

```

1  package engine;
2
3  import chess.ChessController;
4  import chess.ChessView;
5  import chess.views.console.ConsoleView;
6  import chess.views.gui.GUIView;
7
8  /**
9   * Classe lançant le programme du jeu d'échecs.
10  * Il est possible de jouer via un GUI ou en Console selon les envies.
11  *
12  * @author Jonathan Friedli
13  * @author Valentin Kaelin
14  */
15  public class Main {
16      public static void main(String[] args) {
17          ChessController controller = new GameManager();
18
19          // Choix de la vue : mode GUI ou mode Console
20          ChessView view = new GUIView(controller);
21          // ChessView view = new ConsoleView(controller);
22
23          controller.start(view);
24      }
25  }
26
27  // -----
28
29  package engine;
30
31  import chess.ChessController;
32  import chess.ChessView;
33  import chess.views.gui.GUIView;
34
35  /**
36   * Classe lançant le programme de test du jeu d'échecs.
37   * Il est possible de définir quelle position initiale choisir pour les pièces
38   * afin de tester une pièce ou un mouvement spécifique.
39   *
40   * @author Jonathan Friedli
41   * @author Valentin Kaelin
42   */
43  public class MainTest {
44      public static void main(String[] args) {
45          // Choix du test à lancer
46          GameManagerTest.Type type = GameManagerTest.Type.QUEEN;
47          ChessController controller = new GameManagerTest(type);
48
49          ChessView view = new GUIView(controller);
50
51          controller.start(view);
52      }
53  }
54
55  // -----
56
57  package engine;
58
59  import chess.ChessController;
60  import chess.ChessView;
61  import chess.PlayerColor;
62  import engine.pieces.*;
63  import engine.utils.Cell;
64
65  import java.util.Objects;
66
67
68
69

```

```

70  /**
71  * Classe principale de la gestion du jeu d'échecs.
72  * Elle s'occupe de démarrer le jeu ainsi qu'écouter et répondre aux événements de
73  * la view.
74  *
75  * @author Jonathan Friedli
76  * @author Valentin Kaelin
77  */
78  public class GameManager implements ChessController {
79      private ChessView view;
80      private Board board;
81
82      /**
83       * @return le plateau de jeu
84       */
85      protected Board getBoard() {
86          return board;
87      }
88
89      /**
90       * Met à jour le message de la vue
91       */
92      private void updateDisplayMessage() {
93          if (view == null || board == null)
94              return;
95
96          StringBuilder msg = new StringBuilder(
97              "Aux " + (board.currentPlayer() == PlayerColor.WHITE ? "blancs" : "noirs")
98          );
99          if (board.isCheck(board.currentPlayer()))
100              msg.append(" CHECK!");
101          view.displayMessage(msg.toString());
102      }
103
104      /**
105       * Initialise le plateau, écoute les différents événements
106       */
107      private void initBoard() {
108          board = new Board();
109
110          // Events listeners
111          board.setAddPieceListener((piece, cell) -> {
112              if (view != null)
113                  view.putPiece(piece.getType(), piece.getColor(), cell.getX(), cell.getY());
114          });
115
116          board.setRemovePieceListener((piece, cell) -> {
117              if (view != null)
118                  view.removePiece(cell.getX(), cell.getY());
119          });
120
121          board.setPromotionListener((piece) -> {
122              Cell cell = piece.getCell();
123              PlayerColor color = piece.getColor();
124              Piece[] choices = {
125                  new Queen(board, cell, color),
126                  new Knight(board, cell, color),
127                  new Rook(board, cell, color),
128                  new Bishop(board, cell, color)
129              };
130
131              Piece userChoice;
132              while ((userChoice = view.askUser("Promotion",
133                  "Choisir une pièce pour la promotion", choices)) == null) {
134              }
135              board.removePiece(cell);
136              board.setPiece(userChoice, cell);
137          });
138      }

```

```

139
140     @Override
141     public void start(ChessView view) {
142         Objects.requireNonNull(view, "View invalide");
143         this.view = view;
144         view.startView();
145         initBoard();
146         board.fillBoard();
147         updateDisplayMessage();
148     }
149
150     @Override
151     public boolean move(int fromX, int fromY, int toX, int toY) {
152         if (board == null)
153             return false;
154
155         Cell from = new Cell(fromX, fromY);
156         Cell to = new Cell(toX, toY);
157
158         if (!board.move(from, to)) {
159             updateDisplayMessage();
160             return false;
161         }
162
163         updateDisplayMessage();
164
165         return true;
166     }
167
168     @Override
169     public void newGame() {
170         board.resetBoard();
171         board.fillBoard();
172         updateDisplayMessage();
173     }
174 }
175
176 // -----
177
178 package engine;
179
180 import chess.ChessView;
181 import chess.PlayerColor;
182 import engine.pieces.*;
183 import engine.utils.Cell;
184
185 /**
186  * Classe permettant de tester rapidement diverses situations initiales.
187  * Il est possible de spécifier quelle pièce ou mouvement nous souhaitons tester
188  * grâce au constructeur.
189  *
190  * @author Jonathan Friedli
191  * @author Valentin Kaelin
192  */
193 public class GameManagerTest extends GameManager {
194     enum Type {
195         CHECK, CASTLE, ROOK, BISHOP, KING, KNIGHT, PAWN, QUEEN
196     }
197
198     private final Type type;
199
200     /**
201      * Crée une instance de test
202      *
203      * @param type : pièce ou mouvement à tester
204      */
205     public GameManagerTest(Type type) {
206         this.type = type;
207     }

```

```

208
209 @Override
210 public void start(ChessView view) {
211     super.start(view);
212     newGame();
213 }
214
215 @Override
216 public void newGame() {
217     getBoard().resetBoard();
218     // Applique la situation initiale de test
219     fillBoard(type);
220 }
221
222 /**
223  * Remplit le plateau selon le test choisi
224  *
225  * @param type : pièce ou mouvement à tester
226  */
227 private void fillBoard(Type type) {
228     switch (type) {
229         case CHECK:
230             testCheck();
231             break;
232         case CASTLE:
233             testCastle();
234             break;
235         case ROOK:
236             testRook();
237             break;
238         case BISHOP:
239             testBishop();
240             break;
241         case KING:
242             testKing();
243             break;
244         case KNIGHT:
245             testKnight();
246             break;
247         case PAWN:
248             testPawn();
249             break;
250         case QUEEN:
251             testQueen();
252             break;
253         default:
254             break;
255     }
256 }
257
258 /**
259  * Ajoute des pions de la couleur souhaitée autour de la position spécifiée
260  *
261  * @param pos position que l'on souhaite entourer
262  * @param color couleur de pions
263  */
264 private void pawnAroundPos(Cell pos, PlayerColor color) {
265     for (int i = pos.getX() - 1; i < pos.getX() + 2; i++) {
266         for (int j = pos.getY() - 1; j < pos.getY() + 2; j++) {
267             if (i == pos.getX() && j == pos.getY()) {
268                 continue;
269             }
270             getBoard().addPiece(new Pawn(getBoard(), new Cell(i, j), color));
271         }
272     }
273 }
274
275
276

```

```

277  /**
278   * Permet de rapidement position des pièces afin de tester le fonctionnement des
279   * Cavaliers
280   */
281  private void testKnight() {
282      Cell knight1 = new Cell(5, 5);
283      Cell knight2 = new Cell(2, 5);
284      Cell knight3 = new Cell(5, 2);
285      Cell knight4 = new Cell(2, 2);
286      getBoard().addPiece(new Knight(getBoard(), knight1, PlayerColor.WHITE));
287      pawnAroundPos(knight1, PlayerColor.BLACK);
288      getBoard().addPiece(new Knight(getBoard(), knight2, PlayerColor.WHITE));
289      pawnAroundPos(knight2, PlayerColor.WHITE);
290      getBoard().addPiece(new Knight(getBoard(), knight3, PlayerColor.BLACK));
291      pawnAroundPos(knight3, PlayerColor.WHITE);
292      getBoard().addPiece(new Knight(getBoard(), knight4, PlayerColor.BLACK));
293      pawnAroundPos(knight4, PlayerColor.BLACK);
294  }
295
296  /**
297   * Permet de rapidement position des pièces afin de tester le fonctionnement des
298   * Tours.
299   */
300  private void testRook() {
301      Cell rookPos1 = new Cell(5, 4);
302      Cell rookPos2 = new Cell(2, 4);
303      getBoard().addPiece(new Rook(getBoard(), new Cell(0, 0), PlayerColor.WHITE));
304      getBoard().addPiece(new Rook(getBoard(), new Cell(7, 7), PlayerColor.BLACK));
305      getBoard().addPiece(new Rook(getBoard(), rookPos1, PlayerColor.WHITE));
306      getBoard().addPiece(new Rook(getBoard(), rookPos2, PlayerColor.WHITE));
307      pawnAroundPos(rookPos1, PlayerColor.WHITE);
308      pawnAroundPos(rookPos2, PlayerColor.BLACK);
309  }
310
311  /**
312   * Permet de rapidement position des pièces afin de tester le fonctionnement des
313   * Fous.
314   */
315  private void testBishop() {
316      Cell bishopPos1 = new Cell(5, 4);
317      Cell bishopPos2 = new Cell(2, 4);
318      getBoard().addPiece(new Bishop(getBoard(), bishopPos1, PlayerColor.WHITE));
319      getBoard().addPiece(new Bishop(getBoard(), bishopPos2, PlayerColor.WHITE));
320      pawnAroundPos(bishopPos1, PlayerColor.WHITE);
321      pawnAroundPos(bishopPos2, PlayerColor.BLACK);
322  }
323
324  /**
325   * Permet de rapidement position des pièces afin de tester le fonctionnement des
326   * Pions.
327   */
328  private void testPawn() {
329      for (int i = 0; i < 7; i++) {
330          getBoard().addPiece(new Pawn(getBoard(), new Cell(i, 1),
331              PlayerColor.WHITE));
332          getBoard().addPiece(new Knight(getBoard(), new Cell(i, 2), (i % 2 == 0) ?
333              PlayerColor.WHITE : PlayerColor.BLACK));
334      }
335      getBoard().addPiece(new Pawn(getBoard(), new Cell(7, 1), PlayerColor.BLACK));
336      getBoard().addPiece(new Pawn(getBoard(), new Cell(4, 6), PlayerColor.BLACK));
337      getBoard().addPiece(new Pawn(getBoard(), new Cell(3, 4), PlayerColor.WHITE));
338  }
339
340
341
342
343
344
345

```

```

346     /**
347      * Permet de rapidement position des pièces afin de tester le fonctionnement des
348      * Reines.
349      */
350     private void testQueen() {
351         Cell queenPos = new Cell(5, 4);
352         Cell queenPos2 = new Cell(2, 4);
353         getBoard().addPiece(new Queen(getBoard(), queenPos, PlayerColor.WHITE));
354         getBoard().addPiece(new Queen(getBoard(), queenPos2, PlayerColor.WHITE));
355         pawnAroundPos(queenPos, PlayerColor.BLACK);
356         pawnAroundPos(queenPos2, PlayerColor.WHITE);
357     }
358
359     /**
360      * Permet de rapidement position des pièces afin de tester le fonctionnement des
361      * Rois.
362      */
363     private void testKing() {
364         Cell kingPos = new Cell(4, 4);
365         getBoard().addPiece(new King(getBoard(), kingPos, PlayerColor.WHITE));
366         pawnAroundPos(kingPos, PlayerColor.BLACK);
367     }
368
369     /**
370      * Permet de rapidement position des pièces afin de tester le fonctionnement
371      * du roque.
372      */
373     private void testCastle() {
374         getBoard().addPiece(new King(getBoard(), new Cell(4, 0), PlayerColor.WHITE));
375         getBoard().addPiece(new Rook(getBoard(), new Cell(7, 0), PlayerColor.WHITE));
376         getBoard().addPiece(new Rook(getBoard(), new Cell(0, 0), PlayerColor.WHITE));
377         getBoard().addPiece(new Queen(getBoard(), new Cell(3, 7), PlayerColor.BLACK));
378     }
379
380     /**
381      * Permet de rapidement position des pièces afin de tester le fonctionnement
382      * de la mise en échec.
383      */
384     private void testCheck() {
385         getBoard().addPiece(new King(getBoard(), new Cell(3, 5), PlayerColor.BLACK));
386         getBoard().addPiece(new Rook(getBoard(), new Cell(4, 3), PlayerColor.WHITE));
387         getBoard().addPiece(new Queen(getBoard(), new Cell(5, 3), PlayerColor.WHITE));
388         getBoard().addPiece(new Bishop(getBoard(), new Cell(6, 3),
389             PlayerColor.WHITE));
390         getBoard().addPiece(new Knight(getBoard(), new Cell(7, 3),
391             PlayerColor.WHITE));
392         getBoard().addPiece(new King(getBoard(), new Cell(1, 3), PlayerColor.WHITE));
393         getBoard().addPiece(new Pawn(getBoard(), new Cell(2, 3), PlayerColor.WHITE));
394     }
395 }
396
397 // -----
398
399 package engine;
400
401 import chess.PieceType;
402 import chess.PlayerColor;
403 import engine.pieces.*;
404 import engine.utils.Cell;
405
406 import java.util.ArrayList;
407 import java.util.List;
408 import java.util.Objects;
409
410
411
412
413
414

```

```

415  /**
416   * Classe modélisant un plateau virtuel du jeu d'échecs.
417   * Elle s'occupe notamment de stocker et modifier les positions des différentes
418   * pièces.
419   *
420   * @author Jonathan Friedli
421   * @author Valentin Kaelin
422   */
423  public class Board {
424      public interface PieceListener {
425          void action(Piece piece, Cell cell);
426      }
427
428      public interface PromotionListener {
429          void action(Piece piece);
430      }
431
432      public static final int BOARD_SIZE = 8;
433      private int turn;
434      private final Piece[][] pieces;
435      private final List<King> kings;
436      private Piece lastPiecePlayed;
437
438      private PieceListener onAddPiece;
439      private PieceListener onRemovePiece;
440      private PromotionListener onPromotion;
441
442      /**
443       * Constructeur de base initialisant les différentes structures
444       */
445      public Board() {
446          pieces = new Piece[BOARD_SIZE][BOARD_SIZE];
447          kings = new ArrayList<>();
448      }
449
450      /**
451       * Remet le plateau à son état initial
452       */
453      public void resetBoard() {
454          // On vide le plateau pour éviter de recréer un tableau
455          for (int i = 0; i < BOARD_SIZE; i++)
456              for (int j = 0; j < BOARD_SIZE; j++)
457                  removePiece(new Cell(i, j));
458
459          kings.clear();
460          lastPiecePlayed = null;
461          turn = 0;
462      }
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483

```

```

484  /**
485   * Remplit le tableau avec la position habituelle des différentes pièces
486   * Commence par les pièces blanches puis les noires
487   */
488  public void fillBoard() {
489      PlayerColor color = PlayerColor.WHITE;
490      int line = 0, pawnLine = 1;
491      for (int i = 0; i < 2; i++) {
492          addPiece(new Rook(this, new Cell(0, line), color));
493          addPiece(new Knight(this, new Cell(1, line), color));
494          addPiece(new Bishop(this, new Cell(2, line), color));
495          addPiece(new Queen(this, new Cell(3, line), color));
496          addPiece(new King(this, new Cell(4, line), color));
497          addPiece(new Bishop(this, new Cell(5, line), color));
498          addPiece(new Knight(this, new Cell(6, line), color));
499          addPiece(new Rook(this, new Cell(7, line), color));
500
501          // Pions
502          for (int xPawn = 0; xPawn < BOARD_SIZE; xPawn++)
503              addPiece(new Pawn(this, new Cell(xPawn, pawnLine), color));
504
505          color = PlayerColor.BLACK;
506          line = 7;
507          pawnLine = 6;
508      }
509  }
510
511  /**
512   * Vérifie que les coordonnées de la case sont valides
513   *
514   * @param cell : case à vérifier
515   * @throws RuntimeException si la case est invalide
516   */
517  public void checkCoordsOnBoard(Cell cell) {
518      if (cell == null || cell.getX() >= BOARD_SIZE || cell.getX() < 0 ||
519          cell.getY() >= BOARD_SIZE || cell.getY() < 0)
520          throw new RuntimeException("Coordonnées de la pièce invalides.");
521  }
522
523  /**
524   * @return le tour actuel
525   */
526  public int getTurn() {
527      return turn;
528  }
529
530  /**
531   * @return la dernière pièce jouée
532   */
533  public Piece getLastPiecePlayed() {
534      return lastPiecePlayed;
535  }
536
537  /**
538   * @param cell : case souhaitée
539   * @return la pièce à la case souhaitée ou null
540   * @throws RuntimeException si la case est invalide
541   */
542  public Piece getPiece(Cell cell) {
543      checkCoordsOnBoard(cell);
544      return pieces[cell.getX()][cell.getY()];
545  }
546
547
548
549
550
551
552

```



```

553     /**
554     * Ajoute la pièce à la case souhaitée
555     *
556     * @param piece : pièce à ajouter
557     * @param cell : case souhaitée
558     * @throws RuntimeException si la case est invalide
559     */
560     public void setPiece(Piece piece, Cell cell) {
561         Objects.requireNonNull(piece, "Pièce invalide");
562         checkCoordsOnBoard(cell);
563
564         pieces[cell.getX()][cell.getY()] = piece;
565         piece.setCell(cell);
566
567         if (piece.getType() == PieceType.KING)
568             kings.add((King) piece);
569
570         if (onAddPiece != null)
571             onAddPiece.action(piece, cell);
572     }
573
574     /**
575     * Petite fonction helper permettant d'ajouter une pièce à sa case actuelle
576     *
577     * @param piece : pièce à ajouter
578     * @throws RuntimeException si la case est invalide
579     */
580     public void addPiece(Piece piece) {
581         setPiece(piece, piece.getCell());
582     }
583
584     /**
585     * Supprime une pièce du tableau
586     *
587     * @param cell : case de la pièce à supprimer
588     * @throws RuntimeException si la case est invalide
589     */
590     public void removePiece(Cell cell) {
591         Piece piece = getPiece(cell);
592         pieces[cell.getX()][cell.getY()] = null;
593
594         if (piece != null) {
595             if (piece.getType() == PieceType.KING)
596                 kings.remove((King) piece);
597
598             if (onRemovePiece != null)
599                 onRemovePiece.action(piece, cell);
600         }
601     }
602
603     /**
604     * @return le joueur à qui c'est le tour de jouer
605     */
606     public PlayerColor currentPlayer() {
607         return turn % 2 == 0 ? PlayerColor.WHITE : PlayerColor.BLACK;
608     }
609
610
611
612
613
614
615
616
617
618
619
620
621

```

```

622  /**
623   * Vérifie que le déplacement d'une pièce peut se faire. Si c'est le cas,
624   * il est réalisé.
625   *
626   * @param from : case de départ
627   * @param to   : case d'arrivée
628   * @return true si le mouvement a pu être fait, false sinon
629   */
630  public boolean move(Cell from, Cell to) {
631      Piece p;
632      try {
633          p = getPiece(from);
634          checkCoordsOnBoard(to);
635      } catch (RuntimeException e) {
636          return false;
637      }
638
639      if (p == null || p.getColor() != currentPlayer())
640          return false;
641
642      if (p.checkMove(to) && p.applyMove(to)) {
643          postUpdate(p);
644          return true;
645      }
646
647      return false;
648  }
649
650  /**
651   * Applique le mouvement d'une pièce à une destination
652   *
653   * @param piece : pièce à déplacer
654   * @param to    : case d'arrivée
655   */
656  public void applyMove(Piece piece, Cell to) {
657      Objects.requireNonNull(piece, "Pièce invalide");
658      removePiece(piece.getCell());
659      removePiece(to);
660      setPiece(piece, to);
661  }
662
663  /**
664   * Vérifie si une pièce est actuellement menacée/attaquée
665   *
666   * @param color : couleur de la pièce à vérifier
667   * @param cell  : case de la pièce à vérifier
668   * @return true si la pièce est attaquée, false sinon
669   */
670  public boolean isAttacked(PlayerColor color, Cell cell) {
671      Objects.requireNonNull(color, "Couleur invalide");
672      Objects.requireNonNull(cell, "Case invalide");
673
674      for (Piece[] row : pieces)
675          for (Piece piece : row)
676              if (piece != null && piece.getColor() != color && piece.checkMove(cell))
677                  return true;
678
679      return false;
680  }
681
682
683
684
685
686
687
688
689
690

```

```

691  /**
692   * Vérifie si un joueur est actuellement en échec
693   *
694   * @param color : la couleur du joueur
695   * @return true si le joueur est en échec, false sinon
696   */
697  public boolean isCheck(PlayerColor color) {
698      Objects.requireNonNull(color, "Couleur invalide");
699
700      King king = kings.stream()
701          .filter(k -> k.getColor() == color)
702          .findAny()
703          .orElse(null);
704
705      if (king == null)
706          return false;
707
708      return isAttacked(color, king.getCell());
709  }
710
711  /**
712   * Applique les changements nécessaires à la fin d'un tour
713   *
714   * @param piece : pièce jouée
715   */
716  public void postUpdate(Piece piece) {
717      Objects.requireNonNull(piece, "Pièce invalide");
718      lastPiecePlayed = piece;
719      piece.postUpdate();
720      turn++;
721  }
722
723  /**
724   * Définit le listener appelé lors de l'ajout d'une pièce
725   *
726   * @param onAddPiece : listener à exécuter
727   */
728  public void setAddPieceListener(PieceListener onAddPiece) {
729      Objects.requireNonNull(onAddPiece, "Listener invalide");
730      this.onAddPiece = onAddPiece;
731  }
732
733  /**
734   * Définit le listener appelé lors de la suppression d'une pièce
735   *
736   * @param onRemovePiece : listener à exécuter
737   */
738  public void setRemovePieceListener(PieceListener onRemovePiece) {
739      Objects.requireNonNull(onRemovePiece, "Listener invalide");
740      this.onRemovePiece = onRemovePiece;
741  }
742
743  /**
744   * Définit le listener appelé lors de la promotion d'une pièce
745   *
746   * @param onPromotion : listener à exécuter
747   */
748  public void setPromotionListener(PromotionListener onPromotion) {
749      Objects.requireNonNull(onPromotion, "Listener invalide");
750      this.onPromotion = onPromotion;
751  }
752
753  /**
754   * @return le listener appelé lors d'une promotion
755   */
756  public PromotionListener getOnPromotion() {
757      return onPromotion;
758  }
759  }

```

```

760
761 // -----
762
763 package engine.utils;
764
765 import java.util.Objects;
766
767 /**
768  * Classe représentant une case de l'échiquier
769  *
770  * @author Jonathan Friedli
771  * @author Valentin Kaelin
772  */
773 public class Cell {
774     private final int x;
775     private final int y;
776
777     /**
778      * @param x : coordonnée x de la case
779      * @param y : coordonnée y de la case
780      */
781     public Cell(int x, int y) {
782         this.x = x;
783         this.y = y;
784     }
785
786     /**
787      * @return la coordonnée X de la case
788      */
789     public int getX() {
790         return x;
791     }
792
793     /**
794      * @return la coordonnée Y de la case
795      */
796     public int getY() {
797         return y;
798     }
799
800     /**
801      * Additionne une seconde case
802      *
803      * @param cell : la case à ajouter
804      * @return le résultat de l'addition via une nouvelle case
805      * @throws RuntimeException si la case à additionner est invalide
806      */
807     public Cell add(Cell cell) {
808         if (cell == null)
809             throw new RuntimeException("Addition d'une case invalide");
810
811         return new Cell(x + cell.x, y + cell.y);
812     }
813
814     /**
815      * Soustrait une seconde case
816      *
817      * @param cell : la case à soustraire
818      * @return le résultat de la soustraction via une nouvelle case
819      * @throws RuntimeException si la case à soustraire est invalide
820      */
821     public Cell subtract(Cell cell) {
822         if (cell == null)
823             throw new RuntimeException("Soustraction d'une case invalide");
824
825         return new Cell(x - cell.x, y - cell.y);
826     }
827
828

```

```

829     /**
830     * Multiplie la case par un scalaire
831     *
832     * @param n : scalaire
833     * @return le résultat de la multiplication via une nouvelle case
834     */
835     public Cell multiply(int n) {
836         return new Cell(n * x, n * y);
837     }
838
839     /**
840     * Vérifie qu'une case peut être atteinte depuis une autre
841     *
842     * @param cell : case de potentielle arrivée
843     * @return true si la case est atteignable, false sinon
844     */
845     public boolean reachable(Cell cell) {
846         return cell != null && x * cell.y == y * cell.x;
847     }
848
849     /**
850     * Vérifie que deux cases ont les mêmes signes sur leurs deux coordonnées
851     *
852     * @param cell : la seconde case
853     * @return true si les signes sont les mêmes, false sinon
854     */
855     public boolean sameDirection(Cell cell) {
856         return cell != null && (x < 0 == cell.getX() < 0) && (y < 0 == cell.getY() < 0);
857     }
858
859     /**
860     * Retourne la distance jusqu'à une case.
861     * Ne vérifie pas si la case est accessible.
862     *
863     * @param to : case d'arrivée
864     * @return la distance entre les deux cases
865     * @throws RuntimeException si la case d'arrivée est invalide
866     */
867     public int getDistance(Cell to) {
868         Objects.requireNonNull(to, "Case invalide");
869         Cell fromTo = to.subtract(this);
870         return Math.max(Math.abs(fromTo.getX()), Math.abs(fromTo.getY()));
871     }
872
873     @Override
874     public int hashCode() {
875         return Objects.hash(x, y);
876     }
877
878     @Override
879     public boolean equals(Object obj) {
880         return getClass() == obj.getClass() &&
881             this.x == ((Cell) obj).x &&
882             this.y == ((Cell) obj).y;
883     }
884 }
885
886 // -----
887
888
889
890
891
892
893
894
895
896
897

```

```

898 package engine.utils;
899
900 /**
901  * Énumération permettant de modéliser des déplacements dans une certaine direction
902  * Les directions gauches et droites ne sont pas utilisées, mais sont implémentée
903  * dans un souci d'harmonisation.
904  *
905  * @author Jonathan Friedli
906  * @author Valentin Kaelin
907  */
908 public enum Direction {
909     UP(0, 1), DOWN(0, -1), LEFT(-1, 0), RIGHT(1, 0);
910     private final Cell value;
911
912     private Direction(int x, int y) {
913         this.value = new Cell(x, y);
914     }
915
916     /**
917      * @return la valeur de la direction sous forme d'une case
918      */
919     public Cell getValue() {
920         return value;
921     }
922
923     /**
924      * @return la valeur de la direction sous forme d'un nombre
925      */
926     public int intValue() {
927         return value.getX() == 0 ? value.getY() : value.getX();
928     }
929 }
930
931 // -----
932
933 package engine.pieces;
934
935 import chess.ChessView;
936 import chess.PieceType;
937 import chess.PlayerColor;
938 import engine.Board;
939 import engine.moves.Move;
940 import engine.utils.Cell;
941
942 import java.util.ArrayList;
943 import java.util.List;
944
945 /**
946  * Classe abstraite permettant de définir la base de toutes les pièces du jeu
947  * d'échecs.
948  *
949  * @author Jonathan Friedli
950  * @author Valentin Kaelin
951  */
952 public abstract class Piece implements ChessView.UserChoice {
953     private final Board board;
954     private final PlayerColor color;
955     private Cell cell;
956     protected List<Move> moves;
957
958
959
960
961
962
963
964
965
966

```

```

967     /**
968     * Crée une nouvelle pièce
969     *
970     * @param board : plateau de la pièce
971     * @param cell : case de la pièce
972     * @param color : couleur de la pièce
973     * @throws RuntimeException s'il manque un paramètre
974     */
975     public Piece(Board board, Cell cell, PlayerColor color) {
976         if (board == null || cell == null || color == null)
977             throw new RuntimeException("Construction de la pièce invalide");
978
979         this.board = board;
980         this.cell = cell;
981         this.color = color;
982         moves = new ArrayList<>();
983     }
984
985     /**
986     * @return le type de la pièce
987     */
988     public abstract PieceType getType();
989
990     /**
991     * @return le texte en français représentant la pièce
992     */
993     public abstract String textValue();
994
995     /**
996     * @return le plateau de la pièce
997     */
998     public Board getBoard() {
999         return board;
1000     }
1001
1002     /**
1003     * @return la couleur de la pièce
1004     */
1005     public PlayerColor getColor() {
1006         return color;
1007     }
1008
1009     @Override
1010     public String toString() {
1011         return textValue();
1012     }
1013
1014     /**
1015     * @return la case de la pièce
1016     */
1017     public Cell getCell() {
1018         return cell;
1019     }
1020
1021     /**
1022     * Change la case de la pièce
1023     *
1024     * @param cell : nouvelle case
1025     * @throws RuntimeException si le case est inexistante
1026     */
1027     public void setCell(Cell cell) {
1028         if (cell == null)
1029             throw new RuntimeException("Case de la pièce invalide.");
1030
1031         this.cell = cell;
1032     }
1033
1034
1035

```

```

1036  /**
1037  * Vérifie qu'un mouvement peut-être réalisé par la pièce
1038  *
1039  * @param to : case de destination souhaitée
1040  * @return true si le mouvement peut être fait, false sinon
1041  */
1042  public boolean checkMove(Cell to) {
1043      // Si la case de destination est occupée par une pièce de même couleur
1044      if (to == null || (board.getPiece(to) != null &&
1045          board.getPiece(to).getColor() == color))
1046          return false;
1047
1048      for (Move move : moves) {
1049          if (move.canMove(cell, to))
1050              return true;
1051      }
1052
1053      return false;
1054  }
1055
1056  /**
1057  * Vérifie qu'un mouvement (légal) peut être appliqué
1058  *
1059  * @param to : case de destination souhaitée
1060  * @return true s'il peut être appliqué, false s'il met le roi du joueur en échec
1061  */
1062  public boolean applyMove(Cell to) {
1063      Cell oldCell = getCell();
1064      Piece eaten = board.getPiece(to);
1065
1066      board.applyMove(this, to);
1067
1068      // En échec : on annule le move
1069      if (board.isCheck(color)) {
1070          board.applyMove(this, oldCell);
1071          if (eaten != null)
1072              board.setPiece(eaten, to);
1073          return false;
1074      }
1075
1076      return true;
1077  }
1078
1079  /**
1080  * Méthode à implémenter dans les pièces devant réaliser des actions après un
1081  * tour.
1082  */
1083  public void postUpdate() {
1084  }
1085  }
1086
1087  // -----

```



```

1105 package engine.pieces;
1106
1107 import chess.PlayerColor;
1108 import engine.Board;
1109 import engine.utils.Cell;
1110
1111 /**
1112  * Classe abstraite permettant d'ajouter la gestion de premier coup spécifique à
1113  * certaines pièces.
1114  *
1115  * @author Jonathan Friedli
1116  * @author Valentin Kaelin
1117  */
1118 public abstract class FirstMoveSpecificPiece extends Piece {
1119     private boolean hasMoved;
1120
1121     public FirstMoveSpecificPiece(Board board, Cell cell, PlayerColor color) {
1122         super(board, cell, color);
1123         hasMoved = false;
1124     }
1125
1126     /**
1127      * @return true si la pièce a déjà bougé, false sinon
1128      */
1129     public boolean hasMoved() {
1130         return hasMoved;
1131     }
1132
1133     /**
1134      * Indique à la fin du tour que la pièce a déjà bougé
1135      */
1136     public void postUpdate() {
1137         hasMoved = true;
1138     }
1139 }
1140
1141 // -----
1142
1143 package engine.pieces;
1144
1145 import chess.PieceType;
1146 import chess.PlayerColor;
1147 import engine.Board;
1148 import engine.moves.LinearMove;
1149 import engine.utils.Cell;
1150
1151 /**
1152  * Classe représentant un fou
1153  *
1154  * @author Jonathan Friedli
1155  * @author Valentin Kaelin
1156  */
1157 public class Bishop extends Piece {
1158     public Bishop(Board board, Cell cell, PlayerColor color) {
1159         super(board, cell, color);
1160         moves.add(new LinearMove(this, new Cell(1, 1)));
1161         moves.add(new LinearMove(this, new Cell(1, -1)));
1162     }
1163
1164     @Override
1165     public PieceType getType() {
1166         return PieceType.BISHOP;
1167     }
1168
1169     @Override
1170     public String textValue() {
1171         return "Fou";
1172     }
1173 }

```

```
1174
1175 // -----
1176
1177 package engine.pieces;
1178
1179 import chess.PieceType;
1180 import chess.PlayerColor;
1181 import engine.Board;
1182 import engine.moves.LinearMove;
1183 import engine.utils.Cell;
1184
1185 /**
1186  * Classe représentant un roi
1187  *
1188  * @author Jonathan Friedli
1189  * @author Valentin Kaelin
1190  */
1191 public class King extends FirstMoveSpecificPiece {
1192     private static final int CASTLE_DISTANCE = 2;
1193
1194     public King(Board board, Cell cell, PlayerColor color) {
1195         super(board, cell, color);
1196         moves.add(new LinearMove(this, new Cell(0, 1), 1));
1197         moves.add(new LinearMove(this, new Cell(1, 0), 1));
1198         moves.add(new LinearMove(this, new Cell(1, 1), 1));
1199         moves.add(new LinearMove(this, new Cell(1, -1), 1));
1200     }
1201
1202     @Override
1203     public PieceType getType() {
1204         return PieceType.KING;
1205     }
1206
1207     @Override
1208     public String textValue() {
1209         return "Roi";
1210     }
1211
1212     @Override
1213     public boolean checkMove(Cell to) {
1214         return super.checkMove(to) || castle(to);
1215     }
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
```

```

1243 /**
1244  * Vérifie si le déplacement est un roque légal
1245  *
1246  * @param to : case de destination
1247  * @return true si le roque a bien été effectué, false sinon
1248  */
1249 private boolean castle(Cell to) {
1250     if (to == null)
1251         return false;
1252     int deltaY = to.getY() - getCell().getY();
1253     int deltaX = to.getX() - getCell().getX();
1254     if (hasMoved() || Math.abs(deltaX) != CASTLE_DISTANCE || deltaY != 0)
1255         return false;
1256
1257     boolean leftSide = deltaX < 0;
1258     Cell direction = new Cell(leftSide ? -1 : 1, 0);
1259     Cell rookCell = new Cell(leftSide ? 0 : Board.BOARD_SIZE - 1, getCell().getY());
1260     Piece rook = getBoard().getPiece(rookCell);
1261     Cell rookDestination = getCell().add(direction);
1262
1263     // Vérification de la tour et que le chemin est libre
1264     if (rook == null || rook.getType() != PieceType.ROOK ||
1265         ((Rook) rook).hasMoved() || !rook.checkMove(rookDestination))
1266         return false;
1267
1268     // Vérification que le chemin ne met pas le roi en échec
1269     Cell initialPosition = getCell();
1270     for (int i = 0; i <= CASTLE_DISTANCE; i++) {
1271         Cell position = getCell().add(direction.multiply(i));
1272         getBoard().setPiece(this, position);
1273
1274         boolean isAttacked = getBoard().isAttacked(getColor(), position);
1275         getBoard().removePiece(position);
1276         if (isAttacked) {
1277             getBoard().setPiece(this, initialPosition);
1278             return false;
1279         }
1280     }
1281
1282     // Roque appliqué
1283     getBoard().applyMove(rook, rookDestination);
1284     rook.postUpdate();
1285
1286     return true;
1287 }
1288 }
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311

```

```

1312 // -----
1313
1314 package engine.pieces;
1315
1316 import chess.PieceType;
1317 import chess.PlayerColor;
1318 import engine.Board;
1319 import engine.moves.LinearMove;
1320 import engine.utils.Cell;
1321
1322 /**
1323  * Classe représentant un cavalier
1324  *
1325  * @author Jonathan Friedli
1326  * @author Valentin Kaelin
1327  */
1328 public class Knight extends Piece {
1329     public Knight(Board board, Cell cell, PlayerColor color) {
1330         super(board, cell, color);
1331         moves.add(new LinearMove(this, new Cell(1, 2), 2, true));
1332         moves.add(new LinearMove(this, new Cell(1, -2), 2, true));
1333         moves.add(new LinearMove(this, new Cell(2, 1), 2, true));
1334         moves.add(new LinearMove(this, new Cell(2, -1), 2, true));
1335     }
1336
1337     @Override
1338     public PieceType getType() {
1339         return PieceType.KNIGHT;
1340     }
1341
1342     @Override
1343     public String textValue() {
1344         return "Cavalier";
1345     }
1346 }
1347
1348 // -----
1349
1350 package engine.pieces;
1351
1352 import chess.PieceType;
1353 import chess.PlayerColor;
1354 import engine.Board;
1355 import engine.utils.Direction;
1356 import engine.moves.OneDirectionMove;
1357 import engine.utils.Cell;
1358
1359 /**
1360  * Classe représentant un pion
1361  *
1362  * @author Jonathan Friedli
1363  * @author Valentin Kaelin
1364  */
1365 public class Pawn extends FirstMoveSpecificPiece {
1366     private final Direction direction;
1367     private int doubleMoveTurn;
1368
1369     public Pawn(Board board, Cell cell, PlayerColor color) {
1370         super(board, cell, color);
1371         direction = color == PlayerColor.WHITE ? Direction.UP : Direction.DOWN;
1372         moves.add(new OneDirectionMove(this, 1, direction));
1373         moves.add(new OneDirectionMove(this, 2, direction, true));
1374     }
1375
1376     @Override
1377     public PieceType getType() {
1378         return PieceType.PAWN;
1379     }
1380

```

```

1381     @Override
1382     public String textValue() {
1383         return "Pion";
1384     }
1385
1386     @Override
1387     public boolean checkMove(Cell to) {
1388         if (super.checkMove(to)) {
1389             // On stocke le tour actuel si le pion s'est déplacé de deux cases
1390             if (Math.abs(to.getY() - getCell().getY()) == 2)
1391                 doubleMoveTurn = getBoard().getTurn();
1392             return true;
1393         }
1394
1395         if (to == null)
1396             return false;
1397
1398         int deltaX = to.getX() - getCell().getX();
1399         int deltaY = to.getY() - getCell().getY();
1400
1401         // Manger en diagonale
1402         if (Math.abs(deltaX) == 1 && deltaY == direction.intValue()) {
1403             if (getBoard().getPiece(to) != null)
1404                 return true;
1405         }
1406
1407         // En passant
1408         return enPassant(new Cell(to.getX(), getCell().getY()));
1409     }
1410
1411     @Override
1412     public boolean applyMove(Cell to) {
1413         if (to == null)
1414             return false;
1415
1416         Cell oldCell = getCell();
1417         Piece piece = getBoard().getLastPiecePlayed();
1418
1419         // Vérification de la mise en échec du en-passant
1420         if (enPassant(new Cell(to.getX(), oldCell.getY()))) {
1421             getBoard().applyMove(this, to);
1422             getBoard().removePiece(piece.getCell());
1423
1424             // En échec : on annule les moves
1425             if (getBoard().isCheck(getColor())) {
1426                 getBoard().applyMove(this, oldCell);
1427                 getBoard().setPiece(piece, piece.getCell());
1428                 return false;
1429             }
1430             return true;
1431         }
1432
1433         return super.applyMove(to);
1434     }
1435
1436     @Override
1437     public void postUpdate() {
1438         super.postUpdate();
1439
1440         // Gestion de la promotion
1441         if (canBePromoted() && getBoard().getOnPromotion() != null)
1442             getBoard().getOnPromotion().action(this);
1443     }
1444
1445
1446
1447
1448
1449

```

```

1450     /**
1451     * @return true si le pion peut être promu, false sinon
1452     */
1453     public boolean canBePromoted() {
1454         return direction == Direction.UP ?
1455             getCell().getY() == Board.BOARD_SIZE - 1 :
1456             getCell().getY() == 0;
1457     }
1458
1459     /**
1460     * Vérifie si le move en-passant peut être réalisé
1461     *
1462     * @param cell : case de destination
1463     * @return true si le move est légal, false sinon
1464     */
1465     public boolean enPassant(Cell cell) {
1466         Piece piece = getBoard().getLastPiecePlayed();
1467         int lastTurn = getBoard().getTurn() - 1;
1468         return piece != null && piece != this && piece.getColor() != getColor() &&
1469             piece.getClass() == Pawn.class &&
1470             ((Pawn) piece).doubleMoveTurn == lastTurn &&
1471             piece.getCell().equals(cell);
1472     }
1473 }
1474
1475 // -----
1476
1477 package engine.pieces;
1478
1479 import chess.PieceType;
1480 import chess.PlayerColor;
1481 import engine.Board;
1482 import engine.moves.LinearMove;
1483 import engine.utils.Cell;
1484
1485 /**
1486  * Classe représentant une reine
1487  *
1488  * @author Jonathan Friedli
1489  * @author Valentin Kaelin
1490  */
1491 public class Queen extends Piece {
1492     public Queen(Board board, Cell cell, PlayerColor color) {
1493         super(board, cell, color);
1494         moves.add(new LinearMove(this, new Cell(0, 1)));
1495         moves.add(new LinearMove(this, new Cell(1, 0)));
1496         moves.add(new LinearMove(this, new Cell(1, 1)));
1497         moves.add(new LinearMove(this, new Cell(1, -1)));
1498     }
1499
1500     @Override
1501     public PieceType getType() {
1502         return PieceType.QUEEN;
1503     }
1504
1505     @Override
1506     public String textValue() {
1507         return "Reine";
1508     }
1509 }
1510
1511 // -----
1512
1513
1514
1515
1516
1517
1518

```

```

1519 package engine.pieces;
1520
1521 import chess.PieceType;
1522 import chess.PlayerColor;
1523 import engine.Board;
1524 import engine.moves.LinearMove;
1525 import engine.utils.Cell;
1526
1527 /**
1528  * Classe représentant une tour
1529  *
1530  * @author Jonathan Friedli
1531  * @author Valentin Kaelin
1532  */
1533 public class Rook extends FirstMoveSpecificPiece {
1534     public Rook(Board board, Cell cell, PlayerColor color) {
1535         super(board, cell, color);
1536         moves.add(new LinearMove(this, new Cell(0, 1)));
1537         moves.add(new LinearMove(this, new Cell(1, 0)));
1538     }
1539
1540     @Override
1541     public PieceType getType() {
1542         return PieceType.ROOK;
1543     }
1544
1545     @Override
1546     public String textValue() {
1547         return "Tour";
1548     }
1549 }
1550
1551 // -----
1552
1553 package engine.moves;
1554
1555 import engine.Board;
1556 import engine.pieces.Piece;
1557 import engine.utils.Cell;
1558
1559 /**
1560  * Classe abstraite modélisant la base des divers déplacements.
1561  * Le mouvement peut être limité à un nombre de cases.
1562  *
1563  * @author Jonathan Friedli
1564  * @author Valentin Kaelin
1565  */
1566 public abstract class Move {
1567     private final Piece piece;
1568     private final int maxDistance;
1569
1570     /**
1571      * Crée un déplacement
1572      *
1573      * @param piece : pièce concernée
1574      * @param maxDistance : potentielle distance maximale
1575      * @throws RuntimeException si les arguments sont invalides
1576      */
1577     public Move(Piece piece, int maxDistance) {
1578         if (piece == null || maxDistance < 0)
1579             throw new RuntimeException("Création du Move invalide");
1580
1581         this.piece = piece;
1582         this.maxDistance = maxDistance;
1583     }
1584
1585
1586
1587

```

```

1588     /**
1589     * Vérifie qu'une case peut être atteinte grâce au déplacement
1590     *
1591     * @param from : case de départ
1592     * @param to   : case d'arrivée
1593     * @return true si la case est atteignable, false sinon
1594     */
1595     public abstract boolean canMove(Cell from, Cell to);
1596
1597     /**
1598     * @return la pièce du déplacement
1599     */
1600     public Piece getPiece() {
1601         return piece;
1602     }
1603
1604     /**
1605     * Helper permettant de récupérer plus facilement le plateau du déplacement
1606     *
1607     * @return le plateau de la pièce du déplacement
1608     */
1609     public Board getBoard() {
1610         return piece.getBoard();
1611     }
1612
1613     /**
1614     * @return la distance maximale du déplacement
1615     */
1616     public int getMaxDistance() {
1617         return maxDistance;
1618     }
1619 }
1620
1621 // -----
1622
1623 package engine.moves;
1624
1625 import engine.pieces.Piece;
1626 import engine.utils.Cell;
1627
1628 /**
1629 * Classe représentant un déplacement linéaire dans un plan 2D.
1630 * La gestion des collisions est potentiellement gérée.
1631 *
1632 * @author Jonathan Friedli
1633 * @author Valentin Kaelin
1634 */
1635 public class LinearMove extends Move {
1636     protected final Cell direction;
1637     private final boolean flyOver;
1638
1639     /**
1640     * Crée un déplacement linéaire
1641     *
1642     * @param piece      : pièce concernée
1643     * @param direction  : direction du déplacement
1644     * @param maxDistance : potentielle distance maximale
1645     * @param flyOver    : indique si la pièce prend en compte les collisions ou pas
1646     * @throws RuntimeException si les arguments sont invalides
1647     */
1648     public LinearMove(Piece piece, Cell direction, int maxDistance, boolean flyOver) {
1649         super(piece, maxDistance);
1650
1651         if (direction == null)
1652             throw new RuntimeException("Création du LinearMove invalide");
1653
1654         this.direction = direction;
1655         this.flyOver = flyOver;
1656     }

```



```

1657
1658 /**
1659  * Crée un déplacement linéaire
1660  *
1661  * @param piece      : pièce concernée
1662  * @param direction  : direction du déplacement
1663  * @param maxDistance : potentielle distance maximale
1664  */
1665 public LinearMove(Piece piece, Cell direction, int maxDistance) {
1666     this(piece, direction, maxDistance, false);
1667 }
1668
1669 /**
1670  * Crée un déplacement linéaire
1671  *
1672  * @param piece      : pièce concernée
1673  * @param direction  : direction du déplacement
1674  */
1675 public LinearMove(Piece piece, Cell direction) {
1676     this(piece, direction, Integer.MAX_VALUE, false);
1677 }
1678
1679
1680 @Override
1681 public boolean canMove(Cell from, Cell to) {
1682     if (from == null || to == null)
1683         return false;
1684
1685     Cell fromTo = to.subtract(from);
1686     int distance = direction.reachable(fromTo) ? from.getDistance(to) : 0;
1687     int sign = direction.sameDirection(fromTo) ? 1 : -1;
1688
1689     if (distance == 0 || distance > getMaxDistance())
1690         return false;
1691
1692     // Gestion des collisions
1693     if (!flyOver) {
1694         for (int i = 1; i < distance; ++i) {
1695             Cell position = from.add(direction.multiply(i * sign));
1696             // Si une case sur le chemin est occupée
1697             if (getBoard().getPiece(position) != null)
1698                 return false;
1699         }
1700     }
1701
1702     return true;
1703 }
1704
1705
1706 // -----
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725

```

```

1726 package engine.moves;
1727
1728 import engine.pieces.FirstMoveSpecificPiece;
1729 import engine.pieces.Piece;
1730 import engine.utils.Cell;
1731 import engine.utils.Direction;
1732
1733 /**
1734  * Classe représentant un déplacement réduit à une seule direction.
1735  * Le déplacement peut potentiellement être à usage unique.
1736  * La gestion des collisions est également gérée.
1737  *
1738  * @author Jonathan Friedli
1739  * @author Valentin Kaelin
1740  */
1741 public class OneDirectionMove extends Move {
1742     private final Direction boundToDirection;
1743     private final boolean oneTimeMove;
1744
1745     /**
1746      * Crée un déplacement à une direction
1747      *
1748      * @param piece          : pièce concernée
1749      * @param maxDistance    : potentielle distance maximale
1750      * @param boundToDirection : unique direction possible
1751      * @param oneTimeMove    : true si le déplacement est à usage unique
1752      */
1753     public OneDirectionMove(Piece piece, int maxDistance, Direction boundToDirection,
1754                             boolean oneTimeMove) {
1755         super(piece, maxDistance);
1756
1757         if (boundToDirection == null)
1758             throw new RuntimeException("Création du OneDirectionMove invalide");
1759
1760         this.boundToDirection = boundToDirection;
1761         this.oneTimeMove = oneTimeMove;
1762     }
1763
1764     /**
1765      * Crée un déplacement à une direction
1766      *
1767      * @param piece          : pièce concernée
1768      * @param maxDistance    : potentielle distance maximale
1769      * @param boundToDirection : unique direction possible
1770      */
1771     public OneDirectionMove(Piece piece, int maxDistance, Direction boundToDirection) {
1772         this(piece, maxDistance, boundToDirection, false);
1773     }
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794

```

```

1795     @Override
1796     public boolean canMove(Cell from, Cell to) {
1797         if (from == null || to == null)
1798             return false;
1799
1800         // Vérification si le déplacement est à usage unique
1801         if (oneTimeMove && (!(getPiece() instanceof FirstMoveSpecificPiece) ||
1802             ((FirstMoveSpecificPiece) getPiece()).hasMoved()))
1803             return false;
1804
1805         Cell calculatedTo = from.add(
1806             boundToDirection.getValue().multiply(getMaxDistance())
1807         );
1808
1809         for (int i = 1; i < getMaxDistance(); ++i) {
1810             Cell position = from.add(boundToDirection.getValue().multiply(i));
1811             // Si une case sur le chemin est occupée
1812             if (getBoard().getPiece(position) != null)
1813                 return false;
1814         }
1815
1816         return getBoard().getPiece(to) == null && to.equals(calculatedTo);
1817     }
1818 }
1819

```