

~/Documents/HEIG/24-25/P00/Lab08/moves.java

```
package chess.moves;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

import chess.Board;
import chess.Square;

/**
 * Interface pour valider si un chemin entre deux cases sur un échiquier est dégagé.
 */
public interface PathValidator {
    /**
     * Vérifie si le chemin entre deux cases sur l'échiquier est libre de toute obstruction.
     *
     * @param board l'échiquier représentant l'état actuel du jeu.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @return {@code true} si le chemin est dégagé, {@code false} sinon.
     */
    boolean isPathClear(Board board, Square start, Square end);
}

package chess.moves;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

import chess.Board;
import chess.Square;

/**
 * Vérifie si le chemin entre deux cases sur l'échiquier est dégagé.
 * Implémente l'interface {@link PathValidator}.
 *
 * Lève une exception {@link IllegalArgumentException} si la case de départ
 * est identique à la case d'arrivée.</p>
 */
public class DefaultPathValidator implements PathValidator {
    /**
     * Vérifie si le chemin entre deux cases est dégagé.
     *
     * @param board l'état actuel de l'échiquier.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @return {@code true} si le chemin est dégagé, {@code false} sinon.
     * @throws IllegalArgumentException si la case de départ est identique à la case d'arrivée.
     */
    @Override
    public boolean isPathClear(Board board, Square start, Square end) {
        int stepX = end.getX() - start.getX();
        int stepY = end.getY() - start.getY();

        if(stepX == 0 && stepY == 0) {
            throw new IllegalArgumentException("Aucun déplacement");
        }

        int x = (stepX > 0) ? start.getX() + 1 : (stepX == 0) ? end.getX() : start.getX() - 1;
```

```

        int y = (stepY > 0) ? start.getY() + 1 : (stepY == 0) ? end.getY() : start.getY() - 1;

        while (x != end.getX() || y != end.getY()) {
            if (board.getSquare(x, y).isOccupied()) {
                return false; // Une pièce bloque le chemin
            }

            x = (stepX > 0) ? x + 1 : (stepX == 0) ? end.getX() : x - 1;
            y = (stepY > 0) ? y + 1 : (stepY == 0) ? end.getY() : y - 1;
        }

        return true;
    }
}

package chess.moves;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

/**
 * Représente les différents types de déplacements possibles pour les pièces d'échecs.
 * Chaque type de déplacement vérifie si les coordonnées fournies respectent ses règles spécifiques.
 */
public enum MoveType {
    /**
     * Déplacement en diagonale.
     * Valide si les deux décalages sont égaux et strictement positifs.
     */
    DIAGONAL {
        @Override
        public boolean isValid(int deltaX, int deltaY) {
            return deltaX == deltaY && deltaX > 0;
        }
    },
    /**
     * Déplacement horizontal.
     * Valide si le décalage vertical est nul et le décalage horizontal est strictement positif.
     */
    HORIZONTAL {
        @Override
        public boolean isValid(int deltaX, int deltaY) {
            return deltaY == 0 && deltaX > 0;
        }
    },
    /**
     * Déplacement vertical.
     * Valide si le décalage horizontal est nul et le décalage vertical est strictement positif.
     */
    VERTICAL {
        @Override
        public boolean isValid(int deltaX, int deltaY) {
            return deltaX == 0 && deltaY > 0;
        }
    },
    /**
     * Déplacement en forme de "L".
     * Valide si les décalages correspondent aux mouvements possibles d'un cavalier (2x1 ou 1x2).
     */
    L_SHAPE {
        @Override

```

```

        public boolean isValid(int deltaX, int deltaY) {
            return (deltaX == 2 && deltaY == 1) || (deltaX == 1 && deltaY == 2);
        }
    };
    /**
     * Vérifie si un déplacement est valide pour ce type de mouvement.
     *
     * @param deltaX le décalage horizontal entre la case de départ et la case d'arrivée.
     * @param deltaY le décalage vertical entre la case de départ et la case d'arrivée.
     * @return {@code true} si le déplacement respecte les règles du type de mouvement, {@code false} sinon.
     */
    public abstract boolean isValid(int deltaX, int deltaY);
}

```

```

package chess.moves;

```

```

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

```

```

import chess.Board;
import chess.Square;

```

```

/**
 * Interface représentant un coup spécial dans un jeu d'échecs
 * (par exemple, roque ou prise en passant).
 */

```

```

public interface SpecialMove {
    /**
     * Vérifie si le coup spécial est valide pour l'état actuel de l'échiquier.
     *
     * @param board l'échiquier représentant l'état actuel du jeu.
     * @param start la case de départ du mouvement.
     * @param end la case d'arrivée du mouvement.
     * @return {@code true} si le coup spécial est valide, {@code false} sinon.
     */
    boolean isValid(Board board, Square start, Square end);

    /**
     * Exécute le coup spécial sur l'échiquier.
     *
     * @param board l'échiquier représentant l'état actuel du jeu.
     * @param start la case de départ du mouvement.
     * @param end la case d'arrivée du mouvement.
     */
    void execute(Board board, Square start, Square end);
}

```

~/Documents/HEIG/24-25/P00/Lab08/pieces.java

```
package chess.pieces;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

import chess.PieceType;
import chess.PlayerColor;

/**
 * Classe abstraite représentant une pièce qui a un mouvement spécial lors de son premier coup.
 * Les pièces dérivées de cette classe (comme le roi et la tour) peuvent avoir des mouvements spéciaux
 * qui ne peuvent être effectués que si la pièce n'a pas encore bougé, comme le roque.
 */
abstract class SpecialFirstMovePiece extends Piece {
    private boolean hasMoved = false;

    /**
     * Constructeur pour initialiser une pièce avec un type et une couleur.
     *
     * @param color la couleur de la pièce (blanc ou noir).
     * @param type le type de la pièce (par exemple, roi, tour).
     */
    protected SpecialFirstMovePiece(PlayerColor color, PieceType type) {
        super(color, type);
    }

    /**
     * Vérifie si la pièce a déjà été déplacée.
     *
     * @return true si la pièce a été déplacée, false sinon.
     */
    public boolean hasMoved() {
        return hasMoved;
    }

    /**
     * Marque la pièce comme ayant été déplacée.
     * Cette méthode est utilisée après qu'un mouvement ait été effectué sur la pièce,
     * par exemple, après un roque ou tout autre mouvement qui implique cette pièce.
     */
    public void markAsMoved() {
        this.hasMoved = true;
    }

    /**
     * Réinitialise l'état de la pièce pour indiquer qu'elle n'a pas été déplacée.
     * Cette méthode peut être utilisée pour réinitialiser la pièce dans certaines situations
     * (par exemple, lors du retour d'un état du jeu ou d'une autre logique spécifique).
     */
    public void resetMove() {
        this.hasMoved = false;
    }
}

package chess.pieces;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */
```

```

*/

import chess.*;
import chess.moves.MoveType;
import chess.moves.PathValidator;
import chess.moves.DefaultPathValidator;

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

/**
 * Représente la pièce Tour dans le jeu d'échecs.
 * La Tour se déplace horizontalement ou verticalement sur n'importe quelle distance,
 * tant que son chemin est dégagé. La Tour est également impliquée dans le mouvement spécial du roque.
 */
public class Rook extends SpecialFirstMovePiece {

    private final List<MoveStrategy> moveStrategies = new ArrayList<>();

    /**
     * Constructeur pour initialiser la Tour avec sa couleur.
     *
     * @param color la couleur de la Tour (blanc ou noir).
     */
    public Rook(PlayerColor color) {
        super(color, PieceType.ROOK);
        // Ajouter les stratégies de mouvement
        moveStrategies.add(new StandardRookMove());
    }

    /**
     * Vérifie si la Tour peut se déplacer de la case de départ à la case d'arrivée.
     * La Tour peut se déplacer horizontalement ou verticalement,
     * tant que le chemin est dégagé.
     *
     * @param board le plateau de jeu sur lequel le mouvement est effectué.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @return true si le mouvement est valide, false sinon.
     */
    @Override
    public boolean canMove(Board board, Square start, Square end) {
        for (MoveStrategy strategy : moveStrategies) {
            if (strategy.isValid(board, start, end)) {
                return true;
            }
        }
        return false;
    }

    /**
     * Exécute le mouvement de la Tour de la case de départ à la case d'arrivée.
     *
     * @param board le plateau de jeu sur lequel le mouvement est effectué.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @throws IllegalArgumentException si le mouvement est invalide.
     */
    @Override
    public void executeMove(Board board, Square start, Square end) {
        for (MoveStrategy strategy : moveStrategies) {
            if (strategy.isValid(board, start, end)) {
                strategy.execute(board, start, end);
            }
        }
    }
}

```

```

        return;
    }
}
throw new IllegalArgumentException("Mouvement invalide pour la tour.");
}

/**
 * Vérifie si la Tour peut participer au mouvement spécial du roque.
 * La Tour peut participer au roque si elle n'a pas encore bougé.
 *
 * @return true si la Tour peut participer au roque, false sinon.
 */
public boolean canParticipateInCastling() {
    return !hasMoved();
}

/**
 * Marque la Tour comme ayant participé au roque après l'exécution du mouvement.
 */
public void participatedInCastling() {
    markAsMoved();
}

/**
 * Classe interne représentant le mouvement standard de la Tour.
 * La Tour peut se déplacer horizontalement ou verticalement.
 */
private class StandardRookMove implements MoveStrategy {

    private final PathValidator pathValidator = new DefaultPathValidator();

    /**
     * Vérifie si le mouvement de la Tour de la case de départ à la case d'arrivée est valide.
     *
     * @param board le plateau de jeu.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @return true si le mouvement est valide, false sinon.
     */
    @Override
    public boolean isValid(Board board, Square start, Square end) {
        int deltaX = Rook.this.distanceX(end);
        int deltaY = Rook.this.distanceY(end);

        // La tour peut se déplacer horizontalement ou verticalement
        if (!MoveType.HORIZONTAL.isValid(deltaX, deltaY) &&
            !MoveType.VERTICAL.isValid(deltaX, deltaY)) {
            return false;
        }

        // Vérification du chemin via le PathValidator
        return pathValidator.isPathClear(board, start, end) && Rook.super.canMove(board, start, end);
    }

    /**
     * Exécute le mouvement de la Tour de la case de départ à la case d'arrivée.
     *
     * @param board le plateau de jeu.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     */
    @Override
    public void execute(Board board, Square start, Square end) {
        Piece rook = board.getPiece(start.getX(), start.getY());
        board.movePiece(rook, end);
    }
}

```

```

        markAsMoved();
    }
}
}

```

```

package chess.pieces;

```

```

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

```

```

import chess.*;
import chess.moves.MoveType;
import chess.moves.PathValidator;
import chess.moves.DefaultPathValidator;

```

```

/**
 * Représente un fou dans un jeu d'échecs.
 * Le fou peut se déplacer uniquement en diagonale et son chemin doit être dégagé.
 */

```

```

public class Bishop extends Piece {

```

```

    /**
     * Valideur pour vérifier que le chemin entre deux cases est dégagé.
     */

```

```

    private final PathValidator pathValidator = new DefaultPathValidator();

```

```

    /**
     * Initialise un fou avec une couleur spécifiée.
     *
     * @param color la couleur du joueur à laquelle appartient le fou.
     */

```

```

    public Bishop(PlayerColor color) {
        super(color, PieceType.BISHOP);
    }

```

```

    /**
     * Vérifie si le fou peut se déplacer d'une case à une autre.
     *
     * @param board l'échiquier représentant l'état actuel du jeu.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @return {@code true} si le déplacement est valide (diagonal, chemin dégagé,
     *         et respecte les règles générales des pièces), {@code false} sinon.
     */

```

```

    @Override

```

```

    public boolean canMove(Board board, Square start, Square end) {

        // Le fou peut se déplacer uniquement en diagonale
        if (!MoveType.DIAGONAL.isValid(distanceX(end), distanceY(end))) {
            return false;
        }

        // Vérification du chemin via le PathValidator
        return pathValidator.isPathClear(board, start, end) && super.canMove(board, start, end);
    }
}

```

```

package chess.pieces;

```

```

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

```

```

import chess.*;
import chess.moves.MoveType;

/**
 * Représente un cavalier dans un jeu d'échecs.
 * Le cavalier se déplace en forme de "L" (2 cases dans une direction et 1 case perpendiculaire, ou
 * inversement).
 */
public class Knight extends Piece {

    /**
     * Initialise un cavalier avec une couleur spécifiée.
     *
     * @param color la couleur du joueur à laquelle appartient le cavalier.
     */
    public Knight(PlayerColor color) {
        super(color, PieceType.KNIGHT);
    }

    /**
     * Vérifie si le cavalier peut se déplacer d'une case à une autre.
     * Le déplacement doit respecter la forme en "L" (2x1 ou 1x2) et les règles générales de mouvement.
     *
     * @param board l'échiquier représentant l'état actuel du jeu.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @return {@code true} si le déplacement est valide, {@code false} sinon.
     */
    @Override
    public boolean canMove(Board board, Square start, Square end) {
        return (MoveType.L_SHAPE.isValid(distanceX(end), distanceY(end)) && super.canMove(board, start, end) );
    }
}

package chess.pieces;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

import chess.*;

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

/**
 * Représente un roi dans un jeu d'échecs.
 * Le roi peut effectuer des mouvements standards d'une case dans toutes les directions
 * ou des roques dans certaines conditions.
 */
public class King extends SpecialFirstMovePiece {
    /**
     * Liste des stratégies de mouvement du roi, incluant les mouvements standards et le roque.
     */
    private final List<MoveStrategy> moveStrategies = new ArrayList<>();

    /**
     * Initialise un roi avec une couleur spécifiée.
     *
     * @param color la couleur du joueur à laquelle appartient le roi.
     */
    public King(PlayerColor color) {

```



```

        super(color, PieceType.KING);
        // Ajouter les stratégies de mouvement
        moveStrategies.add(new StandardKingMove());
        moveStrategies.add(new CastlingMove());
    }

    /**
     * Vérifie si le roi peut se déplacer d'une case à une autre selon ses règles de mouvement.
     *
     * @param board l'échiquier représentant l'état actuel du jeu.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @return {@code true} si le déplacement est valide, {@code false} sinon.
     */
    @Override
    public boolean canMove(Board board, Square start, Square end) {
        for (MoveStrategy strategy : moveStrategies) {
            if (strategy.isValid(board, start, end)) {
                return strategy instanceof CastlingMove || super.canMove(board, start, end);
            }
        }
        return false;
    }

    /**
     * Exécute un mouvement pour le roi sur l'échiquier.
     *
     * @param board l'échiquier représentant l'état actuel du jeu.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @throws IllegalArgumentException si le mouvement est invalide.
     */
    @Override
    public void executeMove(Board board, Square start, Square end) {
        for (MoveStrategy strategy : moveStrategies) {
            if (strategy.isValid(board, start, end)) {
                strategy.execute(board, start, end);
                if (!hasMoved()) {
                    markAsMoved();
                }
                return;
            }
        }
        throw new IllegalArgumentException("Mouvement invalide pour le roi.");
    }

    /**
     * Stratégie interne pour gérer le roque.
     */
    private class CastlingMove implements MoveStrategy {
        /**
         * Vérifie si un roque est valide dans l'état actuel du jeu.
         *
         * @param board l'échiquier.
         * @param start la case de départ du roi.
         * @param end la case d'arrivée du roi.
         * @return {@code true} si le roque est valide, {@code false} sinon.
         */
        @Override
        public boolean isValid(Board board, Square start, Square end) {
            int deltaX = end.getX() - start.getX();
            if (Math.abs(deltaX) != 2 || start.getY() != end.getY()) {
                return false;
            }
        }
    }

```

```

        int rookX = deltaX > 0 ? 7 : 0;
        Piece rook = board.getPiece(rookX, start.getY());

        if (!(rook instanceof Rook) || !((Rook) rook).canParticipateInCastling() || hasMoved()) {
            return false;
        }

        for (int x = Math.min(start.getX(), rookX) + 1; x < Math.max(start.getX(), rookX); x++) {
            if (board.getSquare(x, start.getY()).isOccupied()) {
                return false;
            }
        }

        return true;
    }

    /**
     * Exécute le roque en déplaçant le roi et la tour sur l'échiquier.
     *
     * @param board l'échiquier.
     * @param start la case de départ du roi.
     * @param end la case d'arrivée du roi.
     */
    @Override
    public void execute(Board board, Square start, Square end) {
        int deltaX = end.getX() - start.getX();
        int rookX = deltaX > 0 ? 7 : 0;
        int rookDestinationX = start.getX() + (deltaX > 0 ? 1 : -1);

        Piece king = board.getPiece(start.getX(), start.getY());
        Rook rook = (Rook) board.getPiece(rookX, start.getY());

        rook.participatedInCastling();

        board.movePiece(king, end);
        board.movePiece(rook, board.getSquare(rookDestinationX, start.getY()));
        board.getGameController().removePiece(rookX, start.getY());
        board.getGameController().setPiece(rook.getSquare().getX(), rook.getSquare().getY());
    }
}

/**
 * Stratégie interne pour gérer le mouvement standard du roi.
 */
private class StandardKingMove implements MoveStrategy {

    /**
     * Vérifie si le mouvement standard du roi est valide (une case dans toutes les directions).
     *
     * @param board l'échiquier.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @return {@code true} si le mouvement est valide, {@code false} sinon.
     */
    @Override
    public boolean isValid(Board board, Square start, Square end) {
        int deltaX = King.this.distanceX(end);
        int deltaY = King.this.distanceY(end);

        // Le roi peut se déplacer d'une case dans toutes les directions
    }
}

```

```

        return deltaX <= 1 && deltaY <= 1;
    }

    /**
     * Exécute le mouvement standard du roi.
     *
     * @param board l'échiquier.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     */
    @Override
    public void execute(Board board, Square start, Square end) {
        Piece king = board.getPiece(start.getX(), start.getY());
        board.movePiece(king, end);
    }
}

```

```
package chess.pieces;
```

```

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

```

```

import chess.*;
import chess.moves.DefaultPathValidator;
import chess.moves.PathValidator;

```

```

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

```

```

/**
 * Représente un pion dans un jeu d'échecs.
 * Le pion se déplace d'une case vers l'avant, mais peut se déplacer de deux cases lors de son premier
 mouvement.
 * Il peut capturer en diagonale, effectuer une prise en passant et être promu lorsqu'il atteint la dernière
 rangée.
 */

```

```

public class Pawn extends SpecialFirstMovePiece {

    private final int direction;
    private final List<MoveStrategy> moveStrategies = new ArrayList<>();
    private final PathValidator pathValidator = new DefaultPathValidator();

```

```

    /**
     * Initialise un pion avec une couleur spécifiée.
     *
     * @param color la couleur du joueur à laquelle appartient le pion.
     */

```

```

    public Pawn(PlayerColor color) {
        super(color, PieceType.PAWN);
        direction = color == PlayerColor.WHITE ? 1 : -1;
        // Ajouter les stratégies de mouvement
        moveStrategies.add(new StandardPawnMove());
        moveStrategies.add(new DoubleStepMove());
        moveStrategies.add(new EnPassantMove());
    }

```

```

    /**
     * Vérifie si le pion peut se déplacer d'une case à une autre selon ses règles de mouvement.
     * Le pion peut avancer d'une case, avancer de deux cases lors de son premier mouvement, ou capturer en
 diagonale.
     *
     * @param board l'échiquier représentant l'état actuel du jeu.
     */

```

```

* @param start la case de départ.
* @param end la case d'arrivée.
* @return {@code true} si le déplacement est valide, {@code false} sinon.
*/
@Override
public boolean canMove(Board board, Square start, Square end) {
    for (MoveStrategy strategy : moveStrategies) {
        if (strategy.isValid(board, start, end)) {
            return true;
        }
    }
    return false;
}
/**
 * Exécute un mouvement pour le pion sur l'échiquier, avec gestion de la promotion si nécessaire.
 *
 * @param board l'échiquier représentant l'état actuel du jeu.
 * @param start la case de départ.
 * @param end la case d'arrivée.
 * @throws IllegalArgumentException si le mouvement est invalide.
 */
@Override
public void executeMove(Board board, Square start, Square end) {
    for (MoveStrategy strategy : moveStrategies) {
        if (strategy.isValid(board, start, end)) {
            strategy.execute(board, start, end);
            markAsMoved();

            // Vérification de la promotion
            if (shouldPromote(end)) {
                promote(board, end);
            }
            return;
        }
    }
    throw new IllegalArgumentException("Mouvement invalide pour le pion.");
}

/**
 * Vérifie si le pion doit être promu (lorsqu'il atteint la dernière ligne).
 *
 * @param end la case d'arrivée.
 * @return {@code true} si le pion doit être promu, {@code false} sinon.
 */
private boolean shouldPromote(Square end) {
    return (getColor() == PlayerColor.WHITE && end.getY() == 7) || (getColor() == PlayerColor.BLACK &&
end.getY() == 0);
}

/**
 * Promeut le pion en une autre pièce (par exemple, une reine) lorsque cela est nécessaire.
 *
 * @param board l'échiquier représentant l'état actuel du jeu.
 * @param square la case où le pion doit être promu.
 */
private void promote(Board board, Square square) {
    board.getGameController().promotePawn(square);
}

// Mouvement standard d'un pion
private class StandardPawnMove implements MoveStrategy {
    /**
     * Vérifie si le mouvement standard du pion est valide (avance d'une case ou capture en diagonale).
     */

```

```

    * @param board l'échiquier.
    * @param start la case de départ.
    * @param end la case d'arrivée.
    * @return {@code true} si le mouvement est valide, {@code false} sinon.
    */
@Override
public boolean isValid(Board board, Square start, Square end) {
    int deltaX = distanceX(end);
    int deltaY = distanceY(end);

    // Vérifier que le pion avance dans la bonne direction
    if ((getColor() == PlayerColor.WHITE && end.getY() <= start.getY()) ||
        (getColor() == PlayerColor.BLACK && end.getY() >= start.getY())) {
        return false;
    }

    // Avancer d'une case
    return (deltaX == 0 && deltaY == 1 && !end.isOccupied()) ||
        (deltaX == 1 && deltaY == 1 && end.isOccupied() && !isNotSameColor(end.getPiece()));
}

/**
 * Exécute le mouvement standard du pion.
 *
 * @param board l'échiquier.
 * @param start la case de départ.
 * @param end la case d'arrivée.
 */
@Override
public void execute(Board board, Square start, Square end) {
    Piece pawn = board.getPiece(start.getX(), start.getY());
    board.movePiece(pawn, end);
    markAsMoved();
}

}

private class DoubleStepMove implements MoveStrategy {
    /**
     * Vérifie si le mouvement de deux cases est valide (le premier mouvement du pion).
     *
     * @param board l'échiquier.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @return {@code true} si le mouvement est valide, {@code false} sinon.
     */
    @Override
    public boolean isValid(Board board, Square start, Square end) {
        int deltaX = distanceX(end);
        int deltaY = distanceY(end);

        // Avancer de deux cases au premier coup
        return !Pawn.super.hasMoved() && deltaX == 0 && deltaY == 2 && !end.isOccupied()
            && Pawn.this.pathValidator.isPathClear(board, start, end);
    }

    /**
     * Exécute le mouvement de deux cases.
     *
     * @param board l'échiquier.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     */

```

```

    */
    @Override
    public void execute(Board board, Square start, Square end) {
        Piece pawn = board.getPiece(start.getX(), start.getY());
        board.movePiece(pawn, end);
        markAsMoved();
    }
}

// Prise en passant
private class EnPassantMove implements MoveStrategy {

    /**
     * Vérifie si la prise en passant est valide.
     *
     * @param board l'échiquier.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @return {@code true} si la prise en passant est valide, {@code false} sinon.
     */
    @Override
    public boolean isValid(Board board, Square start, Square end) {
        GameController.Move lastMove = ((GameController) board.getGameController()).getLastMove();
        if (lastMove == null || !(lastMove.getPiece() instanceof Pawn)) {
            return false;
        }

        Piece lastMovedPiece = lastMove.getPiece();
        Square lastFrom = lastMove.getFrom();
        Square lastTo = lastMove.getTo();

        int deltaX = distanceX(end);
        int deltaY = distanceY(end);

        // Vérification des conditions de la prise en passant
        return deltaX == 1 && deltaY == 1
            && lastMovedPiece.getColor() != Pawn.this.getColor()
            && Math.abs(lastTo.getY() - lastFrom.getY()) == 2
            && lastTo.getX() == end.getX()
            && lastTo.getY() == start.getY();
    }

    /**
     * Exécute la prise en passant.
     *
     * @param board l'échiquier.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     */
    @Override
    public void execute(Board board, Square start, Square end) {
        GameController.Move lastMove = board.getGameController().getLastMove();
        if (lastMove == null) {
            throw new IllegalStateException("Aucune prise en passant valide");
        }

        Piece pawn = board.getPiece(start.getX(), start.getY());
        board.removeCapturedPiece(board.getSquare(end.getX(), start.getY()));
        board.getGameController().removePiece(end.getX(), start.getY());
        board.movePiece(pawn, end);
    }
}

}

package chess.pieces;

```

```

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

import chess.ChessView;
import chess.PieceType;

/**
 * Représente les pièces qui peuvent être choisies par l'utilisateur lors de la promotion d'un pion.
 * Cette classe implémente l'interface {@link ChessView.UserChoice} pour fournir un choix utilisateur lors de
 * la promotion.
 */
public class PromotablePiece implements ChessView.UserChoice {

    private final PieceType pieceType;

    /**
     * Constructeur pour créer une nouvelle pièce avec un type donné.
     *
     * @param pieceType le prochain type de la pièce promue (ex. : Reine, Tour, Fou, Cavalier).
     */
    public PromotablePiece(PieceType pieceType) {
        this.pieceType = pieceType;
    }

    /**
     * Retourne la valeur textuelle du type de la pièce promue, qui est le nom de la pièce dans la classe
     * {@link PieceType}.
     *
     * @return le nom du type de la pièce promue.
     */
    @Override
    public String textValue() {
        return pieceType.name();
    }

    /**
     * Retourne le type de la pièce promue.
     *
     * @return le type de la pièce promue, qui est un membre de l'énumération {@link PieceType}.
     */
    public PieceType getPieceType() {
        return pieceType;
    }
}

package chess.pieces;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

import chess.Board;
import chess.PieceType;
import chess.PlayerColor;
import chess.Square;
import chess.moves.DefaultPathValidator;
import chess.moves.MoveType;
import chess.moves.PathValidator;

```

```

/**
 * Représente la pièce Reine dans le jeu d'échecs.
 * La Reine peut se déplacer sur n'importe quelle case en ligne droite ou en diagonale, sur toute la longueur
 * de la grille,
 * tant que son chemin est libre.
 */
public class Queen extends Piece {

    private final PathValidator pathValidator = new DefaultPathValidator();

    /**
     * Constructeur pour initialiser la Reine avec sa couleur.
     *
     * @param color la couleur de la Reine (blanc ou noir).
     */
    public Queen(PlayerColor color) {
        super(color, PieceType.QUEEN);
    }

    /**
     * Vérifie si la Reine peut se déplacer de la case de départ à la case d'arrivée.
     * La Reine peut se déplacer horizontalement, verticalement ou en diagonale,
     * tant que le chemin est dégagé.
     *
     * @param board le plateau de jeu sur lequel le mouvement est effectué.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @return true si le mouvement est valide, false sinon.
     */
    @Override
    public boolean canMove(Board board, Square start, Square end) {
        int deltaX = distanceX(end);
        int deltaY = distanceY(end);

        // La reine peut se déplacer en diagonale, horizontalement, ou verticalement
        if (!MoveType.DIAGONAL.isValid(deltaX, deltaY) &&
            !MoveType.HORIZONTAL.isValid(deltaX, deltaY) &&
            !MoveType.VERTICAL.isValid(deltaX, deltaY)) {
            return false;
        }

        // Vérification du chemin via le PathValidator
        return pathValidator.isPathClear(board, start, end) && super.canMove(board, start, end);
    }

    /**
     * Définit la case où la Reine se trouve actuellement sur le plateau de jeu.
     * Cette méthode est utilisée pour mettre à jour la position de la Reine après un mouvement.
     *
     * @param square la case de destination où la Reine se déplacera.
     */
    @Override
    public void setSquare(Square square) {
        super.setSquare(square);
    }

    // La reine n'a pas de mouvements spéciaux comme le roque ou la promotion, donc pas de SpecialMove à
    implémenter
}

package chess.pieces;

/**

```



```
* @author Lestiboudois Maxime & Parisod Nathan
* @date 09/01/2025
*/
```

```
import chess.Board;
import chess.PieceType;
import chess.PlayerColor;
import chess.Square;
```

```
/**
 * Représente une pièce du jeu d'échecs (roi, reine, fou, cavalier, tour ou pion).
 * Une pièce a une couleur, un type (roi, reine, etc.) et une position sur l'échiquier.
 * Les méthodes de cette classe permettent de vérifier la validité des mouvements, d'exécuter les déplacements,
 * et de gérer les interactions avec d'autres pièces sur l'échiquier.
 */
```

```
public abstract class Piece {
    private final PlayerColor color;
    private final PieceType type;
    private Square square;

    /**
     * Constructeur pour initialiser une pièce avec une couleur et un type.
     *
     * @param color la couleur de la pièce (blanc ou noir).
     * @param type le type de la pièce (roi, reine, etc.).
     */
    protected Piece(PlayerColor color, PieceType type) {
        this.color = color;
        this.type = type;
    }

    /**
     * Retourne la couleur de la pièce.
     *
     * @return la couleur de la pièce.
     */
    public PlayerColor getColor() {
        return color;
    }

    /**
     * Retourne le type de la pièce (roi, reine, etc.).
     *
     * @return le type de la pièce.
     */
    public PieceType getType() {
        return type;
    }

    /**
     * Retourne la case actuelle où se trouve la pièce sur l'échiquier.
     *
     * @return la case de la pièce.
     */
    public Square getSquare() {
        return square;
    }

    /**
     * Définit la case sur laquelle se trouve la pièce.
     *
     * @param square la nouvelle case où la pièce sera placée.
     */
    public void setSquare(Square square) {
```

```

        this.square = square;
    }

    /**
     * Exécute le mouvement de la pièce si le déplacement est valide.
     * Supprime une pièce capturée et déplace la pièce vers la case d'arrivée.
     *
     * @param board l'échiquier représentant l'état actuel du jeu.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @throws IllegalArgumentException si le mouvement est invalide.
     */
    public void executeMove(Board board, Square start, Square end) {
        if (canMove(board, start, end)) {
            board.removeCapturedPiece(end);
            board.movePiece(this, end);
            setSquare(end);
        } else {
            throw new IllegalArgumentException("Mouvement invalide.");
        }
    }

    /**
     * Calcule la distance horizontale (sur l'axe X) entre la pièce actuelle et la case donnée en valeur absolue.
     *
     * @param square la case de destination.
     * @return la distance horizontale entre la pièce et la case.
     */
    public int distanceX(Square square){
        int x = square.getX();
        int currentX = this.square.getX();

        // Calcul de la distance
        return Math.abs(x - currentX);
    }

    /**
     * Calcule la distance verticale (sur l'axe Y) entre la pièce actuelle et la case donnée en valeur absolue.
     *
     * @param square la case de destination.
     * @return la distance verticale entre la pièce et la case.
     */
    public int distanceY(Square square){
        int y = square.getY();
        int currentY = this.square.getY();

        // Calcul de la distance
        return Math.abs(y - currentY);
    }

    /**
     * Vérifie si la pièce donnée a une couleur différente de celle de la pièce actuelle.
     *
     * @param piece la pièce à comparer.
     * @return {@code true} si les pièces ont des couleurs différentes, {@code false} sinon.
     */
    public boolean isNotSameColor(Piece piece) {
        return !this.color.equals(piece.color);
    }

    /**
     * Vérifie si la pièce peut se déplacer de la case de départ à la case d'arrivée.
     * Une pièce peut se déplacer vers une case vide ou capturer une pièce adverse.

```

```

*
* @param board l'échiquier représentant l'état actuel du jeu.
* @param start la case de départ.
* @param end la case d'arrivée.
* @return {@code true} si le déplacement est possible, {@code false} sinon.
*/
public boolean canMove(Board board, Square start, Square end){
    if(end.isOccupied()) {
        return isNotSameColor(end.getPiece());
    }
    return true;
}
/**
 * Interface représentant une stratégie de mouvement pour les pièces.
 * Chaque stratégie définit si un mouvement est valide et comment l'exécuter.
 */
public interface MoveStrategy {
    /**
     * Vérifie si le mouvement de la pièce selon la stratégie est valide.
     *
     * @param board l'échiquier.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     * @return {@code true} si le mouvement est valide, {@code false} sinon.
     */
    boolean isValid(Board board, Square start, Square end);
    /**
     * Exécute le mouvement de la pièce selon la stratégie.
     *
     * @param board l'échiquier.
     * @param start la case de départ.
     * @param end la case d'arrivée.
     */
    void execute(Board board, Square start, Square end);
}
}

```

~/Documents/HEIG/24-25/P00/Lab08/all.java

```
package chess;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

import chess.pieces.King;
import chess.pieces.Piece;

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

/**
 * Représente l'échiquier du jeu d'échecs.
 * Contient les cases, les pièces, et les fonctionnalités permettant de manipuler les pièces et
 * de vérifier des états spécifiques comme les cases sous attaque.
 */
public class Board {

    private static final int NB_SQUARES_PER_LINE = 8;
    private final Square[][] board = new Square[NB_SQUARES_PER_LINE][NB_SQUARES_PER_LINE];
    private GameController gameController; // Référence au contrôleur

    /**
     * Constructeur de la classe Board.
     * Initialise l'échiquier avec des cases vides.
     */
    public Board() {
        reset();
    }

    /**
     * Définit le contrôleur du jeu pour ce plateau.
     *
     * @param gameController l'instance du contrôleur de jeu.
     */
    public void setGameController(GameController gameController) {
        this.gameController = gameController;
    }

    /**
     * Obtient le contrôleur de jeu associé au plateau.
     *
     * @return l'instance du contrôleur de jeu.
     */
    public GameController getGameController() {
        return gameController;
    }

    /**
     * Récupère une pièce à une position donnée sur l'échiquier.
     *
     * @param x la position horizontale (0-7).
     * @param y la position verticale (0-7).
     * @return la pièce à la position donnée, ou null si la case est vide.
     * @throws IllegalArgumentException si les coordonnées sont hors limites.
     */
    public Piece getPiece(int x, int y) {
        if (x < 0 || x >= NB_SQUARES_PER_LINE || y < 0 || y >= NB_SQUARES_PER_LINE) {
            throw new IllegalArgumentException("Les coordonnées sont hors du plateau.");
        }
    }
}
```

```

    }
    return board[x][y].getPiece();
}

/**
 * Déplace une pièce sur l'échiquier vers une destination donnée.
 *
 * @param piece la pièce à déplacer.
 * @param destination la case cible.
 */
public void movePiece(Piece piece, Square destination) {
    Square currentSquare = piece.getSquare();
    currentSquare.setPiece(null);
    destination.setPiece(piece);
}

/**
 * Supprime une pièce capturée de la case donnée.
 *
 * @param destination la case où une pièce est capturée.
 */
public void removeCapturedPiece(Square destination) {
    if (destination.isOccupied()) {
        destination.setPiece(null);
    }
}

/**
 * Réinitialise l'échiquier en vidant toutes les cases.
 */
public void reset() {
    for (int i = 0; i < NB_SQUARES_PER_LINE; ++i) {
        for (int j = 0; j < NB_SQUARES_PER_LINE; ++j) {
            board[i][j] = new Square(i, j);
        }
    }
}

/**
 * Récupère une case spécifique à une position donnée.
 *
 * @param x la position horizontale (0-7).
 * @param y la position verticale (0-7).
 * @return la case à la position donnée.
 * @throws IllegalArgumentException si les coordonnées sont hors limites.
 */
public Square getSquare(int x, int y) {
    if (x < 0 || x >= NB_SQUARES_PER_LINE || y < 0 || y >= NB_SQUARES_PER_LINE) {
        throw new IllegalArgumentException("Les coordonnées sont hors du plateau.");
    }
    return board[x][y];
}

/**
 * Vérifie si une case est sous attaque par une pièce ennemie.
 *
 * @param square la case à vérifier.
 * @param color la couleur du joueur qui défend la case.
 * @return true si la case est attaquée, false sinon.
 */
public boolean isSquareUnderAttack(Square square, PlayerColor color) {
    List<Piece> enemyPieces = getAllPiecesOfColor(color.opposite());

    for (Piece piece : enemyPieces) {

```

```

        if (piece.canMove(this, piece.getSquare(), square)) {
            return true;
        }
    }
    return false;
}

/**
 * Récupère toutes les pièces d'une couleur donnée, sauf les rois.
 *
 * @param color la couleur des pièces à récupérer.
 * @return une liste contenant toutes les pièces de la couleur donnée.
 */
private List<Piece> getAllPiecesOfColor(PlayerColor color) {
    List<Piece> pieces = new ArrayList<>();
    for (int i = 0; i < NB_SQUARES_PER_LINE; ++i) {
        for (int j = 0; j < NB_SQUARES_PER_LINE; ++j) {
            Piece piece = board[i][j].getPiece();
            if (piece != null && piece.getColor() == color && !(piece instanceof King)) {
                pieces.add(piece);
            }
        }
    }
    return pieces;
}
}

```

```
package chess;
```

```

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

import chess.pieces.*;

/**
 * Contrôleur principal pour le jeu d'échecs.
 * Gère les interactions entre le modèle (échiquier et pièces), la vue (interface utilisateur)
 * et la logique de jeu (mouvements, vérification des règles, etc.).
 */
public class GameController implements ChessController {

    private Board board;
    private ChessView view;
    private Move lastMove; // Attribut pour garder en mémoire le dernier coup joué

    /**
     * Constructeur de la classe GameController.
     * Initialise le plateau de jeu et le dernier coup joué.
     */
    public GameController() {
        board = new Board();
        lastMove = null;
        board.setGameController(this); // Initialisation du dernier coup joué
    }

    /**
     * Démarre une nouvelle partie et initialise la vue.
     *
     * @param view l'interface utilisateur pour le jeu.
     */
    @Override

```

```

public void start(ChessView view) {
    this.view = view;
    view.startView();
    this.newGame();
}

/**
 * Vérifie si le roi d'une couleur donnée est en échec.
 *
 * @param color la couleur du roi à vérifier.
 * @return true si le roi est en échec, false sinon.
 */
public boolean isKingInCheck(PlayerColor color) {
    Square kingSquare = findKing(color);
    return board.isSquareUnderAttack(kingSquare, color);
}

/**
 * Recherche la position du roi d'une couleur donnée.
 *
 * @param color la couleur du roi à localiser.
 * @return la case où se trouve le roi.
 * @throws IllegalStateException si le roi n'est pas trouvé sur l'échiquier.
 */
private Square findKing(PlayerColor color) {
    for (int x = 0; x < 8; x++) {
        for (int y = 0; y < 8; y++) {
            Piece piece = board.getSquare(x, y).getPiece();
            if (piece != null && piece.getType() == PieceType.KING && piece.getColor() == color) {
                return board.getSquare(x, y);
            }
        }
    }
    throw new IllegalStateException("King not found on the board");
}

/**
 * Tente de déplacer une pièce d'une case à une autre.
 *
 * @param fromX la coordonnée x de la case de départ.
 * @param fromY la coordonnée y de la case de départ.
 * @param toX la coordonnée x de la case de destination.
 * @param toY la coordonnée y de la case de destination.
 * @return true si le mouvement est valide et exécuté, false sinon.
 */
@Override
public boolean move(int fromX, int fromY, int toX, int toY) {
    Square fromSquare = board.getSquare(fromX, fromY);
    Square toSquare = board.getSquare(toX, toY);

    if (fromSquare.getPiece() == null || (lastMove == null && fromSquare.getPiece().getColor() !=
PlayerColor.WHITE) || lastMove != null && !lastMove.getPiece().isNotSameColor(fromSquare.getPiece())) {
        return false;
    }

    // Vérification des conditions de déplacement
    if (fromSquare.isOccupied()) {
        Piece piece = fromSquare.getPiece();
        if (piece.canMove(board, fromSquare, toSquare)) {

            //Simuler le mouvement
            Piece capturedPiece = toSquare.getPiece();
            board.movePiece(piece, toSquare);
            boolean isInCheck = isKingInCheck(piece.getColor());

```

```

        //Annuler le mouvement
        board.movePiece(piece, fromSquare);
        toSquare.setPiece(capturedPiece);

        if (!isInCheck) {
            piece.executeMove(board, fromSquare, toSquare);

            // Mise à jour du dernier coup joué
            lastMove = new Move(fromSquare, toSquare, piece);

            view.removePiece(fromX, fromY);
            view.putPiece(toSquare.getPiece().getType(), toSquare.getPiece().getColor(), toX, toY);

            return true;
        }
    }
    return false;
}

/**
 * Retire une pièce de la vue à une position donnée.
 *
 * @param x la coordonnée x de la pièce à retirer.
 * @param y la coordonnée y de la pièce à retirer.
 */
public void removePiece(int x, int y) {
    view.removePiece(x, y);
}

/**
 * Place une pièce dans la vue à une position donnée.
 *
 * @param x la coordonnée x de la pièce à placer.
 * @param y la coordonnée y de la pièce à placer.
 */
public void setPiece(int x, int y) {
    view.putPiece(board.getPiece(x, y).getType(), board.getPiece(x, y).getColor(), x, y);
}

/**
 * Démarre une nouvelle partie en réinitialisant le plateau et les pièces.
 */
@Override
public void newGame() {
    board.reset();// Réinitialisation de l'échiquier
    setLastMoveAtNull();

    // Placement des pièces blanches
    for (int i = 0; i < 8; i++) {
        board.getSquare(i, 1).setPiece(new Pawn(PlayerColor.WHITE));
    }
    board.getSquare(0, 0).setPiece(new Rook(PlayerColor.WHITE));
    board.getSquare(7, 0).setPiece(new Rook(PlayerColor.WHITE));
    board.getSquare(1, 0).setPiece(new Knight(PlayerColor.WHITE));
    board.getSquare(6, 0).setPiece(new Knight(PlayerColor.WHITE));
    board.getSquare(2, 0).setPiece(new Bishop(PlayerColor.WHITE));
    board.getSquare(5, 0).setPiece(new Bishop(PlayerColor.WHITE));
    board.getSquare(3, 0).setPiece(new Queen(PlayerColor.WHITE));
    board.getSquare(4, 0).setPiece(new King(PlayerColor.WHITE));

    // Placement des pièces noires
    for (int i = 0; i < 8; i++) {

```



```

        board.getSquare(i, 6).setPiece(new Pawn(PlayerColor.BLACK));
    }
    board.getSquare(0, 7).setPiece(new Rook(PlayerColor.BLACK));
    board.getSquare(7, 7).setPiece(new Rook(PlayerColor.BLACK));
    board.getSquare(1, 7).setPiece(new Knight(PlayerColor.BLACK));
    board.getSquare(6, 7).setPiece(new Knight(PlayerColor.BLACK));
    board.getSquare(2, 7).setPiece(new Bishop(PlayerColor.BLACK));
    board.getSquare(5, 7).setPiece(new Bishop(PlayerColor.BLACK));
    board.getSquare(3, 7).setPiece(new Queen(PlayerColor.BLACK));
    board.getSquare(4, 7).setPiece(new King(PlayerColor.BLACK));

    // Mise à jour de la vue
    for (int x = 0; x < 8; x++) {
        for (int y = 0; y < 8; y++) {
            Piece piece = board.getSquare(x, y).getPiece();
            if (piece != null) {
                view.putPiece(piece.getType(), piece.getColor(), x, y);
            }
        }
    }
}

/**
 * Gère la promotion d'un pion.
 * Demande à l'utilisateur de choisir une pièce pour remplacer le pion.
 *
 * @param square la case où le pion est promu.
 */
public void promotePawn(Square square) {
    PromotablePiece[] promotionChoices = {new PromotablePiece(PieceType.QUEEN), new
PromotablePiece(PieceType.ROOK), new PromotablePiece(PieceType.BISHOP), new PromotablePiece(PieceType.KNIGHT)}
;
    PromotablePiece choice = this.view.askUser("Promotion", "Choisissez une pièce pour la promotion :",
promotionChoices);
    switch (choice.getPieceType()) {
        case ROOK:
            square.setPiece(new Rook(square.getPiece().getColor()));
            break;
        case BISHOP:
            square.setPiece(new Bishop(square.getPiece().getColor()));
            break;
        case KNIGHT:
            square.setPiece(new Knight(square.getPiece().getColor()));
            break;
        default:
            square.setPiece(new Queen(square.getPiece().getColor()));
            break;
    }
}

/**
 * Classe interne représentant un mouvement.
 */
public static class Move {
    private final Square from;
    private final Square to;
    private final Piece piece;

    public Move(Square from, Square to, Piece piece) {
        this.from = from;
        this.to = to;
        this.piece = piece;
    }

    public Square getFrom() {

```

```

        return from;
    }

    public Square getTo() {
        return to;
    }

    public Piece getPiece() {
        return piece;
    }
}

/**
 * Obtient le dernier coup joué.
 *
 * @return le dernier mouvement.
 */
public Move getLastMove() {
    return lastMove;
}

/**
 * Réinitialise le dernier mouvement à null.
 */
public void setLastMoveAtNull() {
    lastMove = null;
}
}

```

```
package chess;
```

```

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

/**
 * Énumération représentant les types de pièces d'échecs.
 */
public enum PieceType {
    PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING
}

```

```
package chess;
```

```

/**
 * @modified by Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

public enum PlayerColor {
    WHITE, BLACK;

    /**
     * Retourne la couleur opposée.
     *
     * @return PlayerColor opposé (WHITE devient BLACK et vice-versa).
     */
    public PlayerColor opposite() {
        return this == WHITE ? BLACK : WHITE;
    }
}

```

```
}
```

```
package chess;
```

```
/**
```

```
 * @author Lestiboudois Maxime & Parisod Nathan
```

```
 * @date 09/01/2025
```

```
 */
```

```
import chess.pieces.Piece;
```

```
/**
```

```
 * Représente une case sur l'échiquier.
```

```
 * Chaque case est définie par ses coordonnées (x, y) et peut contenir une pièce.
```

```
 */
```

```
public class Square {
```

```
    private final int x;
```

```
    private final int y;
```

```
    private Piece piece;
```

```
    /**
```

```
     * Constructeur d'une case à des coordonnées spécifiques.
```

```
     *
```

```
     * @param x Coordonnée X de la case
```

```
     * @param y Coordonnée Y de la case
```

```
     */
```

```
    public Square(int x, int y) {
```

```
        this.x = x;
```

```
        this.y = y;
```

```
        this.piece = null;
```

```
    }
```

```
    /**
```

```
     * Vérifie si la case est occupée par une pièce.
```

```
     *
```

```
     * @return true si une pièce occupe la case, false sinon.
```

```
     */
```

```
    public boolean isOccupied() {
```

```
        return piece != null;
```

```
    }
```

```
    /**
```

```
     * Obtient la pièce sur la case.
```

```
     *
```

```
     * @return La pièce présente ou null si la case est vide.
```

```
     */
```

```
    public Piece getPiece() {
```

```
        return piece;
```

```
    }
```

```
    /**
```

```
     * Place une pièce sur la case.
```

```
     * Met également à jour la case associée à la pièce.
```

```
     *
```

```
     * @param piece La pièce à placer (null pour vider la case).
```

```
     */
```

```
    public void setPiece(Piece piece) {
```

```
        this.piece = piece;
```

```
        if (piece != null) {
```

```
            piece.setSquare(this);
```

```
        }
```

```
    }
```

```

/**
 * Obtient la coordonnée X de la case.
 *
 * @return La coordonnée X.
 */
public int getX() {
    return x;
}

/**
 * Obtient la coordonnée Y de la case.
 *
 * @return La coordonnée Y.
 */
public int getY() {
    return y;
}
}

package chess;

/**
 * @author Lestiboudois Maxime & Parisod Nathan
 * @date 09/01/2025
 */

import chess.views.console.ConsoleView;
import chess.views.gui.GUIView;

public class Main {
    public static void main(String[] args) {
        ChessController controller = new GameController();
        ChessView view = new GUIView(controller);
        //ChessView view = new ConsoleView(controller); //mode console
        controller.start(view);
    }
}

```