Rapport du Laboratoire 07 - Chess

1. Introduction

Ce document explique le travail réalisé sur le laboratoire 07 Chess. Il montre les choix de conception, les tests effectués et les résultats obtenu.

---

2. En-têtes et commentaires

Bonnes pratiques respectées

- **Commentaires clairs** : Toutes les classes et méthodes ont des explications simples en format Javadoc. Sauf celles dont le nom est suffisemment explicite.
- **Exceptions** : Si une méthode peut provoquer une erreur, c'est indiqué avec `@throws`.
- **Cas spéciaux** : Les retours ou comportements spécifiques sont détaillés.

---

3. Méthodes, attributs et variables

Noms faciles à comprendre

- **Classes** : Noms en CamelCase avec une majuscule au début (ex. `King`, `Pawn`, `ChessGame`).
- **Méthodes et variables** : Noms clairs comme `moveTo`, `getGraphicalType`.
- **Constantes** : Écrites en majuscules avec des underscores. Par exemple : `DEFAULT_WIDTH`.

Sécurité et clarté

- **Visibilité** : Les attributs sont privés et accessibles via des get/set.
- **Validation** : Les entrées des méthodes sont contrôlées pour éviter des erreurs.

---

4. Tests réalisés

Les tests suivants ont été effectués:

Fonctionnalité	Résultat
Pion: 1er mouvement à deux cases	OK
Pion: prise d'une pièce adverse	OK
Pion: prise en passant	OK
Pion: prise en passant contre un pion qui a bougé 2x	Bloqué
Pion: prise en passant contre une pièce autre qu'un pion	Bloqué
Pion: déplacement en arrière	Bloqué
Pion: promotion	OK

Fonctionnalité	Résultat
Pion: promotion en une 3ème tour	OK
Petit roque	OK
Grand roque	OK
Roque en échec	Bloqué
Roque avec 1+ case du chemin menacée	Bloqué
Roque avec la tour ou le roi qui a déjà bougé	Bloqué
Roi: déplacement de 2+ cases	Bloqué
Roi: déplacement sur une case menacée	Bloqué
Général: déplacement standard	OK
Général: déplacement sur une pièce amie	Bloqué
Général: déplacement lors du tour adverse	Bloqué
Général: déplacement qui met le roi ami en échec	Bloqué
Général (sauf cavalier): déplacement obstrué par une pièce	Bloqué

5. Diagramme UML

Un diagramme UML a été créé pour montrer les relations entre les classes. Il se trouve en annexe.

6. Choix de conception

Organisation orientée objet

- **Abstraction** : Les classes abstraites comme `Piece` simplifient l'ajout de nouvelles pièces.
- **Responsabilités** : Chaque classe a un role précis. Par exemple : `Board` pour gérer le plateau, `ChessGame` pour contrôler le jeu.
- **Flexibilité** : Les mouvements sont gérés avec une interface `Movement`, ce qui permet d'ajouter des types de déplacement facilement.

Mouvement de pièces

Les mouvements sont gérés avec une interface `Movement`, ce qui permet d'ajouter des types de déplacement facilement, mais surtout de partager des types de mouvements entre les pièces. Ainsi le mouvement "diagonal" est partagé entre le fou, la dame, et le roi.

Actions liées à l'état du plateau

Les actions liées à l'état du plateau, comme le fait de ne pas bouger une pièce alors que cela mettrait son roi en échec, cela a été implémenté côté Board.java, car celà nécessite une vue d'ensemble d'une partie.

Classe FirstMovePiece.java

---

7. Conclusion

Le projet respecte les bonnes pratiques du développement Java et les consignes données. Les tests montrent que toutes les fonctionnalités fonctionnent correctement. Le diagramme UML est clair et correspond au code.

---

8. Code source

ChessGame.java

```
package engine;

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

/**
 * Manages the chess game, starting the game, handling moves,
 * and updating the view.
 */
public class ChessGame implements ChessController {
    private ChessView view;
    private PlayerColor colorPlaying = PlayerColor.WHITE;
    private Board board;

    /**
     * Sets up the chess game and shows the chessboard on the screen.
     *
     * @param view the screen display for the chess game
     */
    @Override
    public void start(ChessView view) {
        this.board = new Board();

        this.view = view;
        view.startView();

        updateView();
    }

    /**
     * Moves a piece on the board from one position to another.
     *
     * @param fromX the starting column
     * @param fromY the starting row
     * @param toX   the target column
     * @param toY   the target row
     * @return true if the move is valid and false otherwise
```

```
     */
    @Override
    public boolean move(int fromX, int fromY, int toX, int toY) {
        Coordinates from = new Coordinates(fromX, fromY);
        Coordinates to = new Coordinates(toX, toY);

        Piece movingPiece = board.getPieceAt(from);
        if (!board.move(from, to, colorPlaying)) return false;
        colorPlaying = colorPlaying.toggle();

        // Pawn promotion
        if (movingPiece instanceof Pawn && (to.y() == 0 || to.y() == 7)) {
            PieceUserChoice choice = view.askUser("Promotion", "Promotion choice",
                new PieceUserChoice(new Knight(movingPiece.getColor(), new
Coordinates(toX, toY))),
                new PieceUserChoice(new Bishop(movingPiece.getColor(), new
Coordinates(toX, toY))),
                new PieceUserChoice(new Rook(movingPiece.getColor(), new
Coordinates(toX, toY))),
                new PieceUserChoice(new Queen(movingPiece.getColor(), new
Coordinates(toX, toY)))
            );

            board.removePiece(movingPiece);
            board.addPiece(choice.piece(),true);
        }
        updateView();

        return true;
    }

    /**
     * Starts a new game by resetting the board and adding all pieces to their
starting positions.
     */
    @Override
    public void newGame() {
        board = new Board();
        colorPlaying = PlayerColor.WHITE;

        int pieceStartRow;
        int pawnStartRow;
        for (PlayerColor color : PlayerColor.values()) {
            if (color == PlayerColor.WHITE) {
                pieceStartRow = 0;
                pawnStartRow = 1;
            } else {
                pieceStartRow = 7;
                pawnStartRow = 6;
            }

            board.addPiece(new Rook(color, new Coordinates(0, pieceStartRow)));
            board.addPiece(new Rook(color, new Coordinates(7, pieceStartRow)));
            board.addPiece(new Knight(color, new Coordinates(6, pieceStartRow)));
```

```
            board.addPiece(new Knight(color, new Coordinates(1, pieceStartRow)));
            board.addPiece(new Bishop(color, new Coordinates(2, pieceStartRow)));
            board.addPiece(new Bishop(color, new Coordinates(5, pieceStartRow)));
            board.addPiece(new Queen(color, new Coordinates(3, pieceStartRow)));
            board.addPiece(new King(color, new Coordinates(4, pieceStartRow)));

            for (int i = 0; i < 8; ++i) {
                board.addPiece(new Pawn(color, new Coordinates(i, pawnStartRow)));
            }
        }

        updateView();
    }

    /**
     * Updates the chessboard display to show the current state of the game.
     *
     */
    private void updateView() {
        for (int i = 0; i < 8; ++i) {
            for (int j = 0; j < 8; ++j) {
                Piece p = board.getPieceAt(new Coordinates(i, j));
                if (p == null) {
                    this.view.removePiece(i, j);
                } else {
                    this.view.putPiece(p.getGraphicalType(), p.getColor(), i, j);
                }
            }
        }

        if (board.isChecked()) view.displayMessage("Check !");
        else view.displayMessage("");
    }

    /**
     * Represents the player's choice of pieces when promoting a pawn.
     */
    record PieceUserChoice(Piece piece) implements ChessView.UserChoice {

        @Override
        public String textValue() {
            return piece.toString();
        }
    }
}
```

Board.java

```
package engine;

import chess.PlayerColor;
```

```java
import engine.piece.King;
import engine.piece.Knight;
import engine.piece.Pawn;
import engine.piece.Piece;
import engine.piece.Rook;

import java.util.LinkedList;
import java.util.List;

/**
 * Represents the chessboard, manages the state, pieces, and movement.
 * Handles castling, en passant, and checking for checks.
 */
public class Board {
    private static final int WHITE = PlayerColor.WHITE.ordinal();
    private static final int BLACK = PlayerColor.BLACK.ordinal();
    private static final int DEFAULT_WIDTH = 8;
    private static final int DEFAULT_HEIGHT = 8;
    private static final int CASTLE_DIST = 2;

    private final int width;
    private final int height;

    private final King[] kings = new King[2];    // to quickly get the positions
of the kings when needed
    private final Rook[][] castlableRooks = new Rook[2][2];


    private boolean check = false;

    private final List<List<Piece>> pieces = List.of(
            new LinkedList<>(), // white pieces
            new LinkedList<>()    // black pieces
    );

    /**
     * Creates a chessboard with a width and height.
     *
     * @param width  the width of the board
     * @param height the height of the board
     */
    public Board(int width, int height) {
        this.width = width;
        this.height = height;
    }

    /**
     * Creates a chessboard with the default dimensions of 8x8.
     */
    public Board() {
        this(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }

    /**
```

```
     * Adds a piece to the board and updates the board state accordingly.
     *
     * @param piece the piece to be added to the board
     */
    public void addPiece(Piece piece) {
        addPiece(piece, false);
    }

    /**
     * Adds a piece to the board and updates the board state accordingly.
     *
     * @param piece       the piece to be added to the board
     * @param gameStarted will indicate if the game started or not, so that board
knows if piece is added due to promotion
     *                    or not
     */
    public void addPiece(Piece piece, boolean gameStarted) {
        pieces.get(piece.getColor().ordinal()).add(piece);
        if (gameStarted) {
            return;
        }
        if (piece instanceof King)
            kings[piece.getColor().ordinal()] = (King) piece;
        if (piece instanceof Rook)
            add(castlableRooks[piece.getColor().ordinal()], piece);
    }

    /**
     * Removes a piece from the board.
     *
     * @param piece the piece to be removed from the board
     */
    public void removePiece(Piece piece) {
        pieces.get(piece.getColor().ordinal()).remove(piece);
    }

    /**
     * Tries to move a piece from "from" to "dest"
     *
     * @param from         start coordinates
     * @param to           destination coordinates
     * @param colorPlaying boolean representing if white is to play
     * @return boolean representing whether the piece was moved or not
     */
    public boolean move(Coordinates from, Coordinates to, PlayerColor
colorPlaying) {
        Piece p = getPieceAt(from);
        Piece target = getPieceAt(to);

        boolean movementWasValid = isMovementValid(p, target, from, to,
colorPlaying);
        if (!movementWasValid) return false;

        // Castle
```

```
        if (p instanceof King king && (to.equals(from.move(CASTLE_DIST, 0)) ||
to.equals(from.move(-CASTLE_DIST, 0)))) {
            // we detected that king is trying to castle

            int rookId = to.x() < king.getCoordinates().x() ? 0 : 1;
            Rook rook = castlableRooks[colorPlaying.ordinal()][rookId];

            if (!castle(king, rook)) {
                return false;
            }
            return true;
        } else {
            // Handle en passant for pawns
            if (isEnPassantCapture(from, to)) {
                Coordinates enPassantCapturePos = new Coordinates(to.x(),
from.y());
                Piece enPassantTarget = getPieceAt(enPassantCapturePos);
                removePiece(enPassantTarget);
            } else {
                resetEnPassantFlags(colorPlaying.ordinal() == WHITE ? BLACK :
WHITE);
            }

            // Normal move
            if (target != null)
                target.moveTo(new Coordinates(-1, -1));
            p.moveTo(to);

            // Control if any opponent piece can capture the king (check for pins)
            Coordinates playingKingCoordinates =
kings[colorPlaying.ordinal()].getCoordinates();
            if (verifyCheck(colorPlaying.toggle(), playingKingCoordinates)) {
                // Cancel move
                if (target != null)
                    target.moveTo(to);
                p.moveTo(from);

                return false;
            }

            // Remove targeted piece, if any
            if (target != null) {
                pieces.get(target.getColor().ordinal()).remove(target);
            }

            check = false;
        }

        // Control if opponent King is checked or not
        Coordinates opponentKingCoordinates =
kings[colorPlaying.toggle().ordinal()].getCoordinates();
        check = verifyCheck(colorPlaying, opponentKingCoordinates);

        return true;
```

```
    }

    /**
     * Verifies that path between a coordinate to another is obstructed
     *
     * @param from initial coordinates
     * @param dest destination coordinates
     * @return boolean that shows is the path is obstructed
     * @throws ArrayIndexOutOfBoundsException when the given position is out of
the board
     */
    private boolean isPathObstructed(Coordinates from, Coordinates dest) {
        if (from == null || dest == null) throw new NullPointerException();
        if (from.equals(dest)) return false;

        int dx = (int) Math.signum(dest.x() - from.x());
        int dy = (int) Math.signum(dest.y() - from.y());

        // * infinite loop here
        for (Coordinates it = from.move(dx, dy); isInBoundaries(it) &&
!it.equals(dest); it = it.move(dx, dy)) {
            if (getPieceAt(it) != null) return true;
        }

        return false;
    }

    /**
     * Get the piece located at the specified position.
     *
     * @param pos the coordinates of the position
     * @return the piece at the specified position, or null if no piece is present
     * @throws IllegalArgumentException if the position is out of the board
boundaries
     */
    public Piece getPieceAt(Coordinates pos) {
        if (pos == null) throw new NullPointerException("Coordinates cannot be
null");
        if (!isInBoundaries(pos))
            throw new IllegalArgumentException(String.format("Invalid coordinates
%s.", pos));
        for (Piece p : pieces.get(WHITE)) {
            if (pos.equals(p.getCoordinates()))
                return p;
        }
        for (Piece p : pieces.get(BLACK)) {
            if (pos.equals(p.getCoordinates()))
                return p;
        }

        return null;
    }

    /**
```

```
     * Returns whether there is an ongoing check or not
     *
     * @return whether a check is on going or not
     */
    public boolean isChecked() {
        return check;
    }

    /**
     * Verifies if the king is in check.
     *
     * @param opponentColor the color of the opponent pieces
     * @param position      the coordinates of the king
     * @return true if the king is in check and false otherwise
     */
    private boolean verifyCheck(PlayerColor opponentColor, Coordinates position) {
        for (Piece oppenentPiece : pieces.get(opponentColor.ordinal())) {
            boolean isOnPath = oppenentPiece.canCaptureAt(position);
            boolean isReachable = oppenentPiece instanceof Knight ||
!isPathObstructed(oppenentPiece.getCoordinates(), position);

            if (isOnPath && isReachable) return true;
        }

        return false;
    }

    /**
     * Checks if a position is within the boundaries of the board.
     *
     * @param position the coordinates to check
     * @return true if the position is within boundaries and false otherwise
     */
    private boolean isInBoundaries(Coordinates position) {
        return position.x() >= 0 && position.x() < width && position.y() >= 0 &&
position.y() < height;
    }

    /**
     * Validates if a movement is allowed for a piece.
     *
     * @param p            the piece to be moved
     * @param target       the target piece at the destination, if there's one
     * @param from         the starting coordinates
     * @param to           the destination coordinates
     * @param colorPlaying the color of the player making the move
     * @return true if the movement is valid and false otherwise
     */
    private boolean isMovementValid(Piece p, Piece target, Coordinates from,
Coordinates to, PlayerColor colorPlaying) {
        // General invalid movement cases
        if (p == null || !p.getColor().equals(colorPlaying) || !(p instanceof
Knight) && isPathObstructed(from, to)) {
            return false;
```

```
        }
        // Invalid movement cases depending on the destination
        if (target == null) {
            if (p instanceof Pawn && isEnPassantCapture(from, to)) {
                // Tries to enpassant
                return true;
            }
            return p.canMoveTo(to);
        } else {
            if (target.getColor() == p.getColor()) return false;
            else return p.canCaptureAt(target.getCoordinates());
        }
    }

    /**
     * Adds a piece to an array of pieces.
     *
     * @param array the array to add the piece to
     * @param p     the piece to be added
     */
    private void add(Piece[] array, Piece p) {
        int i = 0;
        while (i < array.length && array[i] != null) ++i;

        array[i] = p;
    }

    /**
     * Handles the castling move for a king and a rook.
     *
     * @param king the king involved in castling
     * @param rook the rook involved in castling
     * @return true if the castling move was successful and false otherwise
     */
    private boolean castle(King king, Rook rook) {
        if (king.hasMoved() || rook.hasMoved()) return false;

        int d = rook.getCoordinates().x() < king.getCoordinates().x() ? -1 : 1;
        var to = king.getCoordinates().move(d * CASTLE_DIST, 0);
        // The rook is not going to `to`, but the path has to be clear anyway
        if (isPathObstructed(rook.getCoordinates(), king.getCoordinates())) return
false;

        // Check if an opponent can reach one of the squares on the path

        // TODO ou ajouter "|| check" en début de fonction
        if (verifyCheck(king.getColor().toggle(), king.getCoordinates().move(0,
0))) return false;
        for (var it = king.getCoordinates().move(d, 0); !it.equals(to); it =
it.move(d, 0)) {
            if (verifyCheck(king.getColor().toggle(), it)) return false;
        }

        rook.moveTo(king.getCoordinates().move(d, 0));
```

```
            king.moveTo(king.getCoordinates().move(d * CASTLE_DIST, 0));

            return true;
        }

        /**
         * Reset the en passant flags for a given color
         *
         * @param playerColor the color to reset the en passant flags
         */
        private void resetEnPassantFlags(int playerColor) {
            for (Piece piece : pieces.get(playerColor)) {
                if (piece instanceof Pawn) {
                    ((Pawn) piece).setCapturableByEnpassant(false);
                }
            }
        }

        /**
         * Verifies is a capture is an en passant capture
         *
         * @param from start coordinates
         * @param to   destination coordinates
         * @return boolean value indicating the result
         */
        private boolean isEnPassantCapture(Coordinates from, Coordinates to) {
            Piece p = getPieceAt(from);
            if (!(p instanceof Pawn)) return false;

            if (from.x() != to.x() && getPieceAt(to) == null) {
                Coordinates enPassantCapturePos = new Coordinates(to.x(), from.y());
                Piece enPassantTarget = getPieceAt(enPassantCapturePos);
                return enPassantTarget instanceof Pawn && ((Pawn)
enPassantTarget).isCapturableByEnpassant();
            }
            return false;
        }
}
```

Coordinates.java

```
package engine;

/**
 * Represents a specific location on the chessboard.
 * Helps calculate movement and shows the position as text.
 */
public record Coordinates(int x, int y) {

    /**
```

```
     * Moves the position by adding the given values to x and y.
     *
     * @param dx the change in the x
     * @param dy the change in the y
     * @return a new Coordinates with the updated position
     */
    public Coordinates move(int dx, int dy) {
        int rx = this.x() + dx;
        int ry = this.y() + dy;

        return new Coordinates(rx, ry);
    }

    @Override
    public String toString() {
        return String.format("(%s, %s)", x(), y());
    }
}
```

Piece.java

```
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
import engine.Coordinates;
import engine.movements.Movement;

/**
 * Represents a chess piece. All pieces inherit from this abstract class.
 */
public abstract class Piece {
    private final PlayerColor color;
    private Coordinates coordinates;
    private final Movement[] pieceMovements;

    private final Movement[] pieceMovementRestrictions;

    /**
     * Constructor for the Piece class
     *
     * @param color                     color of the piece
     * @param coordinates               initial coordinates of the piece
     * @param pieceMovements            allowed kind of movements for the piece
     * @param pieceMovementRestrictions movement restrictions for the piece
     */
    public Piece(PlayerColor color, Coordinates coordinates, Movement[]
pieceMovements, Movement[] pieceMovementRestrictions) {
        this.color = color;
        this.coordinates = coordinates;
```

```
        this.pieceMovements = pieceMovements;
        this.pieceMovementRestrictions = pieceMovementRestrictions;
    }

    /**
     * Gets the visual type of the piece.
     *
     * @return the type of the piece
     */
    public abstract PieceType getGraphicalType();

    /**
     * Checks if the piece can move to the target position.
     *
     * @param destination the target position
     * @return true if the move is valid and false otherwise
     */
    public boolean canMoveTo(Coordinates destination) {
        if (isExceptionalMoveAllowed(destination)) return true;
        for (Movement restriction : pieceMovementRestrictions)
            if (!restriction.canMove(this.coordinates, destination)) {
                return false;
            }
        for (Movement movement : pieceMovements)
            if (movement.canMove(this.coordinates, destination)) {
                return true;
            }

        return false;
    }

    /**
     * This method is used to implement special moves that don't fall in the
piece's default moveset.
     *
     * @param dest the target position
     * @return true if the move is allowed
     */
    public boolean isExceptionalMoveAllowed(Coordinates dest) {
        return false;
    }

    /**
     * Check if the piece could capture at the destination REGARDLESS OF THE FACT
THAT THE PATH
     * IS CLEAR OR NOT
     *
     * @param destination the target of the capture
     * @return boolean that represents if the piece can capture at dest
     */
    public boolean canCaptureAt(Coordinates destination) {
        return canMoveTo(destination);
    }
```

```
    /**
     * Definitely moves the piece to a destination
     *
     * @param destination the destination
     */
    public void moveTo(Coordinates destination) {
        this.coordinates = destination;
    }

    /**
     * Returns the color of the piece
     *
     * @return the color of the piece as a PlayerColor value
     */
    public PlayerColor getColor() {
        return color;
    }

    /**
     * Returns the piece's coordinates.
     *
     * @return the piece's coordinates
     */
    public Coordinates getCoordinates() {
        return coordinates;
    }

    /**
     * toString value for the class. Abstract so that child class is forced to
specify the text value.
     *
     * @return the text value for the class
     */
    public abstract String toString();
}
```

FirstMovePiece.java

```
package engine.piece;

import chess.PlayerColor;
import engine.Coordinates;
import engine.movements.Movement;

/**
 * This abstract class represents a piece that tracks if it has already moved
since its creation.
 */
public abstract class FirstMovePiece extends Piece {
```

```
    private boolean hasMoved = false;

    /**
     * Constructor for the FirstMovePiece class
     *
     * @param color                     color of the piece
     * @param coordinates               initial coordinates of the piece
     * @param pieceMovements            allowed kind of movements for the piece
     * @param pieceMovementRestrictions movement restrictions for the piece
     */
    protected FirstMovePiece(PlayerColor color, Coordinates coordinates,
Movement[] pieceMovements,
                             Movement[] pieceMovementRestrictions) {
        super(color, coordinates, pieceMovements, pieceMovementRestrictions);
    }

    /**
     * Definitely moves the piece to a destination
     *
     * @param destination the destination
     */
    @Override
    public void moveTo(Coordinates destination) {
        super.moveTo(destination);
        if (this instanceof Pawn && !hasMoved) {
            ((Pawn) this).setCapturableByEnpassant(true);
        }
        hasMoved = true;
    }

    /**
     * Checks if the piece has moved before.
     *
     * @return true if the piece has already moved, false otherwise
     */
    public boolean hasMoved() {
        return hasMoved;
    }
}
```

Bishop.java

```
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
import engine.Coordinates;
import engine.movements.DiagonalMovement;
import engine.movements.Movement;
```

```java
/**
 * Represents the Bishop piece in chess.
 * Bishops can move diagonally any number of squares.
 */
public class Bishop extends Piece {
    /**
     * Constructor for the Bishop class
     * @param color color of the Bishop
     * @param coordinates initial coordinate of the Bishop
     */
    public Bishop(PlayerColor color, Coordinates coordinates) {
        super(color, coordinates,
            new Movement[]{new DiagonalMovement()},
            new Movement[]{}
        );
    }

    /**
     * Gets the visual type of the piece.
     *
     * @return the type of the piece
     */
    @Override
    public PieceType getGraphicalType() {
        return PieceType.BISHOP;
    }

    /**
     * toString value for the class
     * @return the text value for the class ("Bishop" here)
     */
    public String toString(){
        return "Bishop";
    }
}
```

King.java

```java
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
import engine.Coordinates;
import engine.movements.*;

/**
 * Represents the king piece in chess.
 * The King can move one square in any direction.
 */
```

```
public class King extends FirstMovePiece {
    private static final int CASTLE_DIST = 2;

    /**
     * Constructor for the King Class
     * @param color color of the King
     * @param coordinates initial coordinate of the King
     */
    public King(PlayerColor color, Coordinates coordinates) {
        super(color, coordinates,
            new Movement[]{
                new AxialMovement(),
                new DiagonalMovement()
            },
            new Movement[]{
                new RadiusMovementRestriction(1)
            }
        );
    }

    /**
     * Gets the visual type of the piece.
     *
     * @return the type of the piece
     */
    @Override
    public PieceType getGraphicalType() {
        return PieceType.KING;
    }

    /**
     * This method is used to implement special moves that don't fall in the
piece's default moveset. In the king's
     * case, it will verify if a castle is tried.
     *
     * @param dest the target position
     * @return true if the move is allowed
     */
    @Override
    public boolean isExceptionalMoveAllowed(Coordinates dest) {
        if (hasMoved()) return false;

        boolean isLeftRook = dest.equals(getCoordinates().move(-CASTLE_DIST, 0));
        boolean isRightRook = dest.equals(getCoordinates().move(CASTLE_DIST, 0));

        return isLeftRook || isRightRook;
    }

    /**
     * toString value for the class
     * @return the text value for the class ("King" here)
     */
    public String toString(){
        return "King";
```

```
    }
}
```

Knight.java

```
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
import engine.Coordinates;
import engine.movements.KnightMovement;
import engine.movements.Movement;

/**
 * Represents the Knight piece in chess.
 * Knights move in L shape and can jump over other pieces.
 */
public class Knight extends Piece {
    /**
     * Constructor for the Knight class
     *
     * @param color       color of the Knight
     * @param coordinates initial coordinate of the Knight
     */
    public Knight(PlayerColor color, Coordinates coordinates) {
        super(color, coordinates,
            new Movement[]{
                new KnightMovement()
            },
            new Movement[]{}
        );
    }

    /**
     * Gets the visual type of the piece.
     *
     * @return the type of the piece
     */
    @Override
    public PieceType getGraphicalType() {
        return PieceType.KNIGHT;
    }

    /**
     * toString value for the class
     * @return the text value for the class ("Knight" here)
     */
    public String toString(){
        return "Knight";
    }
```

```
}
```

Pawn.java

```
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
import engine.Coordinates;
import engine.movements.*;

/**
 * Represents the Pawn piece in chess.
 * Pawns move forward one square, and two square if it's their first move.
 * They can capture diagonally, and en passant.
 */
public class Pawn extends FirstMovePiece {

    private static final int LONG_JUMP_DIST = 2;
    private final Movement[] captureRestrictions;
    private boolean capturableByEnpassant = false;

    /**
     * Constructor for the Pawn class
     *
     * @param color       color of the Pawn
     * @param coordinates initial coordinate of the Pawn
     */
    public Pawn(PlayerColor color, Coordinates coordinates) {
        super(color, coordinates,
                new Movement[]{
                        new AxialMovement()
                },
                new Movement[]{
                        new DirectionMovementRestriction(color),
                        new RadiusMovementRestriction(1)  // Radius changed to 1
after first move
                }
        );
        captureRestrictions = new Movement[]{
                new DirectionMovementRestriction(color),
                new RadiusMovementRestriction(1),
                new DiagonalMovement()
        };
    }

    /**
     * Gets the capturableByEnpassant attribute of the pawn.
     *
     * @return the value
```

```
     */
    public boolean isCapturableByEnpassant() {
        return this.capturableByEnpassant;
    }

    /**
     * Sets the capturableByEnpassant attribute of the pawn.
     *
     * @param capturableByEnpassant the value to set
     */
    public void setCapturableByEnpassant(boolean capturableByEnpassant) {
        this.capturableByEnpassant = capturableByEnpassant;
    }

    /**
     * This method is used to implement special moves that don't fall in the
piece's default moveset. In the pawn's case,
     * it will try to move to squares forward.
     *
     * @param dest the target position
     * @return true if the move is allowed
     */
    @Override
    public boolean isExceptionalMoveAllowed(Coordinates dest) {
        int jumpDistance = LONG_JUMP_DIST;
        if (getColor() == PlayerColor.BLACK) jumpDistance *= -1;

        return !hasMoved() && dest.equals(getCoordinates().move(0, jumpDistance));
    }

    /**
     * Check if the pawn could capture at the destination. Does not cover en
passant.
     *
     * @param destination the target of the capture
     * @return boolean that represents if the piece can capture at dest
     */
    @Override
    public boolean canCaptureAt(Coordinates destination) {
        for (var movement : captureRestrictions) {
            if (!movement.canMove(getCoordinates(), destination)) return false;
        }
        return true;
    }

    /**
     * Gets the visual type of the piece.
     *
     * @return the type of the piece
     */
    @Override
    public PieceType getGraphicalType() {
        return PieceType.PAWN;
    }
```

```
    /**
     * toString value for the class
     *
     * @return the text value for the class ("Pawn" here)
     */
    public String toString() {
        return "Pawn";
    }
}
```

Queen.java

```
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
import engine.Coordinates;
import engine.movements.AxialMovement;
import engine.movements.DiagonalMovement;
import engine.movements.Movement;

/**
 * Represents the Queen piece in chess.
 * Queens can move any number of squares in a line or diagonally.
 */
public class Queen extends Piece {
    /**
     * Constructor for the Queen class
     *
     * @param color       color of the Queen
     * @param coordinates initial coordinate of the Queen
     */
    public Queen(PlayerColor color, Coordinates coordinates) {
        super(color, coordinates,
            new Movement[]{
                new AxialMovement(),
                new DiagonalMovement()
            },
            new Movement[]{}
        );
    }

    /**
     * Gets the visual type of the piece.
     *
     * @return the type of the piece
     */
    @Override
    public PieceType getGraphicalType() {
```

```
        return PieceType.QUEEN;
    }

    /**
     * toString value for the class
     * @return the text value for the class ("Queen" here)
     */
    public String toString(){
        return "Queen";
    }
}
```

Rook.java

```
package engine.piece;

import chess.PieceType;
import chess.PlayerColor;
import engine.Coordinates;
import engine.movements.AxialMovement;
import engine.movements.Movement;

/**
 * Represents the Rook piece in chess.
 * Rooks move any number of squares in a line.
 */
public class Rook extends FirstMovePiece {
    /**
     * Constructor for the Rook class
     *
     * @param color       color of the Rook
     * @param coordinates initial coordinate of the Queen
     */
    public Rook(PlayerColor color, Coordinates coordinates) {
        super(color, coordinates,
                new Movement[]{
                        new AxialMovement()
                },
                new Movement[]{}
        );
    }

    /**
     * Gets the visual type of the piece.
     *
     * @return the type of the piece
     */
    @Override
    public PieceType getGraphicalType() {
        return PieceType.ROOK;
    }
```

```
    /**
     * toString value for the class
     *
     * @return the text value for the class ("Rook" here)
     */
    public String toString() {
        return "Rook";
    }
}
```

Movement.java

```
package engine.movements;

import engine.Coordinates;

/**
 * Defines how a chess piece can move on the board.
 */
public interface Movement {
    /**
     * Checks if a piece can move from one position to another.
     *
     * @param from the starting position
     * @param to   the target position
     * @return true if the move is allowed and false otherwise
     */
    boolean canMove(Coordinates from, Coordinates to);
}
```

AxialMovement.java

```
package engine.movements;

import engine.Coordinates;

/**
 * Allows movement in straight lines.
 */
public class AxialMovement implements Movement {
    /**
     * Checks that a piece's movement is strictly axial.
     *
     * @param from the starting position
     * @param to   the target position
```

```
     * @return true if the move is allowed and false otherwise
     */
    @Override
    public boolean canMove(Coordinates from, Coordinates to) {
        return from.x() == to.x() ^ from.y() == to.y();
    }
}
```

DiagonalMovement.java

```
package engine.movements;

import engine.Coordinates;

/**
 * Allows movement diagonally on the board.
 */
public class DiagonalMovement implements Movement {
    /**
     * Checks that a piece's movement is strictly diagonal.
     *
     * @param from the starting position
     * @param to   the target position
     * @return true if the move is allowed and false otherwise
     */
    @Override
    public boolean canMove(Coordinates from, Coordinates to) {
        return Math.abs(from.x() - to.x()) == Math.abs(from.y() - to.y());
    }
}
```

DirectionMovementRestriction.java

```
package engine.movements;

import chess.PlayerColor;
import engine.Coordinates;

/**
 * Restricts movement to a specific direction, useful for pawns.
 */
public class DirectionMovementRestriction implements Movement {

    private final PlayerColor color;

    /**
```

```
     * Sets the movement direction based on the player  color.
     *
     * @param color the player color
     */
    public DirectionMovementRestriction(PlayerColor color) {
        this.color = color;
    }

    /**
     * Checks that a piece moves in it's allowed direction.
     *
     * @param from the starting position
     * @param to   the target position
     * @return true if the move is allowed and false otherwise
     */
    @Override
    public boolean canMove(Coordinates from, Coordinates to) {
        return switch (color) {
            case WHITE -> from.y() < to.y();
            case BLACK -> from.y() > to.y();
        };
    }
}
```

KnightMovement.java

```
package engine.movements;

import engine.Coordinates;

/**
 * Allows movement knight movements, which is complex in shape.
 */
public class KnightMovement implements Movement {
    /**
     * Checks that a piece's is a knight move.
     *
     * @param from the starting position
     * @param to   the target position
     * @return true if the move is allowed and false otherwise
     */
    @Override
    public boolean canMove(Coordinates from, Coordinates to) {
        int xDiff = Math.abs(from.x() - to.x());
        int yDiff = Math.abs(from.y() - to.y());

        return xDiff == 2 && yDiff == 1 || xDiff == 1 && yDiff == 2;
    }
}
```

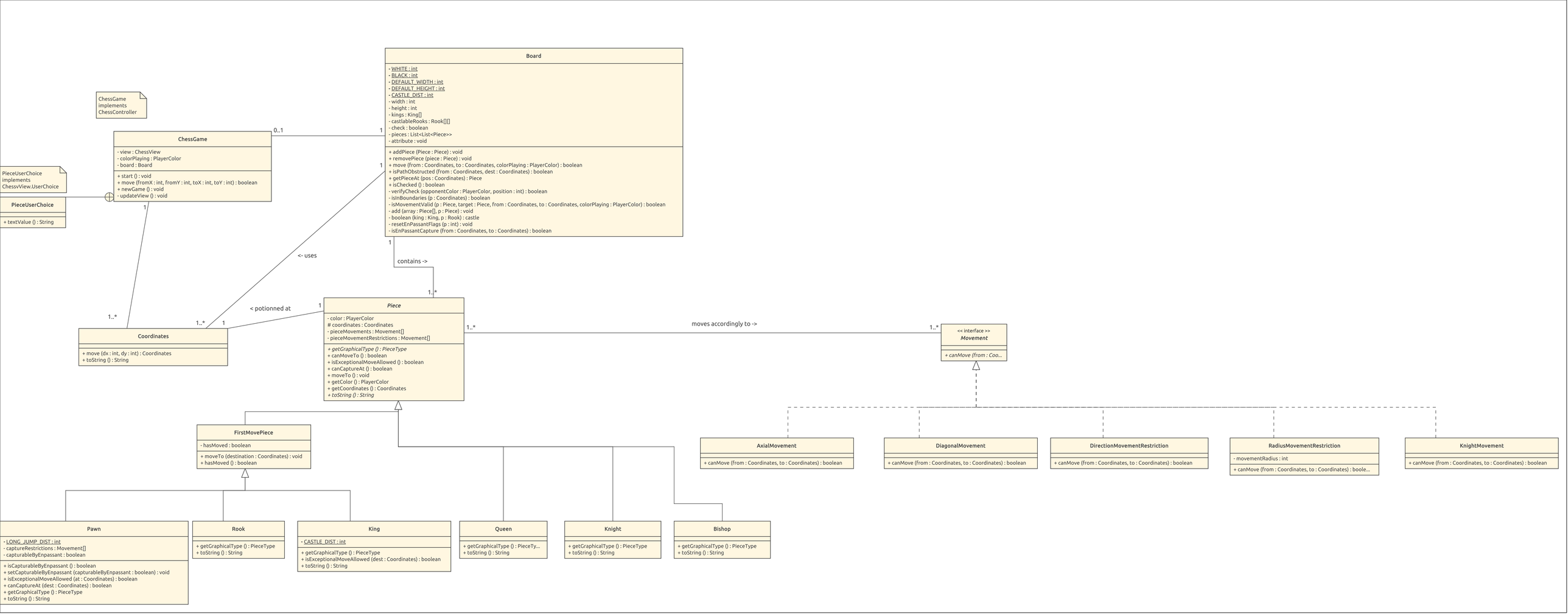
RadiusMovementRestriction.java

```java
package engine.movements;

import engine.Coordinates;

/**
 * Limits movement to a certain radius from the starting position.
 */
public class RadiusMovementRestriction implements Movement {

    private final int movementRadius;

    /**
     * Sets the maximum distance the piece can move.
     *
     * @param movementRadius the maximum number of squares
     */
    public RadiusMovementRestriction(int movementRadius) {
        this.movementRadius = movementRadius;
    }

    /**
     * Checks that a piece does not move more that the allowed radius.
     *
     * @param from the starting position
     * @param to   the target position
     * @return true if the move is allowed and false otherwise
     */
    @Override
    public boolean canMove(Coordinates from, Coordinates to) {
        int xDiff = Math.abs(from.x() - to.x());
        int yDiff = Math.abs(from.y() - to.y());

        return xDiff <= movementRadius && yDiff <= movementRadius;
    }
}
```

---

9. Annexes

- **Code source**
- **Diagramme UML**


Board
- WHITE : int
- BLACK : int
- DEFAULT_WIDTH : int
- DEFAULT_HEIGHT : int
- CASTLE_DIST : int
- width : int
- height : int
- kings : King[]
- castlableRooks : Rook[][]
- check : boolean
- pieces : List<List<Piece>>
- attribute : void
+ addPiece (Piece : Piece) : void
+ removePiece (piece : Piece) : void
+ move (from : Coordinates, to : Coordinates, colorPlaying : PlayerColor) : boolean
+ isPathObstructed (from : Coordinates, dest : Coordinates) : boolean
+ getPieceAt (pos : Coordinates) : Piece
+ isChecked () : boolean
- verifyCheck (opponentColor : PlayerColor, position : int) : boolean
- isInBoundaries (p : Coordinates) : boolean
- isMovementValid (p : Piece, target : Piece, from : Coordinates, to : Coordinates, colorPlaying : PlayerColor) : boolean
- add (array : Piece[], p : Piece) : void
- boolean (king : King, p : Rook) : castle
- resetEnPassantFlags (p : int) : void
- isEnPassantCapture (from : Coordinates, to : Coordinates) : boolean
ChessGame
implements
ChessController
ChessGame
- view : ChessView
- colorPlaying : PlayerColor
- board : Board
+ start () : void
+ move (fromX : int, fromY : int, toX : int, toY : int) : boolean
+ newGame () : void
- updateView () : void
0..1
1
PieceUserChoice
implements
ChessvView.UserChoice
PieceUserChoice
+ textValue () : String
1
<- uses
contains ->
1..*
1..*
1
1..*
< potionned at
Coordinates
+ move (dx : int, dy : int) : Coordinates
+ toString () : String
Piece
- color : PlayerColor
coordinates : Coordinates
- pieceMovements : Movement[]
- pieceMovementRestrictions : Movement[]
+ getGraphicalType () : PieceType
+ canMoveTo () : boolean
+ isExceptionalMoveAllowed () : boolean
+ canCaptureAt () : boolean
+ moveTo () : void
+ getColor () : PlayerColor
+ getCoordinates () : Coordinates
+ toString () : String
1..*
moves accordingly to ->
1..*
<< interface >>
Movement
+ canMove (from : Coo...
FirstMovePiece
- hasMoved : boolean
+ moveTo (destination : Coordinates) : void
+ hasMoved () : boolean
AxialMovement
+ canMove (from : Coordinates, to : Coordinates) : boolean
DiagonalMovement
+ canMove (from : Coordinates, to : Coordinates) : boolean
DirectionMovementRestriction
+ canMove (from : Coordinates, to : Coordinates) : boolean
RadiusMovementRestriction
- movementRadius : int
+ canMove (from : Coordinates, to : Coordinates) : boole...
KnightMovement
+ canMove (from : Coordinates, to : Coordinates) : boolean
Pawn
- LONG_JUMP_DIST : int
- captureRestrictions : Movement[]
- capturableByEnpassant : boolean
+ isCapturableByEnpassant () : boolean
+ setCapturableByEnpassant (capturableByEnpassant : boolean) : void
+ isExceptionalMoveAllowed (at : Coordinates) : boolean
+ canCaptureAt (dest : Coordinates) : boolean
+ getGraphicalType () : PieceType
+ toString () : String
Rook
+ getGraphicalType () : PieceType
+ toString () : String
King
- CASTLE_DIST : int
+ getGraphicalType () : PieceType
+ isExceptionalMoveAllowed (dest : Coordinates) : boolean
+ toString () : String
Queen
+ getGraphicalType () : PieceTy...
+ toString () : String
Knight
+ getGraphicalType () : PieceType
+ toString () : String
Bishop
+ getGraphicalType () : PieceType
+ toString () : String