

```

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         boolean canMove = board.move(from, to);
159         updateDisplayMessage();
160
161         return canMove;
162     }
163
164     @Override
165     public void newGame() {
166         board.resetBoard();
167         board.fillBoard();
168         updateDisplayMessage();
169     }
170 }
171
172
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  private 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  private 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 o) {
880         if (this == o) return true;
881         if (o == null || getClass() != o.getClass()) return false;
882         Cell cell = (Cell) o;
883         return x == cell.x && y == cell.y;
884     }
885 }
886
887 // -----
888
889
890
891
892
893
894
895
896
897

```

```

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

```

```

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

```

```

1036
1037 /**
1038  * Vérifie qu'un mouvement peut-être réalisé par la pièce
1039  *
1040  * @param to : case de destination souhaitée
1041  * @return true si le mouvement peut être fait, false sinon
1042  */
1043 public boolean checkMove(Cell to) {
1044     // Si la case de destination est occupée par une pièce de même couleur
1045     if (to == null || (board.getPiece(to) != null &&
1046         board.getPiece(to).getColor() == color))
1047         return false;
1048
1049     for (Move move : moves) {
1050         if (move.canMove(cell, to))
1051             return true;
1052     }
1053
1054     return false;
1055 }
1056
1057 /**
1058  * Vérifie qu'un mouvement (légal) peut être appliqué
1059  *
1060  * @param to : case de destination souhaitée
1061  * @return true s'il peut être appliqué, false s'il met le roi du joueur en échec
1062  */
1063 public boolean applyMove(Cell to) {
1064     Cell oldCell = getCell();
1065     Piece eaten = board.getPiece(to);
1066
1067     board.applyMove(this, to);
1068
1069     // En échec : on annule le move
1070     if (board.isCheck(color)) {
1071         board.applyMove(this, oldCell);
1072         if (eaten != null)
1073             board.setPiece(eaten, to);
1074         return false;
1075     }
1076
1077     return true;
1078 }
1079
1080 /**
1081  * Méthode à implémenter dans les pièces devant réaliser des actions après un
1082  * tour.
1083  */
1084 public void postUpdate() {
1085 }
1086 }
1087
1088 // -----

```



```

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

```

```
1174 }
1175
1176 // -----
1177
1178 package engine.pieces;
1179
1180 import chess.PieceType;
1181 import chess.PlayerColor;
1182 import engine.Board;
1183 import engine.moves.LinearMove;
1184 import engine.utils.Cell;
1185
1186 /**
1187  * Classe représentant un roi
1188  *
1189  * @author Jonathan Friedli
1190  * @author Valentin Kaelin
1191  */
1192 public class King extends FirstMoveSpecificPiece {
1193     private static final int CASTLE_DISTANCE = 2;
1194
1195     public King(Board board, Cell cell, PlayerColor color) {
1196         super(board, cell, color);
1197         moves.add(new LinearMove(this, new Cell(0, 1), 1));
1198         moves.add(new LinearMove(this, new Cell(1, 0), 1));
1199         moves.add(new LinearMove(this, new Cell(1, 1), 1));
1200         moves.add(new LinearMove(this, new Cell(1, -1), 1));
1201     }
1202
1203     @Override
1204     public PieceType getType() {
1205         return PieceType.KING;
1206     }
1207
1208     @Override
1209     public String textValue() {
1210         return "Roi";
1211     }
1212
1213     @Override
1214     public boolean checkMove(Cell to) {
1215         return super.checkMove(to) || castle(to);
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 /**
1245  * Vérifie si le déplacement est un roque légal
1246  *
1247  * @param to : case de destination
1248  * @return true si le roque a bien été effectué, false sinon
1249  */
1250 private boolean castle(Cell to) {
1251     if (to == null)
1252         return false;
1253     int deltaY = to.getY() - getCell().getY();
1254     int deltaX = to.getX() - getCell().getX();
1255     if (hasMoved() || Math.abs(deltaX) != CASTLE_DISTANCE || deltaY != 0)
1256         return false;
1257
1258     boolean leftSide = deltaX < 0;
1259     Cell direction = new Cell(leftSide ? -1 : 1, 0);
1260     Cell rookCell = new Cell(leftSide ? 0 : Board.BOARD_SIZE - 1, getCell().getY());
1261     Piece rook = getBoard().getPiece(rookCell);
1262     Cell rookDestination = getCell().add(direction);
1263
1264     // Vérification de la tour et que le chemin est libre
1265     if (rook == null || rook.getType() != PieceType.ROOK ||
1266         ((Rook) rook).hasMoved() || !rook.checkMove(rookDestination))
1267         return false;
1268
1269     // Vérification que le chemin ne met pas le roi en échec
1270     Cell initialPosition = getCell();
1271     for (int i = 0; i <= CASTLE_DISTANCE; i++) {
1272         Cell position = initialPosition.add(direction.multiply(i));
1273         getBoard().setPiece(this, position);
1274
1275         boolean isAttacked = getBoard().isAttacked(getColor(), position);
1276         getBoard().removePiece(position);
1277         if (isAttacked) {
1278             getBoard().setPiece(this, initialPosition);
1279             return false;
1280         }
1281     }
1282
1283     // Roque appliqué
1284     getBoard().applyMove(rook, rookDestination);
1285     rook.postUpdate();
1286
1287     return true;
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
1315 package engine.pieces;
1316
1317 import chess.PieceType;
1318 import chess.PlayerColor;
1319 import engine.Board;
1320 import engine.moves.LinearMove;
1321 import engine.utils.Cell;
1322
1323 /**
1324  * Classe représentant un cavalier
1325  *
1326  * @author Jonathan Friedli
1327  * @author Valentin Kaelin
1328  */
1329 public class Knight extends Piece {
1330     public Knight(Board board, Cell cell, PlayerColor color) {
1331         super(board, cell, color);
1332         moves.add(new LinearMove(this, new Cell(1, 2), 2, true));
1333         moves.add(new LinearMove(this, new Cell(1, -2), 2, true));
1334         moves.add(new LinearMove(this, new Cell(2, 1), 2, true));
1335         moves.add(new LinearMove(this, new Cell(2, -1), 2, true));
1336     }
1337
1338     @Override
1339     public PieceType getType() {
1340         return PieceType.KNIGHT;
1341     }
1342
1343     @Override
1344     public String textValue() {
1345         return "Cavalier";
1346     }
1347 }
1348
1349 // -----
1350
1351 package engine.pieces;
1352
1353 import chess.PieceType;
1354 import chess.PlayerColor;
1355 import engine.Board;
1356 import engine.utils.Direction;
1357 import engine.moves.OneDirectionMove;
1358 import engine.utils.Cell;
1359
1360 /**
1361  * Classe représentant un pion
1362  *
1363  * @author Jonathan Friedli
1364  * @author Valentin Kaelin
1365  */
1366 public class Pawn extends FirstMoveSpecificPiece {
1367     private final Direction direction;
1368     private int doubleMoveTurn;
1369
1370     public Pawn(Board board, Cell cell, PlayerColor color) {
1371         super(board, cell, color);
1372         direction = color == PlayerColor.WHITE ? Direction.UP : Direction.DOWN;
1373         moves.add(new OneDirectionMove(this, 1, direction));
1374         moves.add(new OneDirectionMove(this, 2, direction, true));
1375     }
1376
1377     @Override
1378     public PieceType getType() {
1379         return PieceType.PAWN;
1380     }

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```
1795
1796 @Override
1797 public boolean canMove(Cell from, Cell to) {
1798     if (from == null || to == null)
1799         return false;
1800
1801     // Vérification si le déplacement est à usage unique
1802     if (oneTimeMove && (!(getPiece() instanceof FirstMoveSpecificPiece) ||
1803         ((FirstMoveSpecificPiece) getPiece()).hasMoved()))
1804         return false;
1805
1806     Cell calculatedTo = from.add(
1807         boundToDirection.getValue().multiply(getMaxDistance())
1808     );
1809
1810     for (int i = 1; i < getMaxDistance(); ++i) {
1811         Cell position = from.add(boundToDirection.getValue().multiply(i));
1812         // Si une case sur le chemin est occupée
1813         if (getBoard().getPiece(position) != null)
1814             return false;
1815     }
1816
1817     return getBoard().getPiece(to) == null && to.equals(calculatedTo);
1818 }
1819 }
1820
```