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 }
```