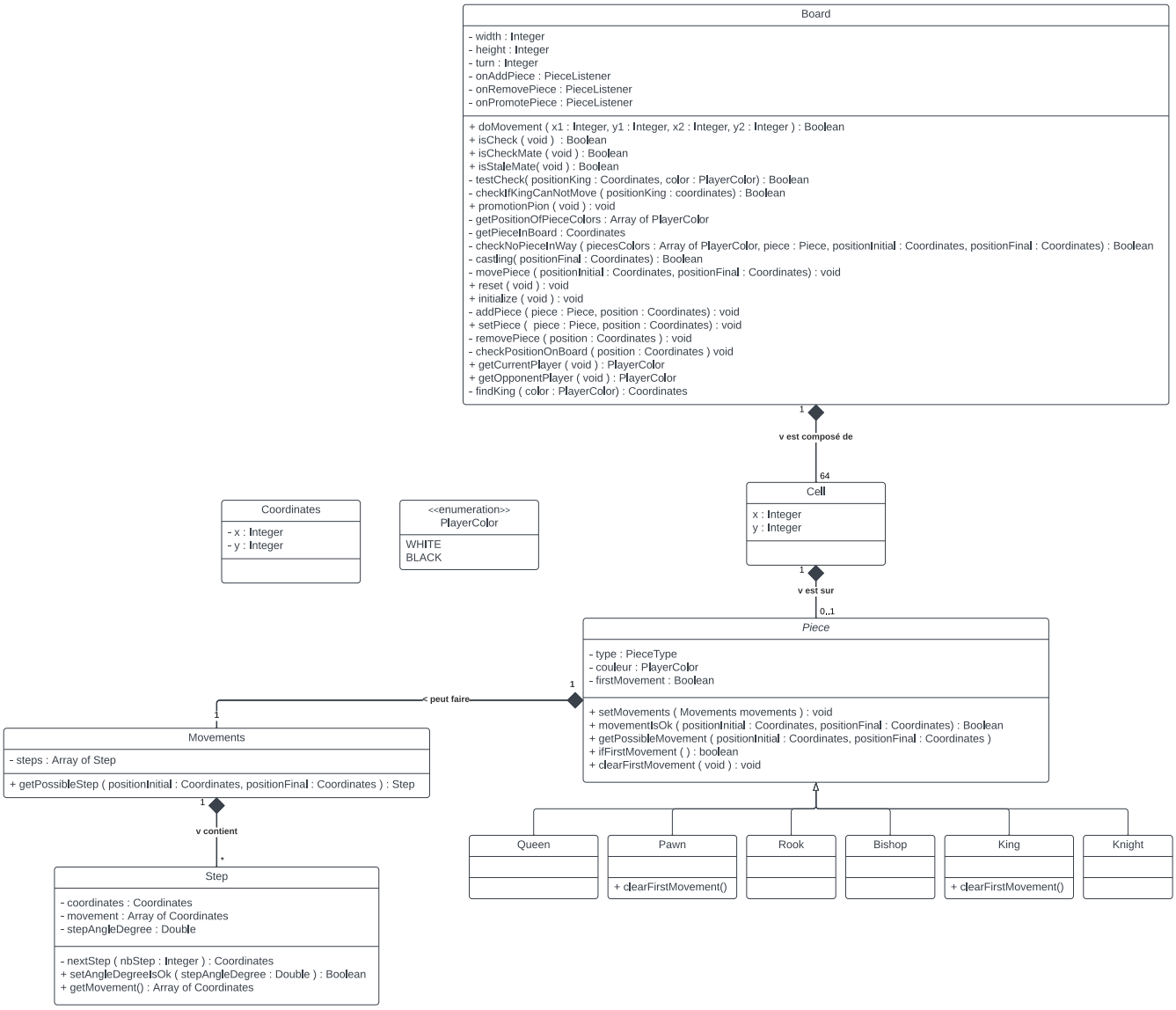
Laboratoire 8 - Jeu d'échecs

Groupe : L08GrK

Étudiants : Rodrigo Lopes dos Santos, Urs Behrmann

Modélisation UML



Choix de conception

GameManager

Le GameManager est la classe qui gère le jeu. Il contient les méthodes pour faire le liens entre l'UI et l'engine. Il hérite de la classe 'ChessController' qui est une classe abstraite qui contient les méthodes pour gérer le jeu. Elle contient la méthode 'initListenres' qui initialise les listeners qui transforme l'UI dans l'engine, lorsque l'utilisateur veut bouger une pièce sur la plateau à travers la fonction 'move'. On a aussi les fonctions 'newGame' qui reset le plateau et l'UI.

La 'updateMessage' est une fonction qui met à jour le message de l'UI. Elle est appelée à chaque fois qu'un mouvement est fait. Elle écrit qui qui doit jouer, si il y a un "check", "checkmate" ou "stalemate".

Plateau ('Board')

Le plateau est une classe qui contient les pièces et les méthodes pour les manipuler. Elle contient un Array de cellule qui représente le plateau affiché sur l'UI.

Elle contient les fonction qui servent à bouger les pièces, à les supprimer et à les ajouter. Elle contient aussi les fonctions qui servent à vérifier si un mouvement est légal ou pas, si le roi est en "check", "checkmate" ou "stalemate".

Elle est aussi responsable de déterminer quel joueur doit jouer et aussi le nombre de tour jouer.

Mouvement

Fichiers et Leur Rôle

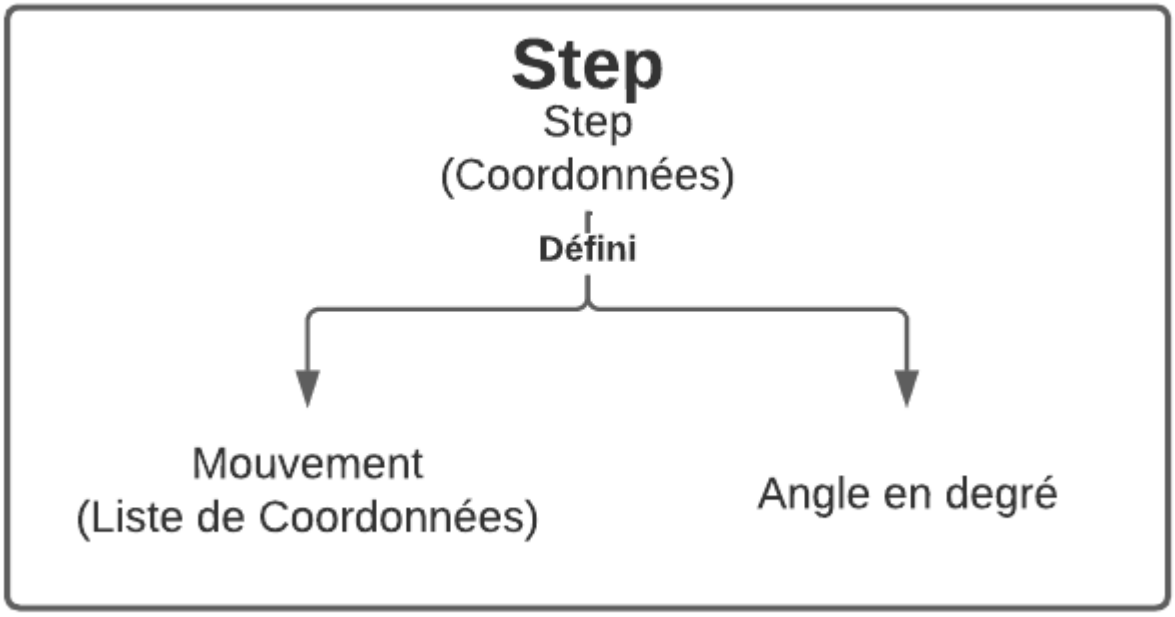
`Step.java`

- Définit un pas de mouvement.
- Selon le maxStep, il définie un mouvement via des les coordonnées :
  - [Pas de mouvement, Pas de mouvement + 1 pas, Pas de mouvement + 2 pas, ..., Pas de mouvement + maxStep]
- Contient les coordonnées d'un pas, une liste de coordonnées du mouvement selon maxStep et l'angle de déplacement.

- Il est possible d'inverser l'axe x et y. Cela est intéressant pour les pièces noir car, par exemple, avancer en avant est un step de (0, -1) et non pas de (0, 1)

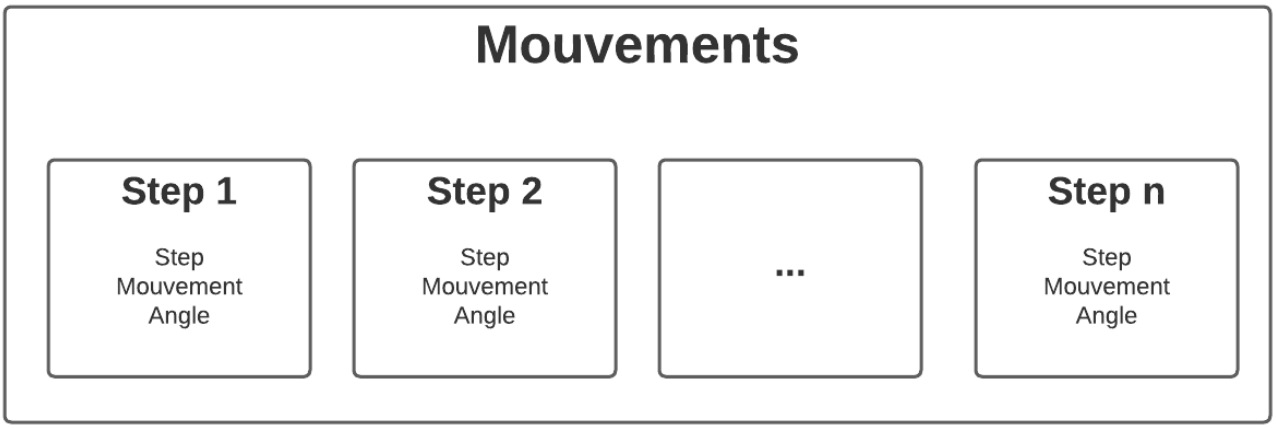
```
public Step(Coordinates step, int maxStep, boolean invertOrdinateAxis, boolean invertAbsc

public class X extends Piece {
    public X(PlayerColor color) {
        super(PieceType.X, color, new Movements(new Step[]{
                new Step(new Coordinates(1,0), 1, color == PlayerColor.BLACK, color == Pl
                // new Step(....), ...
                }));
    }
}
```



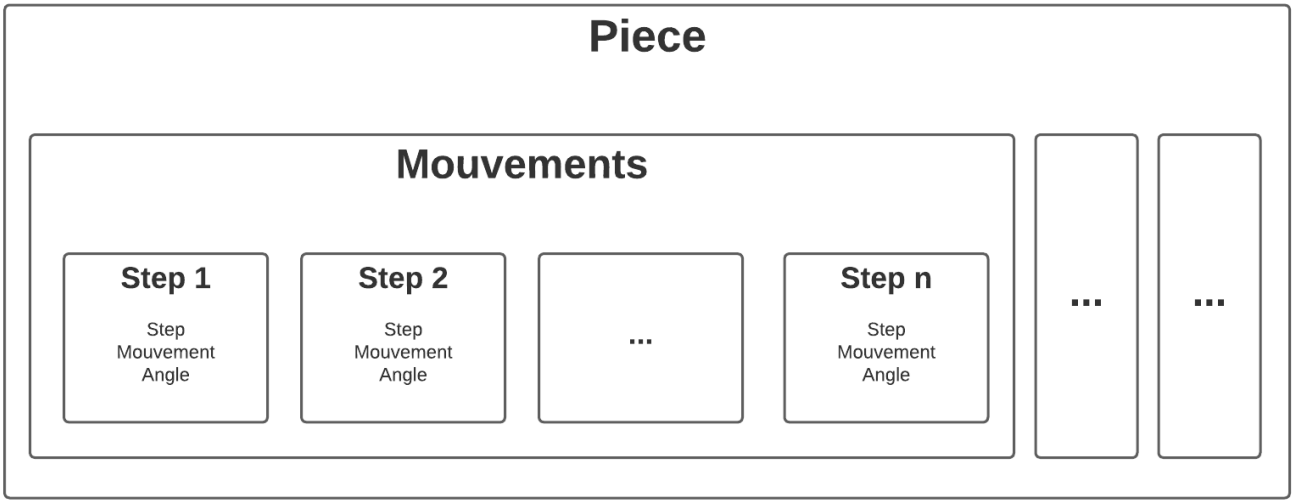
`Movements.java`

- Gère les mouvements des pièces.
- Contient une liste de `Step` et une méthode pour obtenir `Step` possible en fonction des positions initiale et finale.



`Piece.java`

- Contient une méthode `Coordinates[] getPossibleMovement( ... )` qui via les `movements` et les positions initiale et finale détermine le step possible et retourne son mouvement.
- Contient une méthode `boolean movementIsOk( ... )` qui via les positions initiale, finale et `Coordinates[] getPossibleMovement( ... )` détermine le mouvement possible et cherche dans celui-ci si la position final si trouve.
  - Si oui : Le mouvement de la position initiale à finale est possible
  - Sinon : Le mouvement de la position initiale à finale est impossible



Fonctionnement Général

Les mouvements des pièces sont gérés de manière modulaire, où chaque pièce a ses propres règles de mouvement définies dans des classes spécifiques (par exemple, `Knight.java` , `Pawn.java`). Ces classes utilisent la classe `Movements` pour gérer les déplacements possibles.

La classe `Step` joue un rôle clé en définissant la nature d'un pas de mouvement, tandis que `Coordinates` est utilisée pour représenter et manipuler les positions sur le plateau.

Exemple d'utilisation:

```
public class King extends Piece {
    public King(PlayerColor color) {
        super(PieceType.KING, color, new Movements(new Step[]{
                //Horizontal
                new Step(new Coordinates(1,0), 1, color == PlayerColor.BLACK, color == Pl
                new Step(new Coordinates(-1,0), 1, color == PlayerColor.BLACK, color == P

                //Vertical
                new Step(new Coordinates(0,1), 1, color == PlayerColor.BLACK, color == Pl
                new Step(new Coordinates(0,-1), 1, color == PlayerColor.BLACK, color == P

                //Diagonal
                new Step(new Coordinates(1,1), 1, color == PlayerColor.BLACK, color == Pl
                new Step(new Coordinates(1,-1), 1, color == PlayerColor.BLACK, color == P
                new Step(new Coordinates(-1,1), 1, color == PlayerColor.BLACK, color == P
                new Step(new Coordinates(-1,-1), 1, color == PlayerColor.BLACK, color ==
        }));
    }
}
```

Tests

On a crée un GameManagerTest et un BoardTest qui peuvent lancer plusieurs scènarios de test différent :

- test pawn
- test rook
- test knight
- test queen
- test king
- test bishop
- test pawn promotion
- test check

- test checkmate
- test stalemate
- test castling

test pawn

On a que deux pions sur le plateau pour tester le mouvement des pions et leur attack "en passant" et en diagonal.

test rook

On a que deux tours sur le plateau pour tester le mouvement des tours et leur attack.

On peut aussi tester le mouvement des tours quand il y a des pièces entre la tour et la position finale.

test knight

On a que deux cavaliers sur le plateau pour tester le mouvement des cavaliers et leur attack.

test queen

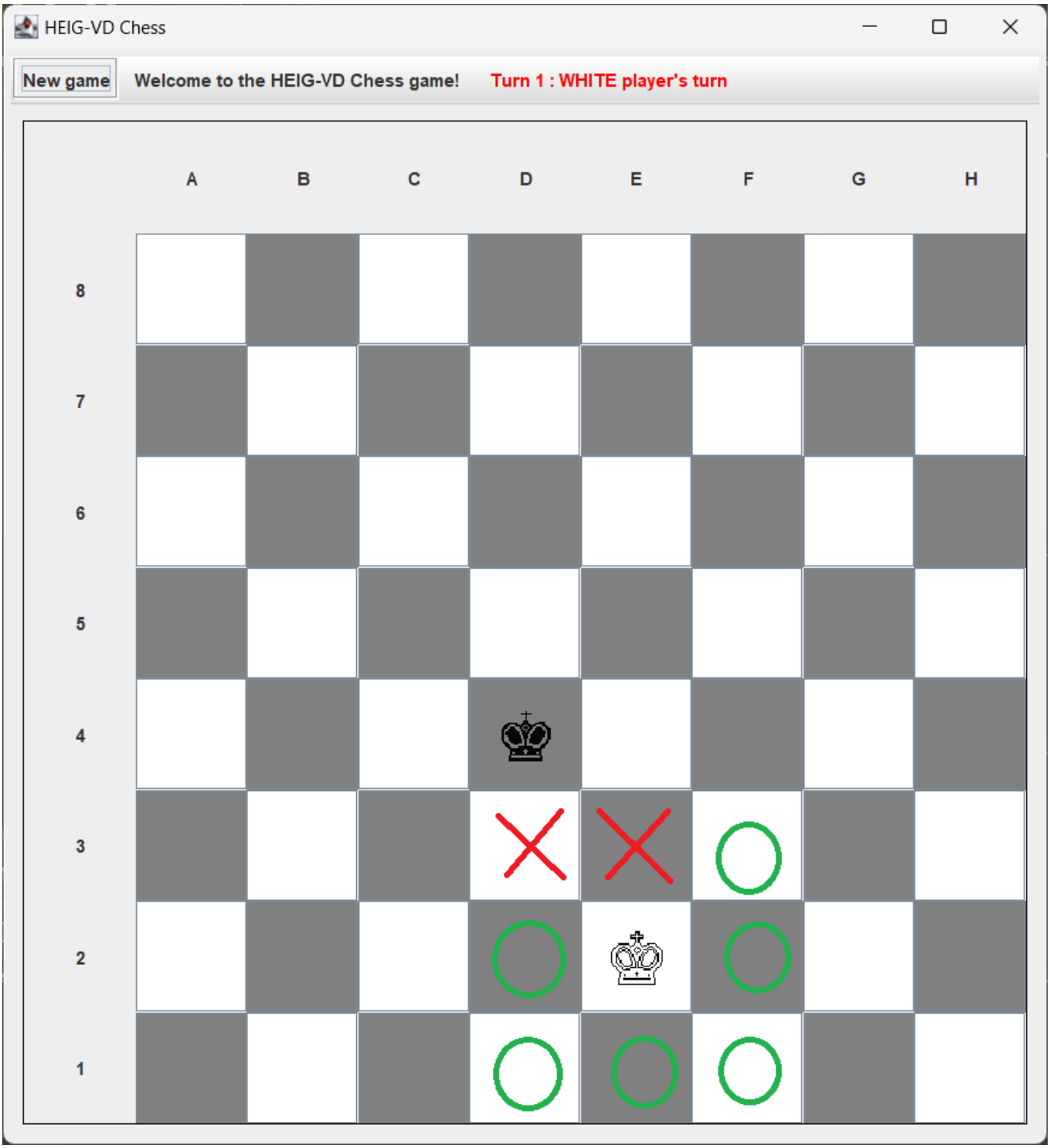
On a que deux reines sur le plateau pour tester le mouvement des reines et leur attack.

On peut aussi tester le mouvement des reines quand il y a des pièces entre la reine et la position finale.

test king

On a que deux rois sur le plateau pour tester le mouvement des rois et leur attack.

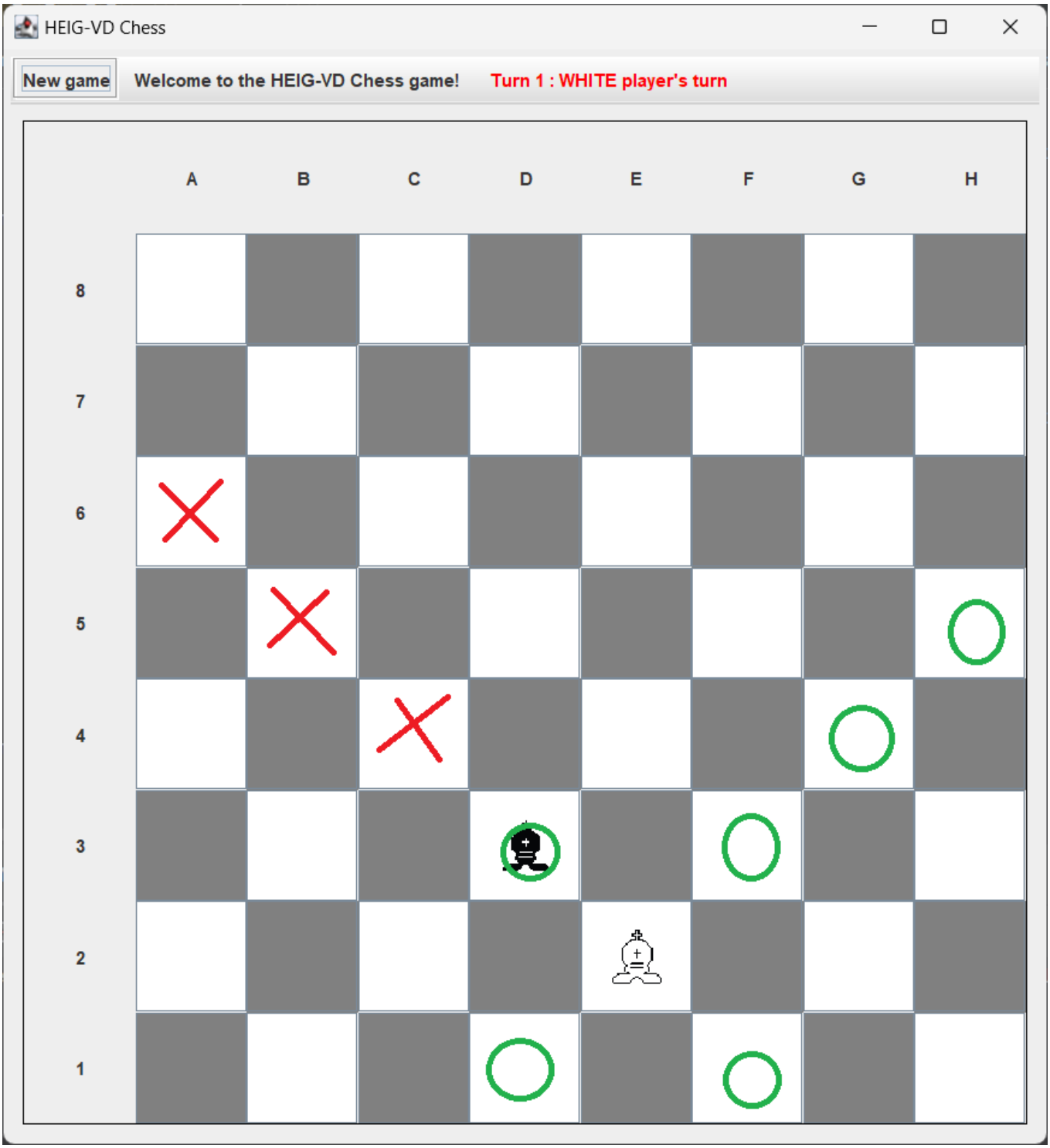
On peut aussi tester le mouvement des rois quand il y a une pièce qui peut attaquer le roi et est donc un mouvement illégal.



test bishop

On a que deux fous sur le plateau pour tester le mouvement des fous et leur attack.

On peut aussi tester le mouvement des fous quand il y a des pièces entre le fou et la position finale.

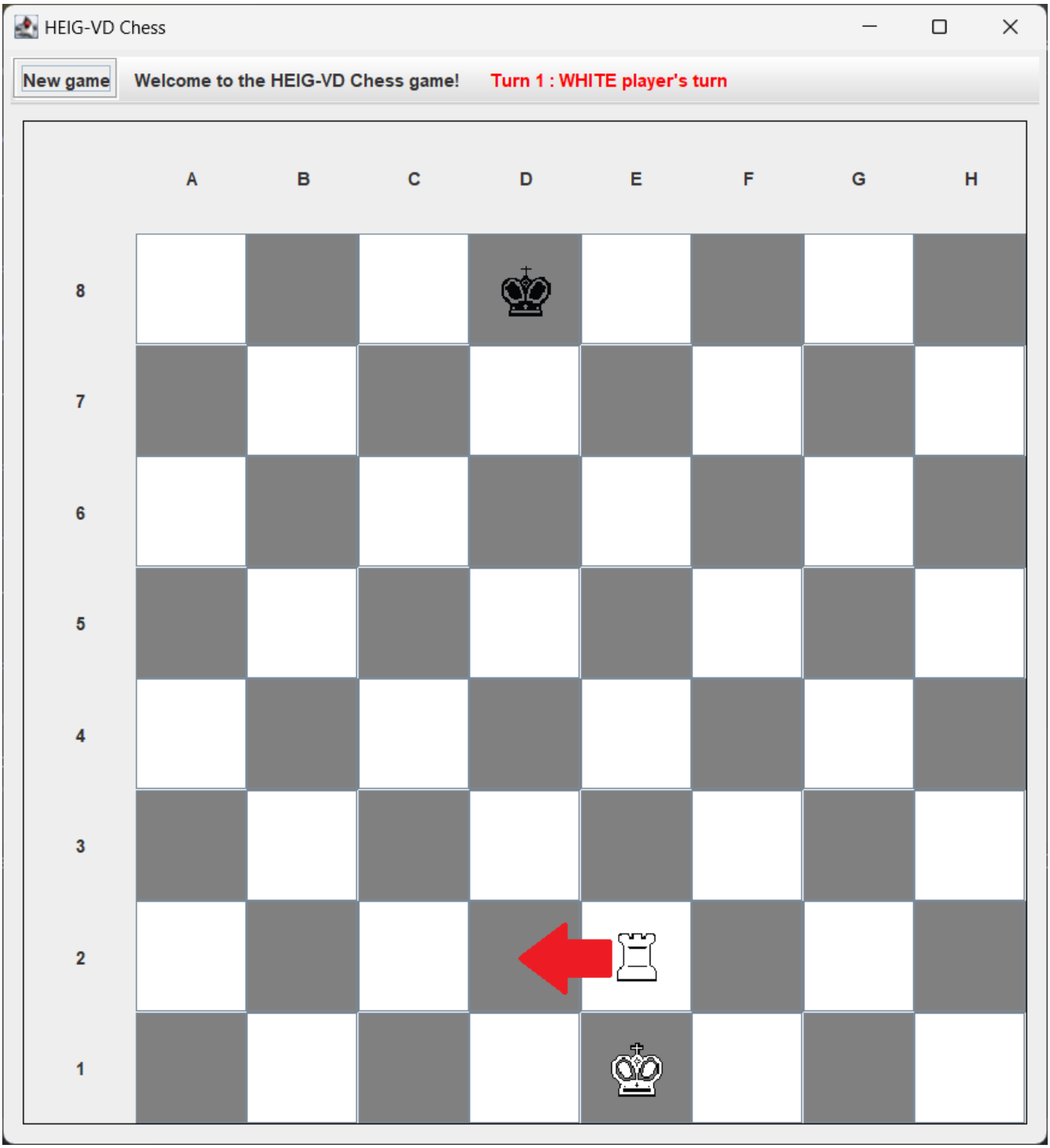


test pawn promotion

On a que deux pions sur le plateau pour tester la promotion des pions.

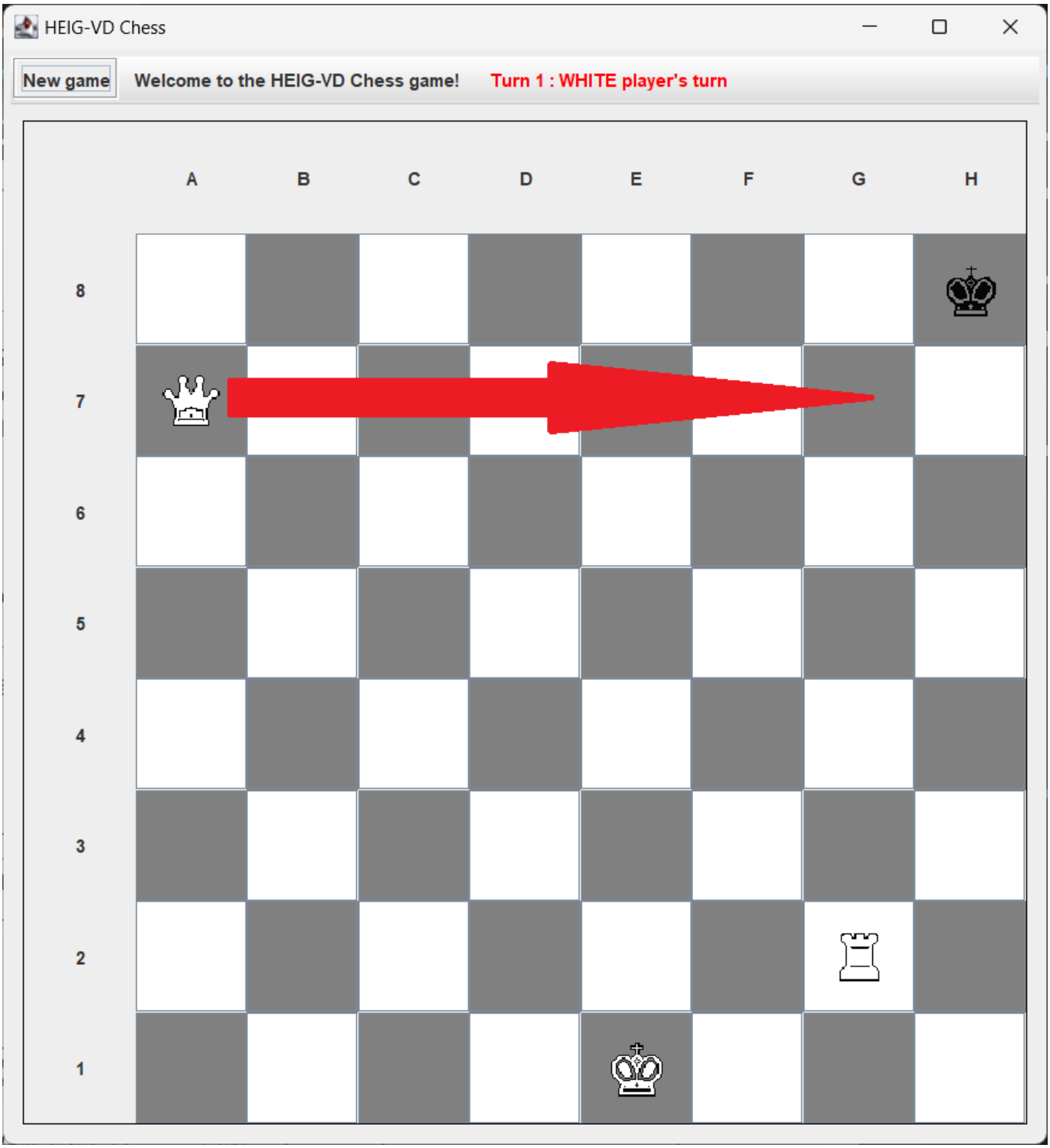
test check

On a quelque pièces sur le plateau pour tester le "check" pour noir.



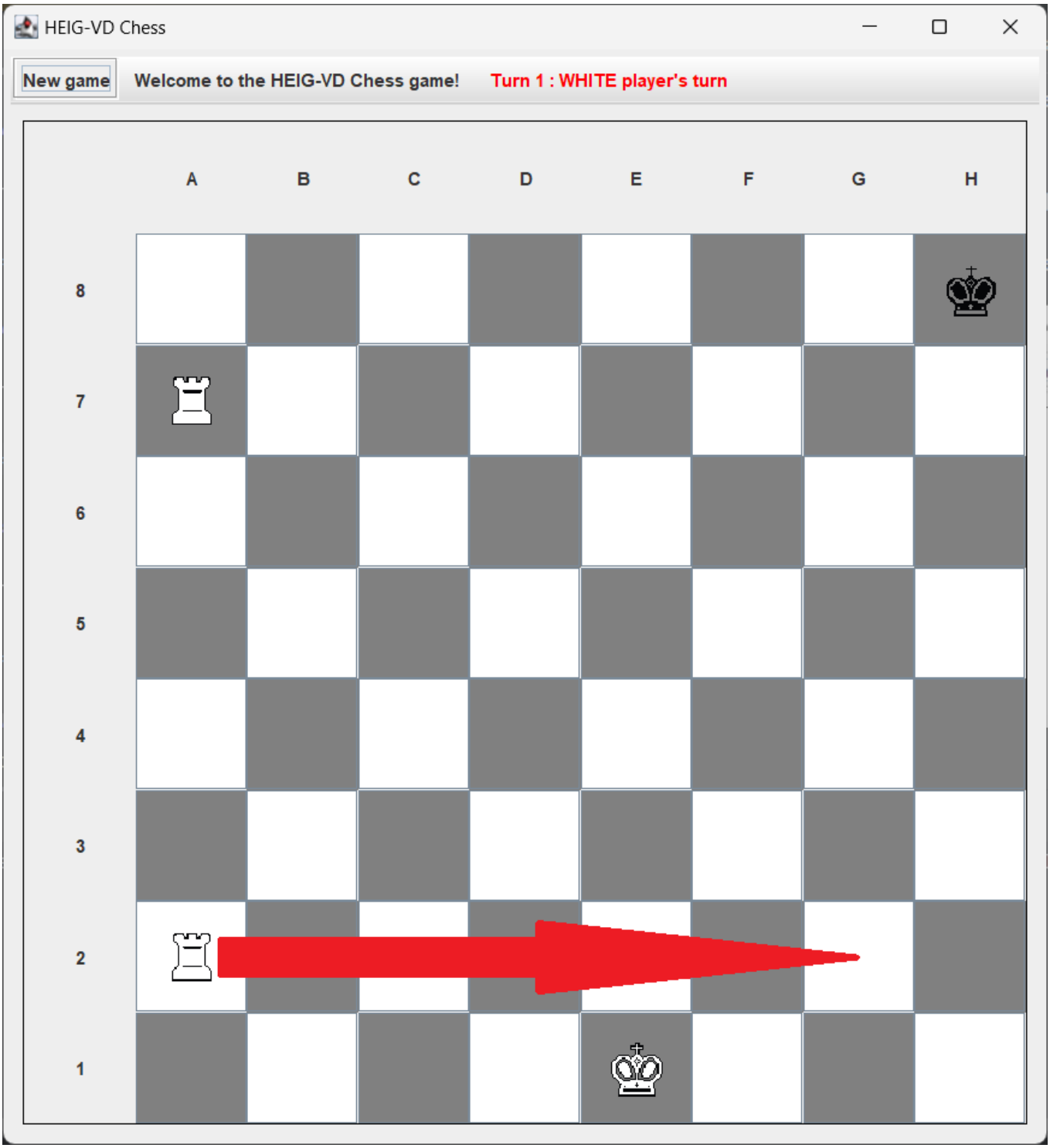
test checkmate

On a quelque pièces sur le plateau pour tester le "checkmate" pour noir.



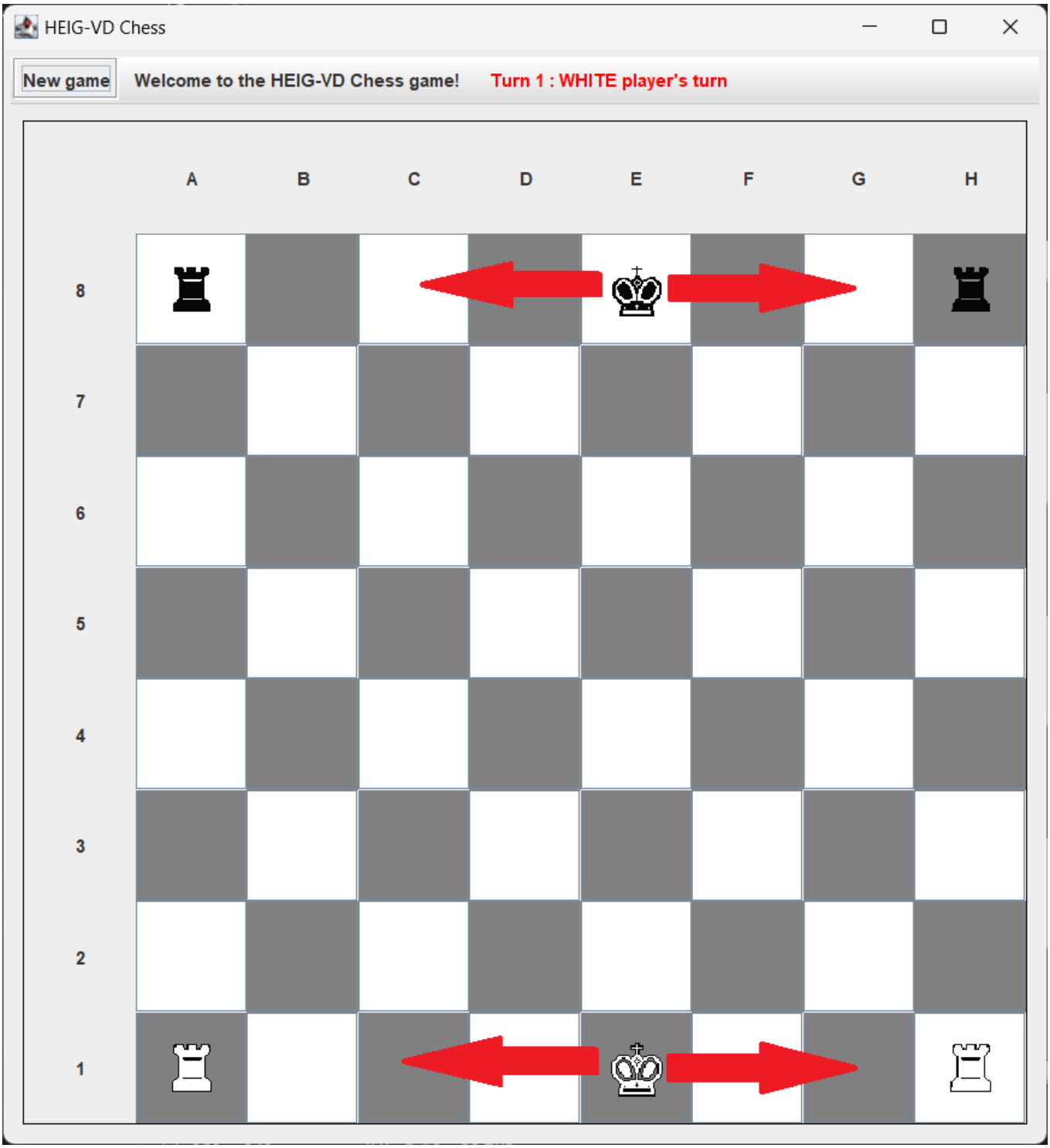
test stalemate

On a quelque pièces sur le plateau pour tester le "stalemate" pour noir.



test castling

On a les deux rois et les quatre tours sur le plateau pour tester les deux "castling" pour les deux couleurs.



Points bonus

L'échec, l'échec et mat et la pat sont implémentés. On a fait plusieurs tests pour vérifier si les fonctions marchent correctement, mais on n'a pas testé toutes les possibilités probablement.

L'implémentation du test si le roi est en échec et mat n'est pas très efficace. Car on essaie de bouger toutes les pièces de la couleur qui doit jouer et on vérifie si le roi est toujours en échec. Si oui, alors c'est un échec et mat. Si non, alors c'est un échec. Ceci fait que beaucoup de mouvements sont faits pour rien. On pourrait améliorer cette implémentation.

Listing du code

File - D:\Projects\Labo_08_Jeu_d_echecs\src\Main.java

```java
import chess.ChessController;
import chess.ChessView;
import chess.views.gui.GUIView;
import engine.GameManager;

public class Main {
    public static void main(String[] args) {

        // 1. Création du contrôleur pour gérer le jeu d'échecs
        ChessController controller = new GameManager();            // Ici, vous devez instancier un ChessController
        // 2. Création de la vue désirée
        ChessView view = new GUIView(controller) ;       // mode GUI
        //ChessView view = new ConsoleView(controller) ; // ou mode Console
        // 3 . Lancement du programme
        controller.start(view) ;

    }
}
```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\GameManager.java

```java
package engine;

import chess.ChessController;
import chess.ChessView;
import engine.utils.BoardDimensions;
import engine.pieces.*;
import engine.utils.Coordinates;

public class GameManager implements ChessController {
    protected ChessView view;
    protected Board board;

    public GameManager() {
        this.board = new Board(BoardDimensions.WIDTH.getValue(),
                BoardDimensions.HEIGHT.getValue());
    }

    /**
     * Update the message displayed by the view
     */
    protected void updateMessage() {
        if (view == null || board == null) return;

        StringBuilder message = new StringBuilder();

        // Add the current turn
        message.append("Turn ").append(board.getTurn()).append(" : ");

        // If the king is in checkmate
        if (board.isCheckMate()) {

            message.append("(Checkmate) ");
            message.append(board.getOpponentPlayer()).append(" player wins");

            // If the king is in stalemate
```

```
36          } else if(board.isStaleMate()){
37              message.append("(Stalemate) ");
38              message.append("Draw");
39
40          }else {
41              // Add the current player
42              message.append(board.getCurrentPlayer()).append(" player's turn ");
43
44              // If the king is in check
45              message.append(board.isCheck() ? "(Check)" : "");
46          }
47
48          // Update the message
49          view.displayMessage(message.toString());
50      }
51
52      /**
53       * Initialize the listeners
54       */
55      private void initListeners() {
56
57          // Add the listener to add pieces
58          board.setAddPieceListener((piece, cell) -> {
59              if (view != null) {
60                  view.putPiece(piece.getType(), piece.getColor(), cell.getX(),
61                          cell.getY());
62              }
63          });
64
65          // Add the listener to remove pieces
66          board.setRemovePieceListener((piece, cell) -> {
67              if (view != null) {
68                  view.removePiece(cell.getX(), cell.getY());
69              }
70          });
```

```
 71
 72         // Add the listener to promote pawns
 73         board.setPromotePawnListener((pawn, cell) -> {
 74             if (view != null) {
 75                 // Ask the user which piece he wants
 76                 ChessView.UserChoice choice = view.askUser("Promotion",
 77                         "Choose a piece to promote your pawn",
 78                         () -> "Queen",
 79                         () -> "Rook",
 80                         () -> "Bishop",
 81                         () -> "Knight");
 82
 83                 Coordinates coordinates = new Coordinates(cell.getX(),
 84                         cell.getY());
 85                 // Set the new piece
 86                 switch (choice.textValue()) {
 87                     case "Queen":
 88                         board.setPiece(new Queen(pawn.getColor()), coordinates);
 89                         break;
 90                     case "Rook":
 91                         board.setPiece(new Rook(pawn.getColor()), coordinates);
 92                         break;
 93                     case "Bishop":
 94                         board.setPiece(new Bishop(pawn.getColor()),
 95                                 coordinates);
 96                         break;
 97                     case "Knight":
 98                         board.setPiece(new Knight(pawn.getColor()),
 99                                 coordinates);
100                         break;
101                 }
102             }
103         });
104     }
105
```

```
106     /**
107      * Start the game
108      *
109      * @param view The view to use
110      */
111     @Override
112     public void start(ChessView view) {
113         this.view = view;
114 
115         // Start the view
116         view.startView();
117 
118         // Initialize the piece listeners
119         initListeners();
120 
121         // Update the message
122         updateMessage();
123     }
124 
125     /**
126      * Move a piece
127      *
128      * @param fromX The X coordinate of the piece to move
129      * @param fromY The Y coordinate of the piece to move
130      * @param toX   The X coordinate of the destination
131      * @param toY   The Y coordinate of the destination
132      * @return True if the movement is valid, false otherwise
133      */
134     @Override
135     public boolean move(int fromX, int fromY, int toX, int toY) {
136 
137         // Check if the movement is valid
138         boolean valid = board.doMovement(fromX, fromY, toX, toY);
139 
140         // Update the message
```

```
141         updateMessage();
142 
143         return valid;
144     }
145 
146     /**
147      * Start a new game
148      */
149     @Override
150     public void newGame() {
151 
152         // Reset the board
153         board.reset();
154 
155         // Put the pieces on the board
156         board.initialize();
157 
158         // Update the message
159         updateMessage();
160     }
161 }
```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\Board.java

```java
package engine;

import chess.PieceType;
import chess.PlayerColor;
import engine.utils.Cell;
import engine.utils.Coordinates;
import engine.pieces.*;

import java.util.Objects;

public class Board {

    /**
     * An interface to listen to the actions on the pieces.
     * The action can be defined in a lambda expression.
     */
    public interface PieceListener {
        void action(Piece piece, Cell cell);
    }

    private final int width;
    private final int height;
    private final Cell[][] cells;
    private PieceListener onAddPiece;
    private PieceListener onRemovePiece;
    private PieceListener onPromotePiece;
    private int turn = 1;

    /**
     * Create a board with the specified dimensions
     *
     * @param width the width of the board
     * @param height the height of the board
     */
    public Board(int width, int height) {
```

```
36
37          this.width = width;
38          this.height = height;
39
40          this.cells = new Cell[width][height];
41
42          reset();
43      }
44
45      /**
46       * Get an array of the colors of the pieces on the board
47       *
48       * @return PlayerColor[][] an array of the colors of the pieces on the board
49       */
50      private PlayerColor[][] getPositionOfPieceColors() {
51          PlayerColor[][] piecesColors = new PlayerColor[this.width][this.height];
52          for (int i = 0; i < this.width; ++i) {
53              for (int j = 0; j < this.height; ++j) {
54                  Piece p = getPieceInBoard(new Coordinates(i, j));
55                  piecesColors[i][j] = p != null ? p.getColor() : null;
56              }
57          }
58          return piecesColors;
59      }
60
61      /**
62       * Get the piece at the specified position
63       *
64       * @param positionInBoard the position of the piece
65       * @return Piece the piece at the specified position
66       */
67      private Piece getPieceInBoard(Coordinates positionInBoard) {
68          return cells[positionInBoard.getX()][positionInBoard.getY()].getPiece();
69      }
70
```

```
 71     /**
 72      * Do the movement of the piece at the specified position to the specified
 73      * position
 74      *
 75      * @param x1 the x position of the piece to move
 76      * @param y1 the y position of the piece to move
 77      * @param x2 the x position of the destination
 78      * @param y2 the y position of the destination
 79      * @return boolean true if the movement is done, false otherwise
 80      */
 81     public boolean doMovement(int x1, int y1, int x2, int y2) {
 82         // Check if the piece is on the board
 83         Piece piece = cells[x1][y1].getPiece();
 84 
 85         // Check if the piece isn't null and if it is the same color as the
 86         // current player
 87         if (piece == null || piece.getColor() != getCurrentPlayer())
 88             return false;
 89 
 90         // Define the initial and final position
 91         Coordinates positionInitial = new Coordinates(x1, y1);
 92         Coordinates positionFinal = new Coordinates(x2, y2);
 93 
 94         // Check if the piece can move to the new position
 95         if (!piece.movementIsOk(positionInitial, positionFinal)) return false;
 96 
 97         // get the position of the pieces' colors on the board
 98         PlayerColor[][] piecesColors = getPositionOfPieceColors();
 99 
100 
101         if (checkPieceInWay(piecesColors, piece, positionInitial, positionFinal)) return false;
102 
103         // King special movements
104         if (piece.getType() == PieceType.KING) {
105             // If the piece is a king, and he would be in check after the
```

```
106             // movement,
107             // the movement is prohibited
108             if (testCheck(positionFinal, piece.getColor())) return false;
109             // If the king is castling, check if the rook is in the correct
110             // position
111             // and if it is the first movement of the king and the rook
112             if (piece.isFirstMovement()) castling(positionFinal);
113         }
114
115         // If the piece is a pawn, and it takes a piece in the "en passant"
116         // movement, remove the piece
117         if (piece.getType() == PieceType.PAWN && Math.abs(x2 - x1) == 1 &&
118                 Math.abs(y2 - y1) == 1) {
119             if (getPieceInBoard(new Coordinates(x2, y1)) != null) {
120                 removePiece(new Coordinates(x2, y1));
121             } else if (getPieceInBoard(new Coordinates(x2, y2)) == null) {
122                 return false;
123             }
124         }
125
126         // do the movement
127         Piece pieceTmp = getPieceInBoard(positionFinal);
128         movePiece(positionInitial, positionFinal);
129
130         // If the king is in check after the movement, the movement is
131         // prohibited
132         if(testCheck(findKing(piece.getColor()), piece.getColor())){
133             movePiece(positionFinal, positionInitial);
134             if(pieceTmp != null) setPiece(pieceTmp, positionFinal);
135             return false;
136         }
137
138         // Update the turn
139         turn++;
140
```

```
141         return true;
142     }
143 
144     /**
145      * Check if there is no piece in the way of the movement
146      *
147      * @param piecesColors the colors of the pieces on the board
148      * @param piece the piece to move
149      * @param positionInitial the position of the piece to move
150      * @param positionFinal the position of the destination
151      * @return boolean true if the movement is done, false otherwise
152      */
153     private boolean checkPieceInWay(PlayerColor[][] piecesColors, Piece piece, Coordinates positionInitial,
    Coordinates positionFinal) {
154         // Get the possible movement of the part from the initial to the final
155         // position
156         // The possible movement is a sequence of coordinates following a step
157         // from the initial position to the final position.
158         Coordinates[] movementPiece = piece.getPossibleMovement(positionInitial,
159                 positionFinal);
160 
161 
162         // position initial -> ... -> position final -> ... -> Max step
163         for (Coordinates positionPiece : movementPiece) {
164             //If there is a piece of the same color as the one making the last
165             // move to the final position, the move is considered forbidden.
166             // Otherwise, (if there is a piece of a different color or no piece
167             // at all), control stops, and the move is allowed.
168             if (Coordinates.equal(positionPiece, positionFinal)) {
169                 if (piecesColors[positionPiece.getX()][positionPiece.getY()] ==
170                         piece.getColor())
171                     return true;
172                 break;
173             }
174 
```

```
175             // If, while traversing the movement, and we have not yet reached
176             // the final position, there is a piece,
177             // regardless of its color, present (obstacle), then the move is
178             // prohibited.
179             if (piecesColors[positionPiece.getX()][positionPiece.getY()] != null)
180                 return true;
181         }
182 
183         return false;
184     }
185 
186     /**
187      * Do the castling movement
188      *
189      * @param positionFinal the position of the king after the castling
190      */
191     private void castling(Coordinates positionFinal) {
192 
193         // Check if the rook is in the correct position
194         if (positionFinal.getX() == 2 || positionFinal.getX() == 6) {
195 
196             Coordinates positionStartRook = new Coordinates(positionFinal.getX() == 2 ? 0 : 7,
197                     positionFinal.getY());
198             Coordinates positionEndRook = new Coordinates(positionFinal.getX() == 2 ? 3 : 5,
199                     positionFinal.getY());
200 
201             // Check if it is the first movement of the king and the rook
202             Piece piece = getPieceInBoard(positionStartRook);
203 
204             if(!(piece instanceof Rook) || !piece.isFirstMovement()) return;
205 
206             // Check if there is a piece between the king and the rook
207             if(checkPieceInWay(getPositionOfPieceColors(), piece, positionStartRook, positionEndRook)) return;
208 
209             // Move the rook
```

```
210             movePiece(positionStartRook, positionEndRook);
211         }
212     }
213
214     /**
215      * Move a piece from the initial position to the final position
216      *
217      * @param positionInitial the initial position of the piece
218      * @param positionFinal the final position of the piece
219      */
220     private void movePiece(Coordinates positionInitial,
221                            Coordinates positionFinal) {
222
223         Piece piece =
224                 cells[positionInitial.getX()][positionInitial.getY()].getPiece();
225
226         removePiece(positionInitial);
227
228         setPiece(piece, positionFinal);
229
230         piece.clearFirstMovement();
231     }
232
233     /**
234      * Check if the king is in check
235      *
236      * @return boolean true if the king is in check, false otherwise
237      */
238     public boolean isCheck() {
239         PlayerColor color = getCurrentPlayer();
240
241         return testCheck(findKing(color), color);
242     }
243
244     /**
```

```
245      * Check if the king is in check
246      *
247      * @param positionKing the position of the king
248      * @param color the color of the king
249      * @return boolean true if the king is in check, false otherwise
250      */
251     private boolean testCheck(Coordinates positionKing, PlayerColor color) {
252 
253         if(positionKing == null || color == null) return false;
254 
255         // Check if the king is in check
256         for (int i = 0; i < width; i++) {
257             for (int j = 0; j < height; j++) {
258                 Coordinates positionPiece = new Coordinates(i, j);
259                 Piece piece = getPieceInBoard(positionPiece);
260                 if (piece != null && piece.getColor() != color &&
261                         piece.movementIsOk(positionPiece, positionKing) &&
262                         !checkPieceInWay(getPositionOfPieceColors(), piece, positionPiece, positionKing))
263                     return true;
264             }
265         }
266 
267         return false;
268     }
269 
270     /**
271      * Check if the king is in checkmate
272      *
273      * @return boolean true if the king is in checkmate, false otherwise
274      */
275     public boolean isCheckMate() {
276         PlayerColor color = getCurrentPlayer();
277 
278         // Check if the king is in check
279         if ( !isCheck() ) return false;
```

```
280
281         // Check if any piece can move to a position where the king is not in
282         // check
283
284         Coordinates positionKing = findKing(getCurrentPlayer());
285
286         for (int i = 0; i < this.cells.length; ++i){
287             for(int j = 0; j < this.cells[i].length; ++j){
288                 Piece piece = this.cells[i][j].getPiece();
289
290                 if(piece == null || piece.getColor() != color ||
291                         piece instanceof King) continue;
292
293                 Coordinates startPosition = new Coordinates(i, j);
294
295                 if (tryEveryMoveToSaveKing(piece, positionKing, startPosition, color))
296                     return false;
297             }
298         }
299
300         // If the king is in check, and no piece can move to a position where
301         // the king is not in check, the king is in checkmate
302         return true;
303     }
304
305     /**
306      * Try every move of the piece to save the king
307      *
308      * @param piece the piece to move
309      * @param positionKing the position of the king
310      * @param startPosition the position of the piece to move
311      * @param color the color of the piece to move
312      * @return boolean true if the king is saved, false otherwise
313      */
314     private boolean tryEveryMoveToSaveKing(Piece piece,
```

```
315                                             Coordinates positionKing,
316                                             Coordinates startPosition,
317                                             PlayerColor color) {
318         for (int i2 = 0; i2 < this.cells.length; ++i2){
319             for(int j2 = 0; j2 < this.cells[i2].length; ++j2){
320 
321                 Coordinates positionTmp = new Coordinates(i2, j2);
322 
323                 if(piece.movementIsOk(new Coordinates(i2, j2),
324                         positionKing)){
325                     if(!checkPieceInWay(getPositionOfPieceColors(),
326                             piece, positionTmp, positionKing)){
327 
328                         Piece pieceTmp = getPieceInBoard(new Coordinates(i2, j2));
329 
330                         movePiece(startPosition, positionTmp);
331 
332                         if(!testCheck(positionKing, color)){
333                             movePiece(positionTmp, startPosition);
334                             if(pieceTmp != null) setPiece(pieceTmp, positionTmp);
335                             return true;
336                         }
337                     }
338                 }
339             }
340         }
341         return false;
342     }
343 
344     /**
345      * Check if the king can not move in any direction
346      *
347      * @param positionKing the position of the king
348      * @return boolean true if the king can not move, false otherwise
349      */
```

```
350     private boolean checkIfKingCanNotMove(Coordinates positionKing) {
351
352         if(positionKing == null) return false;
353
354         for (int i = positionKing.getX() - 1; i <= positionKing.getX() + 1; i++) {
355             for (int j = positionKing.getY() - 1; j <= positionKing.getY() + 1; j++) {
356                 if ((i == positionKing.getX() && j == positionKing.getY()) || i > width -1 || j > height -1
    || i < 0 || j < 0) continue;
357                 if( !testCheck(new Coordinates(i, j), getCurrentPlayer()) ) return false;
358             }
359         }
360
361         return true;
362     }
363
364     /**
365      * Check if the king is in stalemate
366      *
367      * @return boolean true if the king is in stalemate, false otherwise
368      */
369     public boolean isStaleMate() {
370         PlayerColor color = getCurrentPlayer();
371
372         //Find the king
373         Coordinates positionKing = findKing(color);
374
375         if (positionKing == null) return false;
376
377         // Check if the king is in check
378         if (isCheck()) return false;
379
380         //Check if the king can make a move
381         return checkIfKingCanNotMove(positionKing);
382     }
383
```

```
384     /**
385      * Reset the board
386      */
387     public void reset() {
388 
389         // Remove all pieces from the board
390         for (int i = 0; i < width; i++)
391             for (int j = 0; j < height; j++) {
392                 cells[i][j] = new Cell(null, i, j);
393                 removePiece(new Coordinates(i, j));
394             }
395 
396         // Reset the turn counter
397         turn = 1;
398     }
399 
400     /**
401      * Initialize the board, set all the pieces on the board
402      */
403     public void initialize() {
404 
405         // Put the pieces on the board
406         // White pieces
407         addPiece(new Rook(PlayerColor.WHITE), new Coordinates(0, 0));
408         addPiece(new Knight(PlayerColor.WHITE), new Coordinates(1, 0));
409         addPiece(new Bishop(PlayerColor.WHITE), new Coordinates(2, 0));
410         addPiece(new Queen(PlayerColor.WHITE), new Coordinates(3, 0));
411         addPiece(new King(PlayerColor.WHITE), new Coordinates(4, 0));
412         addPiece(new Bishop(PlayerColor.WHITE), new Coordinates(5, 0));
413         addPiece(new Knight(PlayerColor.WHITE), new Coordinates(6, 0));
414         addPiece(new Rook(PlayerColor.WHITE), new Coordinates(7, 0));
415         for (int i = 0; i < 8; i++) {
416             addPiece(new Pawn(PlayerColor.WHITE), new Coordinates(i, 1));
417         }
418 
```

```
419         // Black pieces
420         addPiece(new Rook(PlayerColor.BLACK), new Coordinates(0, 7));
421         addPiece(new Knight(PlayerColor.BLACK), new Coordinates(1, 7));
422         addPiece(new Bishop(PlayerColor.BLACK), new Coordinates(2, 7));
423         addPiece(new Queen(PlayerColor.BLACK), new Coordinates(3, 7));
424         addPiece(new King(PlayerColor.BLACK), new Coordinates(4, 7));
425         addPiece(new Bishop(PlayerColor.BLACK), new Coordinates(5, 7));
426         addPiece(new Knight(PlayerColor.BLACK), new Coordinates(6, 7));
427         addPiece(new Rook(PlayerColor.BLACK), new Coordinates(7, 7));
428         for (int i = 0; i < 8; i++) {
429             addPiece(new Pawn(PlayerColor.BLACK), new Coordinates(i, 6));
430         }
431     }
432 
433     /**
434      * Add a piece on the board
435      *
436      * @param piece the piece to add
437      * @param position the position of the piece
438      */
439     protected void addPiece(Piece piece, Coordinates position) {
440         setPiece(piece, position);
441     }
442 
443     /**
444      * Remove a piece from the board
445      *
446      * @param piece the piece to remove
447      * @param position the position of the piece to remove
448      */
449     public void setPiece(Piece piece, Coordinates position) {
450 
451         Objects.requireNonNull(piece, "Piece cannot be null");
452 
453         checkPositionOnBoard(position);
```

```
454
455         cells[position.getX()][position.getY()].setPiece(piece);
456
457         if (onAddPiece != null) {
458             onAddPiece.action(piece, cells[position.getX()][position.getY()]);
459         }
460
461         if (piece instanceof Pawn && (position.getY() == 0 ||
462                 position.getY() == 7)) {
463             if (onPromotePiece != null) {
464                 onPromotePiece.action(piece,
465                         cells[position.getX()][position.getY()]);
466             }
467         }
468     }
469
470     /**
471      * Remove a piece from the board
472      *
473      * @param position the position of the piece to remove
474      */
475     private void removePiece(Coordinates position) {
476         checkPositionOnBoard(position);
477         cells[position.getX()][position.getY()].removePiece();
478
479         if (onRemovePiece != null) {
480             onRemovePiece.action(null, cells[position.getX()][position.getY()]);
481         }
482     }
483
484     /**
485      * Check if the position is on the board
486      *
487      * @param position the position to check
488      */
```

```
489     private void checkPositionOnBoard(Coordinates position) {
490         if (position.getX() < 0 || position.getX() >= width || position.getY() < 0 || position.getY() >=
    height)
491             throw new IllegalArgumentException("Position out of board");
492     }
493 
494     /**
495      * Get the current player
496      *
497      * @return PlayerColor the color current player
498      */
499     public PlayerColor getCurrentPlayer() {
500         return turn % 2 == 1 ? PlayerColor.WHITE : PlayerColor.BLACK;
501     }
502 
503     /**
504      * Get the opponent player
505      *
506      * @return PlayerColor the color of the opponent player
507      */
508     public PlayerColor getOpponentPlayer() {
509         return turn % 2 == 0 ? PlayerColor.WHITE : PlayerColor.BLACK;
510     }
511 
512     /**
513      * Set the listener to add pieces
514      *
515      * @param listener the listener to add pieces
516      */
517     public void setAddPieceListener(PieceListener listener) {
518         this.onAddPiece = listener;
519     }
520 
521     /**
522      * Set the listener to remove pieces
```

```
523      *
524      * @param listener the listener to remove pieces
525      */
526     public void setRemovePieceListener(PieceListener listener) {
527         this.onRemovePiece = listener;
528     }
529 
530     /**
531      * Set the listener to promote pawns
532      *
533      * @param listener the listener to promote pawns
534      */
535     public void setPromotePawnListener(PieceListener listener) {
536         this.onPromotePiece = listener;
537     }
538 
539     /**
540      * Get the turn number
541      *
542      * @return int the turn number
543      */
544     public int getTurn() {
545         return turn;
546     }
547 
548     /**
549      * Find the king of the specified color
550      *
551      * @param color the color of the king
552      * @return Coordinates the coordinates where the king is
553      */
554     private Coordinates findKing(PlayerColor color) {
555         for (int i = 0; i < width; i++)
556             for (int j = 0; j < height; j++)
557                 if (cells[i][j].getPiece() instanceof King &&
```

```
558                         cells[i][j].getPiece().getColor() == color)
559                     return new Coordinates(i, j);
560         return null;
561     }
562 }
```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\utils\BoardDimensions.java

```java
package engine.utils;

public enum BoardDimensions {
    WIDTH(8),    // Width dimension of the board
    HEIGHT(8),   // Height dimension of the board
    DIAGONAL(8); // Diagonal dimension of the board

    private final int value;

    /**
     * Constructor to initialize a BoardDimensions enum value with a specific integer value.
     *
     * @param value the integer value associated with the enum dimension
     */
    BoardDimensions(int value) {
        this.value = value;
    }

    /**
     * Get the integer value associated with the enum dimension.
     *
     * @return the integer value
     */
    public int getValue() {
        return this.value;
    }
}

```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\utils\Cell.java

```java
package engine.utils;

import engine.pieces.*;

public class Cell {

    private Piece piece;

    private final int x;
    private final int y;

    /**
     * Constructor to initialize a Cell object with a piece and its position (x, y).
     *
     * @param piece the piece placed on the cell (can be null for an empty cell)
     * @param x     the x-coordinate of the cell's position
     * @param y     the y-coordinate of the cell's position
     */
    public Cell(Piece piece, int x, int y) {
        this.piece = piece;
        this.x = x;
        this.y = y;
    }

    /**
     * Set the piece on the cell to a new piece.
     *
     * @param piece the new piece to be placed on the cell
     */
    public void setPiece(Piece piece) {
        this.piece = piece;
    }

    /**
     * Get the piece currently on the cell.
```

```
36      *
37      * @return the piece on the cell (can be null for an empty cell)
38      */
39     public Piece getPiece() {
40         return piece;
41     }
42
43     /**
44      * Check if the cell is empty (contains no piece).
45      *
46      * @return true if the cell is empty, false otherwise
47      */
48     public boolean isEmpty() {
49         return piece == null;
50     }
51
52     /**
53      * Remove the piece from the cell, making it empty.
54      */
55     public void removePiece() {
56         piece = null;
57     }
58
59     /**
60      * Get the x-coordinate of the cell's position.
61      *
62      * @return the x-coordinate of the cell
63      */
64     public int getX() {
65         return x;
66     }
67
68     /**
69      * Get the y-coordinate of the cell's position.
70      *
```

```
71      * @return the y-coordinate of the cell
72      */
73     public int getY() {
74         return y;
75     }
76 }
77
```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\utils\Coordinates.java

```java
package engine.utils;

public class Coordinates {
    int x;
    int y;

    /**
     * Constructor to initialize a Coordinates object with given x and y coordinates.
     *
     * @param x the x-coordinate
     * @param y the y-coordinate
     */
    public Coordinates(int x, int y) {
        this.x = x;
        this.y = y;
    }

    /**
     * Copy constructor to create a new Coordinates object with the same coordinates as the provided Coordinates object.
     *
     * @param coordinates the Coordinates object to copy from
     */
    public Coordinates(Coordinates coordinates) {
        copy(coordinates);
    }

    /**
     * Copy the coordinates from another Coordinates object to this object.
     *
     * @param coordinates the Coordinates object to copy from
     */
    public void copy(Coordinates coordinates) {
        this.x = coordinates.x;
        this.y = coordinates.y;
```

```
35      }
36
37      /**
38       * Set new coordinates for this object.
39       *
40       * @param x the new x-coordinate
41       * @param y the new y-coordinate
42       */
43      public void setCoordinates(int x, int y) {
44          this.x = x;
45          this.y = y;
46      }
47
48      /**
49       * Add the coordinates of another Coordinates object to this object.
50       *
51       * @param coordinates the Coordinates object to add
52       */
53      public void add(Coordinates coordinates) {
54          Coordinates c = addition(this, coordinates);
55          setCoordinates(c.getX(), c.getY());
56      }
57
58      /**
59       * Set a new value for the x-coordinate.
60       *
61       * @param x the new x-coordinate
62       */
63      public void setX(int x) {
64          this.x = x;
65      }
66
67      /**
68       * Get the x-coordinate of this object.
69       *
```

```
 70      * @return the x-coordinate
 71      */
 72     public int getX() {
 73         return this.x;
 74     }
 75
 76     /**
 77      * Set a new value for the y-coordinate.
 78      *
 79      * @param y the new y-coordinate
 80      */
 81     public void setY(int y) {
 82         this.y = y;
 83     }
 84
 85     /**
 86      * Get the y-coordinate of this object.
 87      *
 88      * @return the y-coordinate
 89      */
 90     public int getY() {
 91         return this.y;
 92     }
 93
 94     /**
 95      * Calculate the change in Y-coordinate between two Coordinates objects.
 96      *
 97      * @param positionInitial the initial position
 98      * @param positionFinal   the final position
 99      * @return the change in Y-coordinate
100      */
101     public static int deltaY(Coordinates positionInitial, Coordinates positionFinal) {
102         return positionFinal.y - positionInitial.y;
103     }
104
```

```
105     /**
106      * Calculate the change in X-coordinate between two Coordinates objects.
107      *
108      * @param positionInitial the initial position
109      * @param positionFinal   the final position
110      * @return the change in X-coordinate
111      */
112     public static int deltaX(Coordinates positionInitial, Coordinates positionFinal) {
113         return positionFinal.x - positionInitial.x;
114     }
115 
116     /**
117      * Calculate the angle in radians between two Coordinates objects.
118      *
119      * @param positionInitial the initial position
120      * @param positionFinal   the final position
121      * @return the angle in radians
122      */
123     public static double getAngle(Coordinates positionInitial, Coordinates positionFinal) {
124         int deltaX = Coordinates.deltaX(positionInitial, positionFinal);
125         int deltaY = Coordinates.deltaY(positionInitial, positionFinal);
126         return Math.atan2(deltaY, deltaX);
127     }
128 
129     /**
130      * Calculate the angle in degrees between two Coordinates objects.
131      *
132      * @param positionInitial the initial position
133      * @param positionFinal   the final position
134      * @return the angle in degrees
135      */
136     public static double getAngleDegree(Coordinates positionInitial, Coordinates positionFinal) {
137         return Math.toDegrees(getAngle(positionInitial, positionFinal));
138     }
139 
```

```
140     /**
141      * Perform vector addition of two Coordinates objects and return a new Coordinates object.
142      *
143      * @param c1 the first Coordinates object
144      * @param c2 the second Coordinates object
145      * @return a new Coordinates object representing the result of the addition
146      */
147     public static Coordinates addition(Coordinates c1, Coordinates c2) {
148         return new Coordinates(c1.getX() + c2.getX(), c1.getY() + c2.getY());
149     }
150 
151     /**
152      * Check if two Coordinates objects are equal in terms of their x and y coordinates.
153      *
154      * @param c1 the first Coordinates object
155      * @param c2 the second Coordinates object
156      * @return true if the coordinates are equal, false otherwise
157      */
158     public static boolean equal(Coordinates c1, Coordinates c2) {
159         return (c1.getX() == c2.getX()) && (c1.getY() == c2.getY());
160     }
161 }
162 
```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\pieces\Piece.java

```java
package engine.pieces;

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

public abstract class Piece {
    private final PieceType type;
    private final PlayerColor color;
    private Movements movements;
    protected boolean firstMovement = true;

    public Piece(PieceType type, PlayerColor color, Movements movements) {
        if (type == null)
            throw new IllegalArgumentException("Piece type cannot be null");
        if (color == null)
            throw new IllegalArgumentException("Player color cannot be null");
        if (movements == null)
            throw new IllegalArgumentException("Movements cannot be null");

        this.type = type;
        this.color = color;
        this.movements = movements;
    }

    protected void setMovements(Movements movements) {
        this.movements = movements;
    }


    public PieceType getType() {
        return this.type;

    }
```

```
36
37      public PlayerColor getColor() {
38          return color;
39      }
40
41      public Coordinates[] getPossibleMovement(Coordinates positionInitial,
42                                               Coordinates positionFinal) {
43          Step possibleStep = this.movements.getPossibleStep(positionInitial,
44                  positionFinal);
45          if (possibleStep == null) return null;
46
47          Coordinates[] possibleMovement = possibleStep.getMovement();
48          if (possibleMovement == null) return null;
49
50          Coordinates[] m = new Coordinates[possibleMovement.length];
51          for (int i = 0; i < possibleMovement.length; ++i)
52              m[i] = new Coordinates(Coordinates.addition(possibleMovement[i],
53                      positionInitial));
54
55          return m;
56      }
57
58      public boolean movementIsOk(Coordinates positionInitial,
59                                  Coordinates positionFinal) {
60          Coordinates[] possibleMovement = getPossibleMovement(positionInitial,
61                  positionFinal);
62          if (possibleMovement == null) return false;
63
64          for (Coordinates c : possibleMovement) {
65              if (Coordinates.equal(c, positionFinal)) return true;
66          }
67          return false;
68      }
69
70      public boolean isFirstMovement() {
```

```
71          return this.firstMovement;
72      }
73
74      public void clearFirstMovement() {
75          this.firstMovement = false;
76      }
77  }
```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\pieces\Knight.java

```java
package engine.pieces;

import chess.PieceType;
import chess.PlayerColor;
import engine.movements.Movements;
import engine.movements.Step;
import engine.utils.Coordinates;

public class Knight extends Piece {
    public Knight(PlayerColor color) {
        super(PieceType.KNIGHT, color, new Movements(new Step[]{
                //Special
                new Step(new Coordinates(1, 2), 1, color == PlayerColor.BLACK
                        , color == PlayerColor.BLACK),
                new Step(new Coordinates(1, -2), 1,
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(-1, 2), 1,
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(-1, -2), 1,
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),

                new Step(new Coordinates(2, 1), 1, color == PlayerColor.BLACK
                        , color == PlayerColor.BLACK),
                new Step(new Coordinates(2, -1), 1,
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(-2, 1), 1,
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(-2, -1), 1,
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
        }));
    }
}
```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\pieces\Pawn.java

```java
package engine.pieces;

import chess.PieceType;
import chess.PlayerColor;
import engine.movements.Movements;
import engine.movements.Step;
import engine.utils.Coordinates;

public class Pawn extends Piece {

    public Pawn(PlayerColor color) {
        super(PieceType.PAWN, color, new Movements(new Step[]{
                //Vertical
                new Step(new Coordinates(0, 1), 2, color == PlayerColor.BLACK
                        , color == PlayerColor.BLACK),
                // "En passant" or attack
                new Step(new Coordinates(1, 1), 1, color == PlayerColor.BLACK
                        , color == PlayerColor.BLACK),
                new Step(new Coordinates(-1, 1), 1,
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
        }));
        this.firstMovement = true;
    }

    public void clearFirstMovement() {
        super.setMovements(new Movements(new Step[]{
                //Vertical
                new Step(new Coordinates(0, 1), 1,
                        super.getColor() == PlayerColor.BLACK,
                        super.getColor() == PlayerColor.BLACK),
                // "En passant" or attack
                new Step(new Coordinates(1, 1), 1,
                        super.getColor() == PlayerColor.BLACK,
                        super.getColor() == PlayerColor.BLACK),
                new Step(new Coordinates(-1, 1), 1,
```

```
36                          super.getColor() == PlayerColor.BLACK,
37                          super.getColor() == PlayerColor.BLACK),
38          }));
39          this.firstMovement = false;
40      }
41  }
```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\pieces\Queen.java

```java
package engine.pieces;

import chess.PieceType;
import chess.PlayerColor;
import engine.movements.Movements;
import engine.movements.Step;
import engine.utils.BoardDimensions;
import engine.utils.Coordinates;

public class Queen extends Piece {
    public Queen(PlayerColor color) {
        super(PieceType.QUEEN, color, new Movements(new Step[]{
                //Horizontal
                new Step(new Coordinates(1, 0),
                        BoardDimensions.WIDTH.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(-1, 0),
                        BoardDimensions.WIDTH.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),

                //Vertical
                new Step(new Coordinates(0, 1),
                        BoardDimensions.HEIGHT.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(0, -1),
                        BoardDimensions.HEIGHT.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),

                //Diagonal
                new Step(new Coordinates(1, 1),
                        BoardDimensions.DIAGONAL.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(1, -1),
                        BoardDimensions.DIAGONAL.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
```

```
36                  new Step(new Coordinates(-1, 1),
37                          BoardDimensions.DIAGONAL.getValue(),
38                          color == PlayerColor.BLACK, color == PlayerColor.BLACK),
39                  new Step(new Coordinates(-1, -1),
40                          BoardDimensions.DIAGONAL.getValue(),
41                          color == PlayerColor.BLACK, color == PlayerColor.BLACK),
42          }));
43      }
44 }
```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\pieces\Rook.java

```java
package engine.pieces;

import chess.PieceType;
import chess.PlayerColor;
import engine.movements.Movements;
import engine.movements.Step;
import engine.utils.BoardDimensions;
import engine.utils.Coordinates;

public class Rook extends Piece {
    public Rook(PlayerColor color) {
        super(PieceType.ROOK, color, new Movements(new Step[]{
                //Horizontal
                new Step(new Coordinates(1, 0),
                        BoardDimensions.WIDTH.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(-1, 0),
                        BoardDimensions.WIDTH.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),

                //Vertical
                new Step(new Coordinates(0, 1),
                        BoardDimensions.HEIGHT.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(0, -1),
                        BoardDimensions.HEIGHT.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
        }));
    }
}
```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\pieces\Bishop.java

```java
package engine.pieces;

import chess.PieceType;
import chess.PlayerColor;
import engine.movements.Movements;
import engine.movements.Step;
import engine.utils.BoardDimensions;
import engine.utils.Coordinates;

public class Bishop extends Piece {
    public Bishop(PlayerColor color) {
        super(PieceType.BISHOP, color, new Movements(new Step[]{
                //Diagonal
                new Step(new Coordinates(1, 1),
                        BoardDimensions.DIAGONAL.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(1, -1),
                        BoardDimensions.DIAGONAL.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(-1, 1),
                        BoardDimensions.DIAGONAL.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(-1, -1),
                        BoardDimensions.DIAGONAL.getValue(),
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
        }));
    }
}
```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\pieces\King.java

```java
package engine.pieces;

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

public class King extends Piece {

    public King(PlayerColor color) {
        super(PieceType.KING, color, new Movements(new Step[]{
                //Horizontal
                new Step(new Coordinates(1, 0), 2, color == PlayerColor.BLACK
                        , color == PlayerColor.BLACK),
                new Step(new Coordinates(-1, 0), 2,
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),

                //Vertical
                new Step(new Coordinates(0, 1), 1, color == PlayerColor.BLACK
                        , color == PlayerColor.BLACK),
                new Step(new Coordinates(0, -1), 1,
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),

                //Diagonal
                new Step(new Coordinates(1, 1), 1, color == PlayerColor.BLACK
                        , color == PlayerColor.BLACK),
                new Step(new Coordinates(1, -1), 1,
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(-1, 1), 1,
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
                new Step(new Coordinates(-1, -1), 1,
                        color == PlayerColor.BLACK, color == PlayerColor.BLACK),
        }));
    }
```

```
36
37
38      public void clearFirstMovement() {
39          super.setMovements(new Movements(new Step[]{
40                  //Horizontal
41                  new Step(new Coordinates(1, 0), 1,
42                          super.getColor() == PlayerColor.BLACK,
43                          super.getColor() == PlayerColor.BLACK),
44                  new Step(new Coordinates(-1, 0), 1,
45                          super.getColor() == PlayerColor.BLACK,
46                          super.getColor() == PlayerColor.BLACK),
47
48                  //Vertical
49                  new Step(new Coordinates(0, 1), 1,
50                          super.getColor() == PlayerColor.BLACK,
51                          super.getColor() == PlayerColor.BLACK),
52                  new Step(new Coordinates(0, -1), 1,
53                          super.getColor() == PlayerColor.BLACK,
54                          super.getColor() == PlayerColor.BLACK),
55
56                  //Diagonal
57                  new Step(new Coordinates(1, 1), 1,
58                          super.getColor() == PlayerColor.BLACK,
59                          super.getColor() == PlayerColor.BLACK),
60                  new Step(new Coordinates(1, -1), 1,
61                          super.getColor() == PlayerColor.BLACK,
62                          super.getColor() == PlayerColor.BLACK),
63                  new Step(new Coordinates(-1, 1), 1,
64                          super.getColor() == PlayerColor.BLACK,
65                          super.getColor() == PlayerColor.BLACK),
66                  new Step(new Coordinates(-1, -1), 1,
67                          super.getColor() == PlayerColor.BLACK,
68                          super.getColor() == PlayerColor.BLACK),
69          }));
70          this.firstMovement = false;
```

```
71      }
72  }
```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\movements\Movements.java

```java
package engine.movements;

import engine.utils.Coordinates;

public class Movements {
    Step[] steps;

    public Movements(Step[] steps) {
        if (steps == null)
            throw new IllegalArgumentException("step cannot be null");
        this.steps = steps;
    }

    /**
     * Get the possible step for the initial position and the final position
     *
     * @param positionInitial the initial position
     * @param positionFinal   the final position
     * @return the possible step or null if there is no possible step
     */
    public Step getPossibleStep(Coordinates positionInitial,
                                Coordinates positionFinal) {
        double angleDegree = Coordinates.getAngleDegree(positionInitial,
                positionFinal);
        for (Step step : this.steps) {
            if (step.stepAngleDegreeIsOk(angleDegree)) return step;
        }
        return null;
    }
}
```

File - D:\Projects\Labo_08_Jeu_d_echecs\src\engine\movements\Step.java

```java
package engine.movements;

import engine.utils.Coordinates;

public class Step {
    private Coordinates coordinates;
    private Coordinates[] movement;
    private double stepAngleDegree;

    public Step(Coordinates step, int maxStep, boolean invertOrdinateAxis,
                boolean invertAbscissaAxis) {
        if (step == null)
            throw new IllegalArgumentException("step cannot be null");
        if (maxStep < 0)
            throw new IllegalArgumentException("maxStep cannot be negative");
        int x = invertAbscissaAxis ? (step.getX() * (-1)) : step.getX();
        int y = invertOrdinateAxis ? (step.getY() * (-1)) : step.getY();
        this.coordinates = new Coordinates(x, y);

        Coordinates originCoordinates = new Coordinates(0, 0);
        this.stepAngleDegree = Coordinates.getAngleDegree(originCoordinates,
                this.coordinates);

        this.movement = new Coordinates[maxStep];
        for (int i = 0; i < this.movement.length; ++i)
            this.movement[i] = nextStep(i);
    }

    /**
     * Get the next step for the number of step given
     *
     * @param nbStep the number of step
     * @return the next step
     */
    private Coordinates nextStep(int nbStep) {
```

```java
        int x = this.coordinates.getX();
        int y = this.coordinates.getY();
        x = x == 0 ? 0 : (x < 0 ? (x - nbStep) : (x + nbStep));
        y = y == 0 ? 0 : (y < 0 ? (y - nbStep) : (y + nbStep));
        return new Coordinates(x, y);
    }

    /**
     * Check if the angle of the step is the same as the angle of the movement
     *
     * @param stepAngleDegree the angle of the step
     * @return true if the angle of the step is the same as the angle of the
     *         movement
     */
    public boolean stepAngleDegreeIsOk(double stepAngleDegree) {
        return this.stepAngleDegree == stepAngleDegree;
    }

    /**
     * Get the coordinates of the movement
     *
     * @return the coordinates
     */
    public Coordinates[] getMovement() {
        return this.movement;
    }
}
```