```java
/**
 *  ------------------------------------------------------------------------------------
 * @Fichier      : Main.java
 * @Labo         : Laboratoire 8 : Chess
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Ce programme permet à deux personnes de s'affronter au jeu des échecs.
 *                 Une interface graphique est présente. Elle permet de représenter un
 *                 échiquier ainsi que le matériel nécessaire pour jouer une partie.
 *
 * @Remarque     : Les matches nuls par Pat ou par impossibilité de mater ne sont pas
 *                 implémenter.
 * @Modification : / Aucune modification
 *  ------------------------------------------------------------------------------------
 **/

import chess.ChessController;
import chess.ChessView;
import chess.views.gui.GUIView;

public class Main {
    public static void main(String[] args) {
        // 1. Création du contrôleur pour gérer le jeu d'échec
        ChessController controller = new engine.ChessController(); // Instancier un ChessController

        // 2. Création de la vue
        ChessView view = new GUIView(controller); // mode GUI
        // = new ConsoleView(controller); // mode Console

        // 3. Lancement du programme.
        controller.start(view);
    }
}
```

```java
package engine;

import chess.ChessView;
import chess.PieceType;
import chess.PlayerColor;
import engine.util.Coord;
import engine.util.PromotionChoice;
import java.util.LinkedList;
import java.util.List;

/**
 *  -------------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Cette classe permet de gérer toute l'intelligence nécessaire pour un jeu d'échecs.
 *                 C'est dans cette classe qu'est codée la gestion du rock, de la prise en passant
 *                 ou encore des échecs.
 *  -------------------------------------------------------------------------------
 **/

public class ChessController implements chess.ChessController {
    // region Parameter
    private final int SIZE = 8;
    // indique sur quelle colonne il y a eu un départ de 2 cases pour un pion
    private int pawnJumpStart = -1;
    private ChessView view;
    private PlayerColor turn;
    private Piece[][] board;
    private boolean checkMate;
    // endregion

    // region Methods
    /**
     * Nom          : start
     * Description  : Méthode qui initialise la vue et démarre une nouvelle partie
     * @param view  : la vue à utiliser
     */
    @Override
    public void start(ChessView view) {
        this.view = view;
        view.startView();
        newGame();
    }

    /**
     * Nom          :
     * Description  : Méthode qui gère les mouvements
     * @param fromX : position x de départ
     * @param fromY : position y de départ
     * @param toX   : position x de déstination
     * @param toY   : position y de déstination
     * @return      : boolean qui indique si un mouvement a été éffectué
     */
    @Override
    public boolean move(int fromX, int fromY, int toX, int toY) {
        if (checkMate) {
            view.displayMessage("Game finished, please start again");
            return false;
        }

        // si il n'y a pas de piece sur cette case. on ne pourra pas se déplacer
        if (!acceptMove(fromX, fromY, toX, toY)) {
            return false;
        }
        List<List<Coord>> moves = refactorListMove(getMoves(fromX, fromY, toX, toY));
        if (!findCoordInListMove(moves, toX, toY)) return false;
        // si gestion castle return false ça veut dire qu'on fait pas un roque (pas de déplacement)
        if (!gestionCastle(fromX, fromY, toX, toY)) {
            gestionEnPassant(fromX, fromY, toX, toY);
            movePiece(fromX, fromY, toX, toY);
            gestionPromotion(toX, toY);
        }

        // On vérifie qu'on ne se mette pas en échec tout seul
        if (checkSelfMat()) {
            view.displayMessage("Prohibited movement !");
```

```
            movePiece(toX, toY, fromX, fromY);
            return false;
        }

        // On regarde s'il y a échec et remplie une liste de toutes les pièces qui mettent échec
        LinkedList<Coord> attackers = new LinkedList<>();
        LinkedList<Coord> defenders = new LinkedList<>();
        if (checkMat(attackers)) {
            view.displayMessage("The " + (turn == PlayerColor.WHITE ? "BLACK" : "WHITE") + " king is in
            check !");

            canProtectHisKingByEating(attackers, defenders);
            canCoverHisKing(attackers, defenders);
            canMoveHisKing(attackers);
            canStrengthenAttack(attackers);

            if (defenders.size() < attackers.size()) {
                checkMate = true;
                view.displayMessage("Checkmate, " + (turn != PlayerColor.WHITE ? "BLACK" : "WHITE") + "
                wins !");
                return true;
            }
        }

        changeTurn();
        return true;
    }

    /**
     * Nom              :
     * Description      : Méthode qui va chercher un coordonnée dans une liste de liste de coordonnées
     * @param listMove  : liste de liste dans laquelle on cherche la coord (toX,toY)
     * @param toX       : postion x qu'on cherche
     * @param toY       : postion y qu'on cherche
     * @return          : si on a trouvé la coordonnée dans les listes
     */
    private boolean findCoordInListMove(List<List<Coord>> listMove, int toX, int toY) {
        Coord find = new Coord(toX, toY);
        for (List<Coord> list : listMove) {
            for (Coord c : list) {
                if (find.isEqual(c)) {
                    return true;
                }
            }
        }
        return false;
    }

    /**
     * Nom              : refactorListMove
     * Description      : Méthiode qui va enlever les mouvements illicites
     * @param listMove  : la liste de mouvement que l'on veut remanier
     * @return          : la nouvelle liste sans les mouvements illicites
     */
    private List<List<Coord>> refactorListMove(List<List<Coord>> listMove) {
        List<List<Coord>> refactorListMove = new LinkedList<>();
        List<Coord> refactoredVect;
        for (List<Coord> vect : listMove) {
            refactoredVect = new LinkedList<>();
            for (Coord c : vect) {
                if (board[c.getX()][c.getY()] != null) {
                    if (board[c.getX()][c.getY()].getColor() != turn) {
                        refactoredVect.add(c);
                        if (vect.indexOf(c) < vect.size() && board[c.getX()][c.getY()].type != PieceType.
                        KNIGHT) {
                            break;
                        }
                    } else if (board[c.getX()][c.getY()].type == PieceType.KING) {
                        refactoredVect.add(c);
                    } else {
                        break;
                    }
                } else {
                    refactoredVect.add(c);
                }
            }
            if (!refactoredVect.isEmpty()) {
```

```
                refactorListMove.add(refactoredVect);
            }
        }
        return refactorListMove;
    }

    /**
     * Nom          :
     * Description  : Méthode qui indique si un mouvement est correct,
     *                si il y a une pièce à la coordonnées (fromX,fromY)
     *                si la pièce à la bonne couleur
     *                si cette pièce peut se déplacer vers (toX,toY)
     *                si la destination n'est pas occupé ou est occupé par une pièce adverse
     * @param fromX : position x de départ
     * @param fromY : position y de départ
     * @param toX   : position x de déstination
     * @param toY   : position y de déstination
     * @return      : si le mouvement est accepté
     */
    private boolean acceptMove(int fromX, int fromY, int toX, int toY) {
        if (board[fromX][fromY] == null) return false;
        // si on essaie de deplacer une pièce de la mauvaise couleur
        if (board[fromX][fromY].getColor() != turn) return false;
        // si on ne peut pas se déplacer à la destination on indique cela
        if (!board[fromX][fromY].acceptedMove(toX, toY)) return false;
        // si on essaie de se déplacer sur une pièce de la même couleur que nous
        return board[toX][toY] == null || board[toX][toY].getColor() != turn;
    }

    /**
     * Nom          :
     * Description  : Méthode qui retourne la liste de mouvements correct pour une pièce donné
     * @param fromX : position x de départ
     * @param fromY : position y de départ
     * @param toX   : position x de déstination
     * @param toY   : position y de déstination
     * @return      : la liste de mouvements de la pièce qui se trouve à (fromX,fromY)
     */
    private List<List<Coord>> getMoves(int fromX, int fromY, int toX, int toY) {
        if (board[toX][toY] == null) {
            if (board[fromX][fromY].getType() == PieceType.PAWN && toX == pawnJumpStart && fromY == (turn
             == PlayerColor.WHITE ? 4 : 3)) {
                return board[fromX][fromY].listEatingMove();
            } else {
                return board[fromX][fromY].listMove();
            }
        } else {
            return board[fromX][fromY].listEatingMove();
        }
    }

    /**
     * Nom          : gestionEnPassant
     * Description  : Méthode qui gère la prise en passant
     * @param fromX : position x de départ
     * @param fromY : position y de départ
     * @param toX   : position x de déstination
     * @param toY   : position y de déstination
     */
    private void gestionEnPassant(int fromX, int fromY, int toX, int toY) {
        // check enpassant
        if (board[fromX][fromY].getType() == PieceType.PAWN && toX == pawnJumpStart && fromY == (turn ==
        PlayerColor.WHITE ? 4 : 3)) {
            view.removePiece(pawnJumpStart, (turn == PlayerColor.WHITE ? 4 : 3));
            board[pawnJumpStart][(turn == PlayerColor.WHITE ? 4 : 3)] = null;
        }
        if (board[fromX][fromY].getType() == PieceType.PAWN && Math.abs(fromY - toY) == 2) {
            pawnJumpStart = fromX;
        } else {
            pawnJumpStart = -1;
        }
    }

    /**
     * Nom          : gestionCastle
     * Description  : Méthode qui gère le grand et petit roque
     * @param fromX : position x de départ
```

```
     * @param fromY : position y de départ
     * @param toX   : position x de déstination
     * @param toY   : position y de déstination
     * @return        si on effectue un roque
     */
    private boolean gestionCastle(int fromX, int fromY, int toX, int toY) {
        // si on est en échec
        if(checkSelfMat()){
            // on ne peut pas effectué de roque
            return false;
        }
        // check si on a un roi et une tour de la même couleur et qui n'ont pas bougé
        if (board[fromX][fromY].getType() != PieceType.KING
                || board[toX][toY] != null
                || board[fromX][fromY].hasMoved
                || fromY != toY
                || !(toX == 1 || toX == 6)) return false;
        int castleX = (toX == Math.min(fromX, toX) ? 0 : 7);
        if (!(board[castleX][fromY] != null
                && board[castleX][fromY].getType() == PieceType.ROOK
                && board[castleX][fromY].getColor() == board[fromX][fromY].getColor()
                && !board[castleX][fromY].hasMoved)) return false;
        int left = Math.min(fromX, castleX);
        int right = Math.max(fromX, castleX);
        // vérif si la voie est libre entre le roi et la tour
        for (int i = left + 1; i < right; ++i) {
            if (board[i][fromY] != null) return false;
        }
        // on peut effectuer le rock
        // bouge le roi
        movePiece(fromX, fromY, toX, toY);
        // bouge la tour
        movePiece(castleX, fromY, toX + ((fromX - toX) / 2), toY);
        return true;
    }

    /**
     * Nom          : changeTurn
     * Description  : Méthode qui change le tour
     */
    private void changeTurn() {
        turn = (turn == PlayerColor.WHITE ? PlayerColor.BLACK : PlayerColor.WHITE);
    }

    /**
     * Nom          :
     * Description  : Méthode qui gère le deplacement dans la view et dans le tableau
     * @param fromX : position x de départ
     * @param fromY : position y de départ
     * @param toX   : position x de déstination
     * @param toY   : position y de déstination
     */
    private void movePiece(int fromX, int fromY, int toX, int toY) {
        // déplacer la pièce au bon endroit
        board[fromX][fromY].move(toX, toY);
        board[toX][toY] = board[fromX][fromY];
        board[fromX][fromY] = null;
        view.removePiece(fromX, fromY);
        view.putPiece(board[toX][toY].getType(), board[toX][toY].getColor(), toX, toY);
    }

    /**
     * Nom          : newGame
     * Description  : Méthode qui initialse une nouvelle partie
     */
    @Override
    public void newGame() {
        cleanGUI();
        checkMate = false;
        board = new Piece[SIZE][SIZE];
        turn = PlayerColor.WHITE;
        setBoard();
        setGUI();
    }

    /**
     * Nom          : setBoard
```

```
     * Description  : Méthode qui initialise le tableau de jeu
     */
    private void setBoard() {
        if (board == null) return;
        PlayerColor c = PlayerColor.WHITE;
        for (int i = 0; i < SIZE; ++i) {
            for (int j = 0; j < SIZE; ++j) {
                board[i][j] = null;
            }
        }
        for (int i = 0; i < 2; ++i) {
            board[0][(i * 7)] = new Rook(new Coord(0, i * 7), c);
            board[1][(i * 7)] = new Knight(new Coord(1, i * 7), c);
            board[2][(i * 7)] = new Bishop(new Coord(2, i * 7), c);
            board[3][(i * 7)] = new Queen(new Coord(3, i * 7), c);
            board[4][(i * 7)] = new King(new Coord(4, i * 7), c);
            board[5][(i * 7)] = new Bishop(new Coord(5, i * 7), c);
            board[6][(i * 7)] = new Knight(new Coord(6, i * 7), c);
            board[7][(i * 7)] = new Rook(new Coord(7, i * 7), c);
            for (int j = 0; j < SIZE; ++j) {
                board[j][1 + 5 * i] = new Pawn(new Coord(j, 1 + i * 5), c);
            }
            c = PlayerColor.BLACK;
        }
    }

    /**
     * Nom          : cleanGUI
     * Description  : Méthode qui nettoye la view
     */
    private void cleanGUI() {
        if (view == null) return;
        for (int i = 0; i < SIZE; ++i) {
            for (int j = 0; j < SIZE; ++j) {
                view.removePiece(i, j);
            }
        }
    }

    /**
     * Nom          : setGUI
     * Description  : Méthode qui initialise la view avec les pièces
     */
    private void setGUI() {
        if (view == null || board == null) return;
        for (int i = 0; i < SIZE; ++i) {
            for (int j = 0; j < SIZE; ++j) {
                if (board[i][j] == null) continue;
                view.putPiece(board[i][j].getType(), board[i][j].getColor(), i, j);
            }
        }
    }

    /**
     * Nom         : gestionPromotion
     * Description : Vérifie si la promotion à lieu d'être et échange le pion promu avec la pièce
     *               choisie avec l'utilisateur.
     * @param toX  : Coordonné en X ou le pion a été déplacé et ou il faut créer la nouvelle pièce
     * @param toY  : Coordonné en Y ou le pion a été déplacé et ou il faut créer la nouvelle pièce
     * @return     : / void
     **/
    private void gestionPromotion(int toX, int toY) {
        if (board[toX][toY].getType() == PieceType.PAWN) {
            if (turn == PlayerColor.WHITE && toY == SIZE - 1 || turn == PlayerColor.BLACK && toY == 0) {
                view.removePiece(toX, toY);
            } else return;
        } else return;

        PromotionChoice[] options = {
                new PromotionChoice() {
                    @Override
                    public String textValue() {
                        return "Queen";
                    }

                    @Override
                    public Piece create() {
```

```
                    return new Queen(new Coord(toX, toY), turn);
                }
            },
            new PromotionChoice() {
                @Override
                public String textValue() {
                    return "Rook";
                }

                @Override
                public Piece create() {
                    return new Rook(new Coord(toX, toY), turn);
                }
            },
            new PromotionChoice() {
                @Override
                public String textValue() {
                    return "Bishop";
                }

                @Override
                public Piece create() {
                    return new Bishop(new Coord(toX, toY), turn);
                }
            },
            new PromotionChoice() {
                @Override
                public String textValue() {
                    return "Knight";
                }

                @Override
                public Piece create() {
                    return new Knight(new Coord(toX, toY), turn);
                }
            }
    };

    PromotionChoice result = view.askUser("Promotion", "Choose a piece", options);
    Piece p = result.create();
    board[toX][toY] = p;
    view.putPiece(p.getType(), p.getColor(), toX, toY);
}

/**
 * Nom              : checkMat
 * Description      : Vérifie l'on met l'adversaire en échec et mat
 * @param attackers : Liste contenant toutes les pièces qui attaquent le roi
 * @return          : Booléen indiquant si le roi adverse est en échec
 **/
private boolean checkMat(LinkedList<Coord> attackers) {
    // On cherche la position du roi adverse
    List<Coord> kingPos = findKings();
    Coord enemyKingPos = turn == PlayerColor.WHITE ? kingPos.get(1) : kingPos.get(0);

    for (int i = 0; i < SIZE; ++i) {
        for (int j = 0; j < SIZE; ++j) {
            if (board[i][j] != null && board[i][j].color == turn) {
                // On récupère la liste des positions ou on peut manger une pièce adverse
                List<List<Coord>> moves = board[i][j].listEatingMove();
                List<List<Coord>> eatingMoves = refactorListMove(moves);

                // On regarde si une pièce peut manger le roi adverse
                if (findCoordInListMove(eatingMoves, enemyKingPos.getX(), enemyKingPos.getY())) {
                    attackers.add(board[i][j].coord);
                }
            }
        }
    }
    return attackers.size() != 0;
}

/**
 * Nom              : checkSelfMat
 * Description      : Vérifie l'on ne se met pas tout seul en échec à la suite d'un mouvement
 * @return          : Booléen indiquant si notre roi est mis en échec.
 **/
```

```
    private boolean checkSelfMat() {
        // On cherche la position de son roi
        List<Coord> kingPos = findKings();
        Coord allyKingPos = turn != PlayerColor.WHITE ? kingPos.get(1) : kingPos.get(0);

        for (int i = 0; i < SIZE; ++i) {
            for (int j = 0; j < SIZE; ++j) {
                if (board[i][j] != null && board[i][j].color != turn) {
                    // On récupère la liste des positions ou l'adversaire peut nous manger une pièce
                    List<List<Coord>> moves = board[i][j].listEatingMove();
                    List<List<Coord>> eatingMoves = refactorListMove(moves);

                    // On regarde si une pièce peut manger notre roi
                    if (findCoordInListMove(eatingMoves, allyKingPos.getX(), allyKingPos.getY())) {
                        return true;
                    }
                }
            }
        }
        return false;
    }


    /**
     * Nom              : findKings
     * Description      : Permet de trouver la position des rois.
     * @return          : List de coordonnées des rois. Le roi blanc est toujours en tête de liste
     *                    suivi par le roi noir.
     **/
    private List<Coord> findKings() {   // list[0] = white, liste[1] = black
        List<Coord> kingsPos = new LinkedList<>();

        for (int i = 0; i < SIZE; ++i) {
            for (int j = 0; j < SIZE; ++j) {
                if (board[i][j] != null && board[i][j].type == PieceType.KING) {
                    if (board[i][j].color == PlayerColor.WHITE) {
                        if (kingsPos.size() == 2) {
                            kingsPos.add(0, board[i][j].coord);
                            kingsPos.remove(1);
                        } else {
                            kingsPos.add(0, board[i][j].coord);
                        }

                    } else if (board[i][j].color == PlayerColor.BLACK && kingsPos.size() == 0) {
                        Coord temp = new Coord(4, 0);
                        kingsPos.add(0, temp);  // Position par défaut du roi blanc (pour éviter de
                        trier la liste la fin et permet d'avoir toujours le roi blanc à l'index 0. Et
                        permet d'éviter une exception)
                        kingsPos.add(1, board[i][j].coord);
                        continue;
                    } else {
                        kingsPos.add(1, board[i][j].coord);
                    }
                    if (kingsPos.size() == 2) {
                        return kingsPos;
                    }
                }
            }
        }
        return null;
    }

    /**
     * Nom              : canProtectHisKingByEating
     * Description      : Vérifie si on peut protéger son roi en mangeant les attaquants
     * @param attackers : Liste contenant toutes les pièces qui attaquent le roi
     * @param defenders : Liste contenant toutes les pièces alliées qui peuvent protéger le roi
     *                    en mangeant un attaquant.
     * @return          : Booléen indiquant si le roi peut être protégé de tous les attaquants
     **/
    private boolean canProtectHisKingByEating(LinkedList<Coord> attackers, LinkedList<Coord> defenders) {
        for (int i = 0; i < SIZE; ++i) {
            for (int j = 0; j < SIZE; ++j) {
                if (board[i][j] != null && board[i][j].color != turn) {
                    List<List<Coord>> allyMoves = board[i][j].listEatingMove();
                    List<List<Coord>> defensiveMoveByEating = defensiveMoveByEating(allyMoves);
```

```
                for (Coord c : attackers) {
                    if (findCoordInListMove(defensiveMoveByEating, c.getX(), c.getY())) {
                        if (!defenders.contains(board[i][j].coord)) {
                            defenders.add(board[i][j].coord);
                        }
                    }
                }
            }
        }
    }
    return defenders.size() != 0;
}

/**
 * Nom              : defensiveMoveByEating
 * Description      : Cette méthode prend la liste qui contient tous les mouvement possibles d'une
 pièce
 *                    puis cette liste est mise à jour en gardant uniquement les mouvements ou l'on
 peut
 *                    manger une pièce ennemie.
 * @param listMove : Liste de tous les mouvements possibles d'une pièce
 * @return          : La liste mise à jours.
 **/
private List<List<Coord>> defensiveMoveByEating(List<List<Coord>> listMove) {
    List<List<Coord>> refactorListMove = new LinkedList<>();
    List<Coord> refactoredVect;
    for (List<Coord> vect : listMove) {
        refactoredVect = new LinkedList<>();
        for (Coord c : vect) {
            if (board[c.getX()][c.getY()] != null) {
                if (board[c.getX()][c.getY()].getColor() == turn) {
                    refactoredVect.add(c);
                    break;
                }
                if (vect.indexOf(c) < vect.size() && board[c.getX()][c.getY()].type != PieceType.
                KNIGHT) {
                    break;
                }
            }
        }
        if (!refactoredVect.isEmpty()) {
            refactorListMove.add(refactoredVect);
        }
    }
    return refactorListMove;
}

/**
 * Nom              : canCoverHisKing
 * Description      : Vérifie si on peut protéger son roi en s'interposant entre le roi et un
 attaquant.
 * @param attackers : Liste contenant toutes les pièces qui attaquent le roi.
 * @param defenders : Liste contenant toutes les pièces alliées qui peuvent protéger le roi.
 * @return          : Booléen indiquant si le roi peut être protégé de tous les attaquantss
 **/
private boolean canCoverHisKing(LinkedList<Coord> attackers, LinkedList<Coord> defenders) {
    // On cherche la position du roi allié qui est attaqué
    List<Coord> kingPos = findKings();
    Coord allyKingPos = turn == PlayerColor.WHITE ? kingPos.get(1) : kingPos.get(0);

    List<Coord> listWhichContainsKingPos = new LinkedList<>();

    // On génére toute les positions des attaquants du roi
    for (Coord c : attackers) {
        List<List<Coord>> moves = board[c.getX()][c.getY()].listEatingMove();
        List<List<Coord>> eatingMove = refactorListMove(moves);

        // On cherche le vecteur qui va de la pièce attaquante au roi
        for (List<Coord> l : eatingMove) {
            for (Coord ck : l) {
                if (ck.isEqual(allyKingPos)) {
                    listWhichContainsKingPos = l;
                    listWhichContainsKingPos.remove(listWhichContainsKingPos.size() - 1);
                    break;
                }
            }
            if (listWhichContainsKingPos.size() != 0) {
```

```
                    break;
                }
            }

            // Pour toute les pièces alliés, on regarde si une pièce peut s'interrposer entre le roi et
            l'attaquant
            for (int i = 0; i < SIZE; ++i) {
                for (int j = 0; j < SIZE; ++j) {
                    if (board[i][j] != null && board[i][j].color != turn && board[i][j].type != PieceType
                    .KING) {
                        List<List<Coord>> allyMoves = board[i][j].listEatingMove();
                        List<List<Coord>> defensiveMoveByMoving = defensiveMoveByCover(allyMoves); //mskn

                        if (defensiveMoveByMoving.size() == 0) {
                            continue;
                        }

                        for (Coord pos : listWhichContainsKingPos) {
                            if (findCoordInListMove(defensiveMoveByMoving, pos.getX(), pos.getY())) {
                                if (!defenders.contains(board[i][j].coord)) {
                                    defenders.add(board[i][j].coord);
                                }
                            }
                        }
                    }
                }
            }
        }
        return defenders.size() != 0;
    }

    /**
     * Nom              : defensiveMoveByCover
     * Description      : Cette méthode prend la liste qui contient tous les mouvements possibles d'une
     pièce
     *                    puis cette liste est mise à jour en gardant uniquement les mouvements ou l'on
     peut
     *                    couvrir le roi.
     * @param listMove  : Liste de tous les mouvements possibles d'une pièce
     * @return          : La liste mises à jour.
     **/
    private List<List<Coord>> defensiveMoveByCover(List<List<Coord>> listMove) {
        List<List<Coord>> refactorListMove = new LinkedList<>();
        List<Coord> refactoredVect;
        for (List<Coord> vect : listMove) {
            refactoredVect = new LinkedList<>();
            for (Coord c : vect) {
                if (board[c.getX()][c.getY()] == null) {
                    refactoredVect.add(c);
                }
                break;
            }
            if (!refactoredVect.isEmpty()) {
                refactorListMove.add(refactoredVect);
            }
        }
        return refactorListMove;
    }

    /**
     * Nom              : canMoveHisKing
     * Description      : Vérifie si le roi peut se déplacer est se sortir de la mise en échec
     * @param attackers : Liste contenant toutes les pièces qui attaquent le roi
     * @return          : Booléen indiquant si le roi peut se protéger en se déplaçant.
     **/
    private boolean canMoveHisKing(LinkedList<Coord> attackers) {
        // On cherche la position du roi allié qui est attaqué
        List<Coord> kingPos = findKings();
        Coord allyKingPos = turn == PlayerColor.WHITE ? kingPos.get(1) : kingPos.get(0);

        // Génère les déplacements possibles du roi attaqué
        List<List<Coord>> allyMoves = board[allyKingPos.getX()][allyKingPos.getY()].listEatingMove();
        List<List<Coord>> defensiveMoveByMoving = defensiveMoveByCover(allyMoves);

        List<Coord> listWhichContainsKingPos = new LinkedList<>();

        // On génére toute les positions des attaquants du roi
```

```
        for (Coord c : attackers) {
            List<List<Coord>> moves = board[c.getX()][c.getY()].listEatingMove();
            List<List<Coord>> eatingMove = refactorListMove(moves);

            // On cherche le vecteur qui va de la pièce attaquante au roi
            for (List<Coord> l : eatingMove) {
                for (Coord ck : l) {
                    if (ck.isEqual(allyKingPos)) {
                        listWhichContainsKingPos = l;
                        listWhichContainsKingPos.remove(listWhichContainsKingPos.size() - 1);
                        break;
                    }
                }
                if (listWhichContainsKingPos.size() != 0) {
                    break;
                }
            }

            // On enlève les dépalcements du roi qui sont dans le vecteur d'attaque de l'attaquant
            for (List<Coord> l : defensiveMoveByMoving) {
                for (Coord kingMove : l) {
                    for (Coord enemyMove : listWhichContainsKingPos) {
                        if (enemyMove.isEqual(kingMove)) {
                            l.remove(kingMove);

                            if (l.size() == 0) {
                                defensiveMoveByMoving.remove(l);
                            }
                        }
                    }
                }
            }
        }
        return defensiveMoveByMoving.size() > 0;    // Si le roi ne peut pas se déplacer, il est maté
    }

    /**
     * Nom              : canStrengthenAttack
     * Description      : Vérifie si une pièce peut couvrir une pièce qui met échec au roi adverse
     * @param attackers : Liste contenant toutes les pièces qui attaquent le roi adverse
     * @return          : Booléen indiquant si au moins une pièce peut supporter l'attaque.
     **/
    private boolean canStrengthenAttack(LinkedList<Coord> attackers) {
        int nbrAttackers = attackers.size();
        LinkedList<Coord> tmp = new LinkedList<>();
        for (int i = 0; i < SIZE; ++i) {
            for (int j = 0; j < SIZE; ++j) {
                if (board[i][j] != null && board[i][j].color == turn) {
                    List<List<Coord>> allyMoves = board[i][j].listEatingMove();
                    List<List<Coord>> supportAttackMoves = supportAttackMove(allyMoves);
                    for (Coord c : attackers) {
                        for(List<Coord> l : supportAttackMoves) {
                            for(Coord s : l) {
                                if (s.isEqual(c)) {
                                    tmp.add(s);
                                }
                            }
                        }
                    }
                }
            }
        }
        attackers.addAll(tmp);
        return attackers.size() - nbrAttackers != 0;
    }

    /**
     * Nom              : supportAttackMove
     * Description      : Cette méthode prend la liste qui contient tous les mouvements possibles d'une
     pièce
     *                    puis cette liste est mise à jour en gardant uniquement les mouvements ou l'on
     peut
     *                    couvrir une pièce alliée qui attaque le roi adverse.
     * @param listMove  : Liste de tous les mouvements possibles d'une pièce
     * @return          : La liste mises à jour.
     **/
    private List<List<Coord>> supportAttackMove(List<List<Coord>> listMove) {
```

```
        List<List<Coord>> refactorListMove = new LinkedList<>();
        List<Coord> refactoredVect;
        for (List<Coord> vect : listMove) {
            refactoredVect = new LinkedList<>();
            for (Coord c : vect) {
                if (board[c.getX()][c.getY()] != null) {
                    if (board[c.getX()][c.getY()].getColor() == turn) {
                        refactoredVect.add(c);
                    }
                }
            }
            refactorListMove.add(refactoredVect);
        }
        return refactorListMove;
    }
    // en region
}
```

```java
package engine;

import chess.PieceType;
import chess.PlayerColor;
import engine.util.Coord;

import java.util.List;

/**
 *  ---------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Il s'agit d'une classe abstraite qui représente le matériel d'un jeu d'échecs.
 *                 Les classes concrètent qui héritent de cette classe abstraites sont les 6 pièces
 *                 d'un jeu d'échecs traditionnel : King, Queen, Rook, Bishop, Knight, Pawn.
 *  ---------------------------------------------------------------------------
 **/

public abstract class Piece {

    // region parameter
    boolean hasMoved;
    PlayerColor color;
    PieceType type;
    Coord coord;
    // endregion

    // region ctor
    /**
     * Nom          : Piece
     * Description  : Permet de contruire une pièce concrète en spécifiant la coordonné, la couleur et
     son type.
     * @param coord : Coordonné ou la pièce va être crée sur un échiquier.
     * @param color : Couleur de la pièce (blanche ou noire)
     * @param type  : Indique le type de pièce à créer (Pawn, king, queen etc.)
     * @return      : L'objet Piece construit par le constructeur
     **/
    public Piece(Coord coord, PlayerColor color, PieceType type){
        this.coord = coord;
        this.color = color;
        this.type = type;
        hasMoved = false;
    }
    // endregion

    // region Method
    /**
     * Nom         : acceptedMove
     * Description : Indique si la pièce sélectionnée peut effectuer le mouvement que
     *               l'utilisateur a joué sur l'échiquier.
     * @param toX  : Coordonné en X ou l'on souhaite se déplacer sur l'échiquier
     * @param toY  : Coordonné en Y ou l'on souhaite se déplacer sur l'échiquier
     * @return     : Booléen indiquant si oui ou non le mouvement est valide.
     **/
    abstract boolean acceptedMove(int toX,int toY);

    /**
     * Nom         : move
     * Description : Permet de déplacer la pièce sélectionnée sur l'échiquier.
     * @param toX  : Coordonné en X ou l'on souhaite se déplacer sur l'échiquier
     * @param toY  : Coordonné en Y ou l'on souhaite se déplacer sur l'échiquier
     * @return     : / void
     **/
    public void move(int toX,int toY){
        coord.setCoord(toX, toY);
    }

    /**
     * Nom         : listMove
     * Description : Permet de générer tous les mouvements qu'une pièce peut effectuer.
     * @return     : List avec tous les vecteurs de points.
     **/
    abstract List<List<Coord>> listMove();

    /**
     * Nom         : listEatingMove
```

```
     * Description : Permet de générer tous les mouvements qu'une pièce peut effectuer
     *               pour manger une pièce.
     * @return     : List avec tous les vecteurs de points.
     **/
    abstract List<List<Coord>> listEatingMove();


    /**
     * Nom         : getType
     * Description : Permet d'obtenir le type d'une pièce.
     * @return     : Le type de la pièce.
     **/
    public PieceType getType(){
        return type;
    }

    /**
     * Nom         : getColor
     * Description : Permet d'obtenir la couleur d'une pièce.
     * @return     : La couleur de la pièce.
     **/
    public PlayerColor getColor() {
        return color;
    }
    // endregion
}
```

```java
package engine;

import chess.PieceType;
import chess.PlayerColor;
import engine.util.Coord;
import engine.util.MoveDiag;
import engine.util.MoveLin;

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

/**
 *  ----------------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Cette classe représente la pièce "King" dans un jeu d'échecs.
 *  ----------------------------------------------------------------------------------
 **/
public class King extends Piece{

    // coordonnées du petit rook et grand rook
    private Coord lCastle, bCastle;
    public King(Coord coord, PlayerColor color) {
        super(coord, color, PieceType.KING);
        if(color == PlayerColor.WHITE){
            lCastle = new Coord(6,0);
            bCastle = new Coord(2,0);
        } else{
            lCastle = new Coord(6,7);
            bCastle = new Coord(2,7);
        }
        md = new MoveDiag(1);
        ml = new MoveLin(1);
    }

    // region parametre
    MoveDiag md;
    MoveLin ml;
    //endregion

    /***
     * @author Oscar Baume
     * @brief La fonction retourne si le mouvement peut être fait.
     * @param toX : coordonnée X de la destination du mouvement
     * @param toY : coordonnée Y de la destination du mouvement
     * @return boolean qui indique si le mouvement peut être fait
     */
    @Override
    boolean acceptedMove(int toX, int toY) {
        // on fait les delta pour x et y
        int deltaX = coord.getX() - toX;
        int deltaY = coord.getY() - toY;
        // si on se déplace pas le mouvement n'est pas accepté
        if(deltaY == deltaX && deltaY == 0) return false;

        // le roi pouvant se deplacer d'une case dans les 8 directions
        // si le delta^2 est égale à 0 ou à 1 alors on peut se deplacer là (1ère ligne du return)
        // si le roi n'a pas bougé et qu'il veut rooker alors il peut se deplacer là  (2e et 3e ligne du
        return)
        return (deltaX * deltaX <= 1 && deltaY * deltaY <= 1) ||
               (!hasMoved && ((color == PlayerColor.WHITE) && toY == 0 && (toX == 2 || toX == 6) ||
               ((color == PlayerColor.BLACK) && toY == 7 && (toX == 2 || toX == 6))));
    }

    /***
     * @author Oscar Baume
     * @brief fonction qui retourne les mouvements possible de la piece
     * @return une liste de liste de coordonée qui correspont au "vecteur" de deplacement de la pièce
     */
    @Override
    List<List<Coord>> listMove() {
        // le roi peut se déplacer dans les 8 directions d'une case
        // on construit alors une liste composées de 8 listes avec chacune d'elle la coordonnées du
        mouvement
        List<List<Coord>> vectors = new LinkedList<>();
```

```
        List<Coord> v = new LinkedList<>();
        for(int i = -1; i < 2; ++i){
            for(int j = -1; j < 2; ++j){
                if(!(i==0&&j==0)){
                    try{
                        v = new ArrayList<Coord>();
                        v.add(new Coord(coord.getX() + i, coord.getY() +j));
                        vectors.add(v);
                    }catch (RuntimeException e){System.out.println(e.getMessage());}
                }
            }
        }
        // si le roi n'a pas bougé alors il peut rooker
        if(!hasMoved){
            v = new ArrayList<>();
            v.add(lCastle);
            vectors.add(v);
            v = new ArrayList<>();
            v.add(bCastle);
            vectors.add(v);
        }
        return vectors;
    }

    /***
     * @author Oscar Baume
     * @brief fonction qui retourne les coordonnées ou le roi peut manger une pièce adverse.
     * @return les coordonées au le roi peut manger une pièce adverse.
     */
    @Override
    List<List<Coord>> listEatingMove() {
        // le roi peut se déplacer dans les 8 directions d'une case
        // on construit alors une liste composées de 8 listes avec chacune d'elle la coordonnées du
        mouvement
        List<List<Coord>> vectors = new ArrayList<>();
        List<Coord> v = new ArrayList<>();
        for(int i = -1; i < 2; ++i){
            for(int j = -1; j < 2; ++j){
                if(!(i==0&&j==0)){
                    try{
                        v = new ArrayList<Coord>();
                        v.add(new Coord(coord.getX() + i, coord.getY() +j));
                        vectors.add(v);
                    }catch (RuntimeException e){System.out.println(e.getMessage());}
                }
            }
        }
        return vectors;
    }
}
```

```java
package engine;

import chess.PieceType;
import chess.PlayerColor;
import engine.util.Coord;
import engine.util.MoveDiag;
import engine.util.MoveLin;
import java.util.LinkedList;
import java.util.List;

/**
 *  ------------------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Cette classe représente la pièce "Queen" dans un jeu d'échecs.
 *  ------------------------------------------------------------------------------------
 **/

public class Queen extends Piece {

    // region Parameters
    MoveDiag md;
    MoveLin ml;
    // endregion

    // region Constructor
    /**
     * Nom          : Queen
     * Description  : Permet de construire une pièce de type Queen
     * @param coord : Coordonnée à laquelle il faut créer la pièce
     * @param color : Couleur de la pièce.
     * @return      : L'objet Queen construit par le constructeur
     **/
    public Queen(Coord coord, PlayerColor color) {
        super(coord, color, PieceType.QUEEN);
        md = new MoveDiag(8);
        ml = new MoveLin(8);
    }
    // endregion

    // region Methods
    @Override
    boolean acceptedMove(int toX, int toY) {
        int deltaX = Math.abs(toX - coord.getX());
        int deltaY = Math.abs(toY - coord.getY());

        return (deltaX != 0 && deltaY == 0 || deltaX == 0 && deltaY != 0) || (deltaX == deltaY);
    }

    @Override
    List<List<Coord>> listMove() {
        List<List<Coord>> list = md.listMove(coord);
        list.addAll(ml.listMove(coord));
        return list;
    }


    @Override
    List<List<Coord>> listEatingMove() {
        return listMove();
    }
    // endregion
}
```

```java
package engine;

import chess.PieceType;
import chess.PlayerColor;
import engine.util.Coord;
import engine.util.MoveLin;
import java.util.LinkedList;
import java.util.List;

/**
 *  ------------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Cette classe représente la pièce "Rook" dans un jeu d'échecs.
 *  ------------------------------------------------------------------------------
 **/

public class Rook extends Piece {

    // region Parameters
    MoveLin ml;
    // endregion

    // region Constructor
    /**
     * Nom          : Rook
     * Description  : Permet de construire une pièce de type Rook
     * @param coord : Coordonnée à laquelle il faut créer la pièce
     * @param color : Couleur de la pièce.
     * @return      : L'objet Rook construit par le constructeur
     **/
    public Rook(Coord coord, PlayerColor color) {
        super(coord, color, PieceType.ROOK);
        ml = new MoveLin(8);
    }
    // endregion

    // region Methods
    @Override
    boolean acceptedMove(int toX, int toY) {
        int deltaX = Math.abs(toX - coord.getX());
        int deltaY = Math.abs(toY - coord.getY());

        return deltaX != 0 && deltaY == 0 || deltaX == 0 && deltaY != 0;
    }

    @Override
    List<List<Coord>> listMove() {
        return ml.listMove(coord);
    }

    @Override
    List<List<Coord>> listEatingMove() {
        return listMove();
    }
    // endregion
}
```

```java
package engine;

import chess.PieceType;
import chess.PlayerColor;
import engine.util.Coord;
import engine.util.MoveDiag;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.lang.Math;

/**
 *  ------------------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Cette classe représente la pièce "Bishop" dans un jeu d'échecs.
 *  ------------------------------------------------------------------------------------
 **/
public class Bishop extends Piece{

    // region Contructor
    public Bishop(Coord coord, PlayerColor color) {
        super(coord, color, PieceType.BISHOP);
        md = new MoveDiag(8);
    }
    // endregion

    // region Parametre
    private MoveDiag md;
    // endregion


    // region Methods
    @Override
    boolean acceptedMove(int toX, int toY) {
        int deltaX = coord.getX() - toX;
        int deltaY = coord.getY() - toY;
        if(deltaY == 0 || deltaX == 0) return false;

        return Math.abs(deltaX) == Math.abs(deltaY);
    }

    @Override
    List<List<Coord>> listMove() {
    return md.listMove(coord);
    }

    @Override
    List<List<Coord>> listEatingMove() {
        return md.listEatingMove(coord);
    }
    // endregion
}
```

```java
package engine;

import chess.PieceType;
import chess.PlayerColor;
import engine.util.Coord;
import engine.util.MoveKnight;
import java.util.List;

/**
 *  ---------------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Cette classe représente la pièce "Knight" dans un jeu d'échecs.
 *  ---------------------------------------------------------------------------------
 **/

public class Knight extends Piece{

    // region Parameter
    MoveKnight mk;
    // endregion

    // region Constructor
    /**
     * Nom          : Knight
     * Description  : Permet de construire une pièce de type Knight
     * @param coord : Coordonnée à laquelle il faut créer la pièce
     * @param color : Couleur de la pièce.
     * @return      : L'objet Knight construit par le constructeur
     **/
    public Knight(Coord coord, PlayerColor color) {
        super(coord, color, PieceType.KNIGHT);
        mk = new MoveKnight(2);
    }
    // endregion

    // region Methods
    @Override
    boolean acceptedMove(int toX, int toY) {
        int deltaX = Math.abs(toX - coord.getX());
        int deltaY = Math.abs(toY - coord.getY());

        return deltaX == 2 && deltaY == 1  || deltaX == 1 && deltaY == 2;
    }

    @Override
    List<List<Coord>> listMove() {
        return mk.listMove(coord);
    }

    @Override
    List<List<Coord>> listEatingMove() {
        return mk.listEatingMove(coord);
    }
    // endregion
}
```

```java
package engine;

import chess.PieceType;
import chess.PlayerColor;
import engine.util.Coord;
import engine.util.MoveDiag;
import engine.util.MoveLin;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
/**
 *  ------------------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Cette classe représente la pièce "Pawn" dans un jeu d'échecs.
 *  ------------------------------------------------------------------------------------
 **/
public class Pawn extends Piece{
    // region Constructor
    public Pawn(Coord coord, PlayerColor color) {
        super(coord, color, PieceType.PAWN);
        if(color == PlayerColor.WHITE) facing = 1;
        else facing = -1;
        md = new MoveDiag(1);
        ml = new MoveLin(2);
    }
    // endregion


    // region Parameter
    MoveLin ml;
    MoveDiag md;
    private int facing;
    // endregion


    // region Methods
     @Override
     public void move(int toX,int toY){
         super.move(toX,toY);
         ml = new MoveLin(1);
     }

    @Override
    boolean acceptedMove(int toX, int toY) {
        int deltaX = coord.getX() - toX;
        int deltaY = coord.getY() - toY;
        return (!hasMoved && deltaY == -2*facing && deltaX == 0) || (deltaY == -facing && deltaX >= -1 &&
         deltaX <= 1);
    }

    @Override
    List<List<Coord>> listMove() {
        List<List<Coord>> vect = ml.listMove(coord);
        List<List<Coord>> out = new LinkedList<>();
        for(List<Coord> moves : vect){
            List<Coord> v = new LinkedList<>();
            for(Coord c : moves){
                if(acceptedMove(c.getX(),c.getY())){
                    v.add(c);
                }
                out.add(v);
            }
        }
        return out;
    }

    @Override
    List<List<Coord>> listEatingMove() {
        List<List<Coord>> vect = md.listMove(coord);
        List<List<Coord>> out = new LinkedList<>();
        for(List<Coord> moves : vect){
            List<Coord> v = new LinkedList<>();
            for(Coord c : moves){
                if(acceptedMove(c.getX(),c.getY())){
```

```
                    v.add(c);
                }
                out.add(v);
            }
        }
        return out;
    }
    // endregion
}
```

```java
package engine.util;
import java.util.List;

/**
 *  ---------------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Il s'agit d'une classe abstraite qui représente des mouvements.
 *                 Un mouvement est caractérisé par des vecteurs de coordonnées.
 *                 Les vecteurs partent d'un point central. Le paramètre range correspond
 *                 au nombre de coordonnées parcouru depuis le point central dans une direction.
 *  ---------------------------------------------------------------------------------
 **/

public abstract class Movement {

    // region Parmètre
    int range;
    // endregion

    // region Constructor
    /**
     * Nom          : Movement
     * Description  : Permet de contruire un Movement en spécifiant la range de déplacement autorisée.
     * @param range : Range de déplacement autorisée.
     * @return      : L'objet Movement construit par le constructeur
     **/
    public Movement(int range) {
        this.range = range;
    }
    // endregion

    // region Methods
    /**
     * Nom          : listMove
     * Description  : Gènère des listes de mouvements à partir du point coord
     * @param coord : Point d'origine à partir d'ou les vecteurs de mouvements sont
     *                générés.
     * @return      : Liste contenant des vecteurs de mouvements
     **/
    public abstract List<List<Coord>> listMove(Coord coord);

    /**
     * Nom          : listEatingMove
     * Description  : Gènère des listes de mouvements à partir du point coord. A la différence
     *                de "listMove", on gènère les mouvements ou l'on peut manger une pièce.
     * @param coord : Point d'origine à partir d'ou les vecteurs de mouvements sont
     *                générés.
     * @return      : Liste contenant des vecteurs de mouvements
     **/
    public abstract List<List<Coord>> listEatingMove(Coord coord);
    // endregion
}
```

```java
package engine.util;

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

/**
 *  ------------------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Cette classe permet de générer quatre vecteurs de points à partir d'un
 *                 point central. Les vecteurs sont disposés en croix diagonale, autour du point central
 *                 (croix diagonale x).
 *  ------------------------------------------------------------------------------------
 **/

public class MoveDiag extends Movement {

    // region Parameter
    private final int NBR_VECTORS = 4;
    // endregion

    // region Contructor
    /**
     * Nom          : MoveDiag
     * Description  : Permet de contruire un MoveDiag en spécifiant la range de déplacement autorisée.
     * @param range : Range de déplacement autorisée.
     * @return      : L'objet MoveDiag construit par le constructeur
     **/
    public MoveDiag(int range) {
        super(range);
    }
    // endregion

    // region Methods
    @Override
    public List<List<Coord>> listMove(Coord coord) {
        List<List<Coord>> vectors = new LinkedList<>();
        List<Coord> v = new LinkedList<>();
        int coefX = 1,coefY=1;
        for(int i = 0; i < NBR_VECTORS; ++i){
            v = new ArrayList<>();
            for(int j = 1; j < this.range + 1; ++j){
                try{
                    v.add(new Coord(coord.getX() + j*coefX,coord.getY() + j*coefY));
                }catch (RuntimeException e){
                    System.out.println(e.getMessage());
                    break;
                }
            }
            vectors.add(v);

            if(i == 0 || i == 2)coefY = -1;
            if(i == 1){coefX = -1;coefY=1;}
        }
        return vectors;
    }

    @Override
    public List<List<Coord>> listEatingMove(Coord coord) {
        return listMove(coord);
    }
    // endregion
}
```

```java
package engine.util;

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

/**
 *  ------------------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Cette classe permet de générer quatre vecteurs de points à partir d'un
 *                 point central. Les vecteurs sont disposés en croix autour du point central
 *                 (croix droite +).
 *  ------------------------------------------------------------------------------------
 **/

public class MoveLin extends Movement{

    // region Parameter
    private final int NBR_VECTORS = 4;
    // endregion

    // region Constructor
    /**
     * Nom          : MoveLin
     * Description  : Permet de contruire un MoveLin en spécifiant la range de déplacement autorisée.
     * @param range : Range de déplacement autorisée.
     * @return      : L'objet MoveLin construit par le constructeur
     **/
    public MoveLin(int range) {
        super(range);
    }
    // endregion

    // region Methods
    @Override
    public List<List<Coord>> listMove(Coord coord) {
        List<List<Coord>> vectors = new LinkedList<>();
        List<Coord> v = new LinkedList<>();
        int coefX = 1,coefY=0;
        for(int i = 0; i < NBR_VECTORS; ++i){
            v = new ArrayList<>();
            for(int j = 1; j < this.range + 1; ++j){
                try{
                    v.add(new Coord(coord.getX() + j*coefX,coord.getY() + j*coefY));
                }catch (RuntimeException e){
                    System.out.println(e.getMessage());
                    break;
                }
            }
            vectors.add(v);
            if(i == 0 ) {
                coefX = 0;
                coefY = 1;
            }
            if(i == 1 ) {
                coefX = -1;
                coefY = 0;
            }
            if(i == 2 ) {
                coefX = 0;
                coefY = -1;
            }
        }
        return vectors;
    }

    @Override
    public List<List<Coord>> listEatingMove(Coord coord) {
        return listMove(coord);
    }
    // endregion
}
```

```java
package engine.util;

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

/**
 *  ----------------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Cette classe permet de générer les mouvements de la pièce cavalier dans
 *                 un jeu d'échecs.
 *  ----------------------------------------------------------------------------------
 **/

public class MoveKnight extends Movement {

    // region Parameter
    private final int NBR_VECTORS = 8;
    // endregion

    // region Contructor
    /**
     * Nom          : MoveKnight
     * Description  : Permet de contruire un MoveKnight en spécifiant la range de déplacement autorisée.
     * @param range : Range de déplacement autorisée.
     * @return      : L'objet MoveKnight construit par le constructeur
     **/
    public MoveKnight(int range) {
        super(range);
    }
    // endregion

    // region Methods
    @Override
    public List<List<Coord>> listMove(Coord coord) {
        List<List<Coord>> vectors = new LinkedList<>();

        int coefX = 1, coefY = 2;
        for (int i = 0; i < NBR_VECTORS; ++i) {
            List<Coord> v = new LinkedList<>();

            try {
                v.add(new Coord(coord.getX() + coefX, coord.getY() + coefY));
                vectors.add(v);
            } catch (RuntimeException e) {
                System.out.println(e.getMessage());
            }


            if (i == 0) {
                coefX = 2;
                coefY = 1;
            }
            if (i == 1) {
                coefX = 2;
                coefY = -1;
            }
            if (i == 2) {
                coefX = 1;
                coefY = -2;
            }
            if (i == 3) {
                coefX = -1;
                coefY = -2;
            }
            if (i == 4) {
                coefX = -2;
                coefY = -1;
            }
            if (i == 5) {
                coefX = -2;
                coefY = 1;
            }
            if (i == 6) {
                coefX = -1;
                coefY = 2;
```

```
            }
        }
        return vectors;
    }

    @Override
    public List<List<Coord>> listEatingMove(Coord coord) {
        return listMove(coord);
    }
    // endregion
}
```

```java
package engine.util;

/**
 *  ------------------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Il s'agit d'une classe qui modélise un position sur l'échiquier
 *  ------------------------------------------------------------------------------------
 **/
public class Coord {
    private int x;
    private int y;
    private final int min = 0, max = 7;

    /**
     * Nom          : Coord
     * Description  : Constructeur de coordonnées
     * @param x     : Position x de la coord
     * @param y     : Position y de la coord
     * @throws RuntimeException
     */
    public Coord(int x,int y) throws RuntimeException{
        if(x < min || x > max || y < min || y > max){
            throw new RuntimeException("La coordonnée [" + x + ","+ y +"] n'est pas inclus dans
            l'echiquier");
        }
        this.x = x;
        this.y = y;
    }

    /**
     * Nom          : setCoord
     * Description  : Setter de coordonnées
     * @param x     : Nouvelle position x
     * @param y     : Nouvelle position y
     * @throws RuntimeException
     */
    public void setCoord(int x,int y) throws RuntimeException{
        if(x < min || x > max || y < min || y > max){
            throw new RuntimeException("Coordonnées [" + x + ","+ y +"] pas inclus dans l'echiquier");
        }
        this.x = x;
        this.y = y;
    }

    /**
     * Nom          : getX
     * Description  : getter de la positon x
     * @return      : valeur de x
     */
    public int getX() {
        return x;
    }
    /**
     * Nom          : getY
     * Description  : getter de la positon y
     * @return      : valeur de y
     */
    public int getY() {
        return y;
    }

    /**
     * Nom          : isEqual
     * Description  : Méthode qui retourne si une coordonnées est égale à la coord actuelle
     * @param other : coordonnées avec laquelle on se compare
     * @return      : si c'est égale
     */
    public boolean isEqual(Coord other){
        return x == other.x && y == other.y;
    }

    /**
     * Nom          : toString
     * Description  : Méthode qui retourne la coordonnées au format string
     * @return      : String de la coordonneés
```

```
     */
    public String toString(){
        return "("+x+","+y+")";
    }
}
```

```java
package engine.util;
import chess.ChessView;
import engine.Piece;

/**
 *  ----------------------------------------------------------------------------------
 * @Authors      : Slimani Walid & Baume Oscar
 * @Date         : 07.01.2023
 *
 * @Description  : Classe permettant de créer une pièce lorsqu'un pion doit être promu.
 *  ----------------------------------------------------------------------------------
 **/

abstract public class PromotionChoice implements ChessView.UserChoice {
    // region Methods
    /**
     * Nom          : create
     * Description  : Permet de créer la pièce choisi par l'utilisateur lors d'une promotion.
     * @return      : La pièce choisie par l'utilisateur.
     **/
    public abstract Piece create();
    // endregion
}
```