader board, Position from) {
        Moves candidateMoves = super.getPossibleMoves(board, from);
        Moves validMoves = new Moves();

        for (ChessMove move : candidateMoves.getAllMoves()) {
            Position to = move.getTo();
            if (isValidMove(board, from, to)) {
                validMoves.addMove(createAppropriateMove(from, to));
            }
        }

        addEnPassantMoves(board, from, validMoves);
        return validMoves;
    }

    /**
     * Validates whether a move is legal according to pawn movement rules.
     * Checks both forward moves and diagonal captures.
     *
     * @param board The current state of the chess board
     * @param from  The starting position of the pawn
     * @param to    The target position for the move
     * @return true if the move is legal, false otherwise
     */
    private boolean isValidMove(ChessBoardReader board, Position from, Position to) {
        if (isDiagonalMove(from, to)) {
            // Capture
            return board.containsKey(to) && board.get(to).isOpponent(this);
        }
        return !board.containsKey(to); // Forward moves require empty square
    }

    /**
     * Creates the appropriate type of move based on the movement type and position.
     * Handles standard moves, captures, and promotions.
     *
     * @param from The starting position of the pawn
     * @param to   The target position for the move
     * @return The appropriate ChessMove object for the given move
     */
    private ChessMove createAppropriateMove(Position from, Position to) {
        if (isDiagonalMove(from, to)) {
            return createCaptureMove(from, to);
        }
        return createForwardMove(from, to);
```

```
}

/**
 * Creates a capture move, either as a regular capture or a promotion with
 * capture.
 *
 * @param from The starting position of the pawn
 * @param to   The target position for the capture
 * @return A Capture or PromotionWithCapture move
 */
private ChessMove createCaptureMove(Position from, Position to) {
    return isAtPromotionRank(to)
            ? new PromotionWithCapture(from, to, this)
            : new Capture(from, to, this);
}

/**
 * Creates a forward move, either as a standard move or a promotion.
 *
 * @param from The starting position of the pawn
 * @param to   The target position for the move
 * @return A StandardMove or Promotion move
 */
private ChessMove createForwardMove(Position from, Position to) {
    return isAtPromotionRank(to)
            ? new Promotion(from, to, this)
            : new StandardMove(from, to, this);
}

/**
 * Adds any possible en passant captures to the list of valid moves.
 * Checks adjacent squares for opponent pawns that have just moved two squares.
 *
 * @param board The current state of the chess board
 * @param from  The current position of the pawn
 * @param moves The collection of moves to add to
 */
private void addEnPassantMoves(ChessBoardReader board, Position from, Moves moves) {
    Position[] adjacentPositions = {
            Direction.LEFT.add(from, color),
            Direction.RIGHT.add(from, color)
    };
    for (Position adjacent : adjacentPositions) {
        if (isValidEnPassantPosition(board, adjacent)) {
            Position captureSquare = Direction.FORWARDS.add(adjacent, color);
            moves.addMove(new EnPassant(from, captureSquare, this, adjacent));
        }
    }
}

/**
 * Checks if an en passant capture is valid from the given position.
 * Validates that there is an opponent's pawn in the correct position
 * and that it just moved two squares forward.
 *
 * @param board    The current state of the chess board
 * @param adjacent The position adjacent to the pawn
 * @return true if an en passant capture is possible, false otherwise
 */
private boolean isValidEnPassantPosition(ChessBoardReader board, Position adjacent) {
    if (!isPawnAtPosition(board, adjacent))
        return false;
    ChessMove lastMove = board.getLastMove();
    return lastMove != null &&
            wasDoublePawnAdvance(lastMove) &&
```

```
                adjacent.equals(lastMove.getTo()) &&
                board.get(adjacent).isOpponent(this);
    }

    /**
     * Determines if a move is a diagonal based on the positions.
     *
     * @param from The starting position
     * @param to   The target position
     * @return true if the move is diagonal, false otherwise
     */
    private boolean isDiagonalMove(Position from, Position to) {
        Position delta = from.sub(to).abs();
        return delta.x() == delta.y();
    }

    /**
     * Checks if there is a pawn at the given position.
     *
     * @param board The current state of the chess board
     * @param pos   The position to check
     * @return true if there is a pawn at the position, false otherwise
     */
    private boolean isPawnAtPosition(ChessBoardReader board, Position pos) {
        return board.containsKey(pos) && board.get(pos).getType() == PieceType.PAWN;
    }

    /**
     * Determines if a move was a double square pawn advance.
     *
     * @param move The move to check
     * @return true if the move was a double square advance, false otherwise
     */
    private boolean wasDoublePawnAdvance(ChessMove move) {
        return Math.abs(move.getFrom().y() - move.getTo().y()) == 2;
    }

    /**
     * Checks if a position is on the promotion rank for this pawn's color.
     * White pawns promote on rank 8 (MAX_Y), black pawns promote on rank 1 (0).
     *
     * @param pos The position to check
     * @return true if the position is on the promotion rank, false otherwise
     */
    private boolean isAtPromotionRank(Position pos) {
        return color == PlayerColor.WHITE
                ? pos.y() == Position.MAX_Y
                : pos.y() == 0;
    }
}
```

engine/piece/Position.java

```
package engine.piece;

import chess.PlayerColor;

/**
 * Represents a position on the chessboard with x and y coordinates.
 * Provides utility methods for position validation and arithmetic operations.
 *
 * @param x the x-coordinate (column) of the position
 * @param y the y-coordinate (row) of the position
```

```java
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public record Position(int x, int y) {
    public static final int MAX_X = 7;
    public static final int MAX_Y = 7;

    /**
     * Checks if the position is within the bounds of the chessboard.
     *
     * @return true if the position is valid, false otherwise
     */
    public boolean isValid() {
        return x >= 0 && y >= 0 && x <= MAX_X && y <= MAX_Y;
    }

    /**
     * Calculates the chessboard-compatible distance to another position.
     * The distance is defined as the maximum of the horizontal or vertical steps.
     *
     * @param other the other position to calculate the distance to
     * @return the maximum of horizontal or vertical steps to the other position
     */
    public int dist(Position other) {
        int dx = Math.abs(x - other.x);
        int dy = Math.abs(y - other.y);
        return Math.max(dx, dy);
    }

    /**
     * Adds the coordinates of another position to this position.
     *
     * @param other the position to add
     * @return a new {@link Position} representing the sum
     */
    public Position add(Position other) {
        return new Position(x + other.x, y + other.y);
    }

    /**
     * Subtracts the coordinates of another position from this position.
     *
     * @param other the position to subtract
     * @return a new {@link Position} representing the difference
     */
    public Position sub(Position other) {
        return new Position(x - other.x, y - other.y);
    }

    /**
     * Converts the position's coordinates to their absolute values.
     *
     * @return a new {@link Position} with absolute x and y coordinates
     */
    public Position abs() {
        return new Position(Math.abs(x), Math.abs(y));
    }

    /**
     * Gets the position color
     *
     * @return WHITE if the position is on a white square, BLACK otherwise
     */
    public PlayerColor getColor() {
        return (x + y) % 2 == 0 ? PlayerColor.BLACK : PlayerColor.WHITE;
```

```
    }

    /**
     * Provides a string representation of the position in the format "(x, y)".
     *
     * @return a string representation of the position
     */
    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

engine/piece/PromotableChessPiece.java

```
package engine.piece;

import chess.ChessView;
import chess.PieceType;
import chess.PlayerColor;
import engine.generator.MoveGenerator;

/**
 * Represents a promotable chess piece (e.g., pawn promotion) that can be chosen
 * by the user during gameplay.
 * Extends {@link ChessPiece} and implements {@link ChessView.UserChoice}.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public abstract class PromotableChessPiece extends ChessPiece implements ChessView.UserChoice {
    /**
     * Constructs a promotable chess piece with a specified type, color, and move
     * generators.
     *
     * @param type           the type of the promotable chess piece
     * @param color          the color of the chess piece
     * @param validationList the move generators for the piece
     */
    public PromotableChessPiece(PieceType type, PlayerColor color, MoveGenerator... validationList) {
        super(type, color, validationList);
    }

    /**
     * Provides a string representation of the piece's type for display purposes.
     *
     * @return the string value of the piece's {@link PieceType}
     */
    @Override
    public String textValue() {
        return type.toString();
    }
}
```

engine/piece/Queen.java

```
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
```

```java
import engine.generator.Direction;
import engine.generator.DirectionalGenerator;

/**
 * Represents the Queen chess piece.
 * The Queen can move any number of squares in any direction: horizontally,
 * vertically, or diagonally.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class Queen extends PromotableChessPiece {

    /**
     * Constructs a Queen chess piece with the specified color.
     * The Queen moves in all directions (horizontal, vertical, and diagonal) with
     * no restrictions
     * on the number of squares.
     *
     * @param color the color of the Queen
     */
    public Queen(PlayerColor color) {
        super(PieceType.QUEEN, color, new DirectionalGenerator(Direction.ALL));
    }
}
```

engine/piece/Rook.java

```java
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
import engine.generator.Direction;
import engine.generator.DirectionalGenerator;

/**
 * Represents the Rook chess piece.
 * The Rook can move any number of squares horizontally or vertically.
 *
 * @author Leonard Cseres
 * @author Aladin Iseni
 */
public final class Rook extends PromotableChessPiece {

    /**
     * Constructs a Rook chess piece with the specified color.
     * The Rook moves in straight lines either horizontally or vertically with no
     * restrictions
     * on the number of squares.
     *
     * @param color the color of the Rook
     */
    public Rook(PlayerColor color) {
        super(PieceType.ROOK, color, new DirectionalGenerator(Direction.STRAIGHT));
    }
}
```