

```

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          String color = board.currentPlayer() == PlayerColor.WHITE ? "blancs" : "noirs";
97          StringBuilder msg = new StringBuilder("Aux " + color);
98
99          if (board.isCheck(board.currentPlayer())) {
100              if (board.isCheckMate(board.currentPlayer())) {
101                  msg.setLength(0);
102                  String winner = board.currentPlayer() == PlayerColor.WHITE ?
103                      "noirs" : "blancs";
104                  msg.append("CHECKMATE! Les ").append(winner).append(" ont gagnés!");
105              } else {
106                  msg.append(" CHECK!");
107              }
108          }
109
110          view.displayMessage(msg.toString());
111      }
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138

```

```

139     /**
140     * Initialise le plateau, écoute les différents événements
141     */
142     private void initBoard() {
143         board = new Board();
144
145         // Events listeners
146         board.setAddPieceListener((piece, cell) -> {
147             if (view != null)
148                 view.putPiece(piece.getType(), piece.getColor(), cell.getX(), cell.getY());
149         });
150
151         board.setRemovePieceListener((piece, cell) -> {
152             if (view != null)
153                 view.removePiece(cell.getX(), cell.getY());
154         });
155
156         board.setPromotionListener((piece) -> {
157             Cell cell = piece.getCell();
158             PlayerColor color = piece.getColor();
159             Piece[] choices = {
160                 new Queen(board, cell, color),
161                 new Knight(board, cell, color),
162                 new Rook(board, cell, color),
163                 new Bishop(board, cell, color)
164             };
165
166             Piece userChoice;
167             while ((userChoice = view.askUser("Promotion",
168                 "Choisir une pièce pour la promotion", choices)) == null) {
169             }
170             board.removePiece(cell);
171             board.setPiece(userChoice, cell);
172         });
173     }
174
175     @Override
176     public void start(ChessView view) {
177         Objects.requireNonNull(view, "View invalide");
178         this.view = view;
179         view.startView();
180         initBoard();
181         board.fillBoard();
182         updateDisplayMessage();
183     }
184
185     @Override
186     public boolean move(int fromX, int fromY, int toX, int toY) {
187         if (board == null)
188             return false;
189
190         Cell from = new Cell(fromX, fromY);
191         Cell to = new Cell(toX, toY);
192
193         boolean canMove = board.move(from, to);
194         updateDisplayMessage();
195
196         return canMove;
197     }
198
199     @Override
200     public void newGame() {
201         board.resetBoard();
202         board.fillBoard();
203         updateDisplayMessage();
204     }
205 }
206
207

```

```
208
209
210
211 // -----
212
213 package engine;
214
215 import chess.ChessView;
216 import chess.PlayerColor;
217 import engine.pieces.*;
218 import engine.utils.Cell;
219
220 /**
221  * Classe permettant de tester rapidement diverses situations initiales.
222  * Il est possible de spécifier quelle pièce ou mouvement nous souhaitons tester
223  * grâce au constructeur.
224  *
225  * @author Jonathan Friedli
226  * @author Valentin Kaelin
227  */
228 public class GameManagerTest extends GameManager {
229     enum Type {
230         CHECK, CHECKMATE, CASTLE, ROOK, BISHOP, KING, KNIGHT, PAWN, QUEEN
231     }
232
233     private final Type type;
234
235     /**
236      * Crée une instance de test
237      *
238      * @param type : pièce ou mouvement à tester
239      */
240     public GameManagerTest(Type type) {
241         this.type = type;
242     }
243
244     @Override
245     public void start(ChessView view) {
246         super.start(view);
247         newGame();
248     }
249
250     @Override
251     public void newGame() {
252         getBoard().resetBoard();
253         // Applique la situation initiale de test
254         fillBoard(type);
255     }
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
```

```

277  /**
278  * Remplit le plateau selon le test choisi
279  *
280  * @param type : pièce ou mouvement à tester
281  */
282  private void fillBoard(Type type) {
283      switch (type) {
284          case CHECK:
285              testCheck();
286              break;
287          case CHECKMATE:
288              testCheckMate();
289              break;
290          case CASTLE:
291              testCastle();
292              break;
293          case ROOK:
294              testRook();
295              break;
296          case BISHOP:
297              testBishop();
298              break;
299          case KING:
300              testKing();
301              break;
302          case KNIGHT:
303              testKnight();
304              break;
305          case PAWN:
306              testPawn();
307              break;
308          case QUEEN:
309              testQueen();
310              break;
311          default:
312              break;
313      }
314  }
315
316  /**
317  * Ajoute des pions de la couleur souhaitée autour de la position spécifiée
318  *
319  * @param pos    position que l'on souhaite entourer
320  * @param color  couleur de pions
321  */
322  private void pawnAroundPos(Cell pos, PlayerColor color) {
323      for (int i = pos.getX() - 1; i < pos.getX() + 2; i++) {
324          for (int j = pos.getY() - 1; j < pos.getY() + 2; j++) {
325              if (i == pos.getX() && j == pos.getY()) {
326                  continue;
327              }
328              getBoard().addPiece(new Pawn(getBoard(), new Cell(i, j), color));
329          }
330      }
331  }
332
333
334
335
336
337
338
339
340
341
342
343
344
345

```

```

346  /**
347   * Permet de rapidement position des pièces afin de tester le fonctionnement des
348   * Cavaliers
349   */
350  private void testKnight() {
351      Cell knight1 = new Cell(5, 5);
352      Cell knight2 = new Cell(2, 5);
353      Cell knight3 = new Cell(5, 2);
354      Cell knight4 = new Cell(2, 2);
355      getBoard().addPiece(new Knight(getBoard(), knight1, PlayerColor.WHITE));
356      pawnAroundPos(knight1, PlayerColor.BLACK);
357      getBoard().addPiece(new Knight(getBoard(), knight2, PlayerColor.WHITE));
358      pawnAroundPos(knight2, PlayerColor.WHITE);
359      getBoard().addPiece(new Knight(getBoard(), knight3, PlayerColor.BLACK));
360      pawnAroundPos(knight3, PlayerColor.WHITE);
361      getBoard().addPiece(new Knight(getBoard(), knight4, PlayerColor.BLACK));
362      pawnAroundPos(knight4, PlayerColor.BLACK);
363  }
364
365  /**
366   * Permet de rapidement position des pièces afin de tester le fonctionnement des
367   * Tours.
368   */
369  private void testRook() {
370      Cell rookPos1 = new Cell(5, 4);
371      Cell rookPos2 = new Cell(2, 4);
372      getBoard().addPiece(new Rook(getBoard(), new Cell(0, 0), PlayerColor.WHITE));
373      getBoard().addPiece(new Rook(getBoard(), new Cell(7, 7), PlayerColor.BLACK));
374      getBoard().addPiece(new Rook(getBoard(), rookPos1, PlayerColor.WHITE));
375      getBoard().addPiece(new Rook(getBoard(), rookPos2, PlayerColor.WHITE));
376      pawnAroundPos(rookPos1, PlayerColor.WHITE);
377      pawnAroundPos(rookPos2, PlayerColor.BLACK);
378  }
379
380  /**
381   * Permet de rapidement position des pièces afin de tester le fonctionnement des
382   * Fous.
383   */
384  private void testBishop() {
385      Cell bishopPos1 = new Cell(5, 4);
386      Cell bishopPos2 = new Cell(2, 4);
387      getBoard().addPiece(new Bishop(getBoard(), bishopPos1, PlayerColor.WHITE));
388      getBoard().addPiece(new Bishop(getBoard(), bishopPos2, PlayerColor.WHITE));
389      pawnAroundPos(bishopPos1, PlayerColor.WHITE);
390      pawnAroundPos(bishopPos2, PlayerColor.BLACK);
391  }
392
393  /**
394   * Permet de rapidement position des pièces afin de tester le fonctionnement des
395   * Pions.
396   */
397  private void testPawn() {
398      for (int i = 0; i < 7; i++) {
399          getBoard().addPiece(new Pawn(getBoard(), new Cell(i, 1),
400              PlayerColor.WHITE));
401          getBoard().addPiece(new Knight(getBoard(), new Cell(i, 2), (i % 2 == 0) ?
402              PlayerColor.WHITE : PlayerColor.BLACK));
403      }
404      getBoard().addPiece(new Pawn(getBoard(), new Cell(7, 1), PlayerColor.BLACK));
405      getBoard().addPiece(new Pawn(getBoard(), new Cell(4, 6), PlayerColor.BLACK));
406      getBoard().addPiece(new Pawn(getBoard(), new Cell(3, 4), PlayerColor.WHITE));
407  }
408
409
410
411
412
413
414

```

```

415  /**
416   * Permet de rapidement position des pièces afin de tester le fonctionnement des
417   * Reines.
418   */
419  private void testQueen() {
420      Cell queenPos = new Cell(5, 4);
421      Cell queenPos2 = new Cell(2, 4);
422      getBoard().addPiece(new Queen(getBoard(), queenPos, PlayerColor.WHITE));
423      getBoard().addPiece(new Queen(getBoard(), queenPos2, PlayerColor.WHITE));
424      pawnAroundPos(queenPos, PlayerColor.BLACK);
425      pawnAroundPos(queenPos2, PlayerColor.WHITE);
426  }
427
428  /**
429   * Permet de rapidement position des pièces afin de tester le fonctionnement des
430   * Rois.
431   */
432  private void testKing() {
433      Cell kingPos = new Cell(4, 4);
434      getBoard().addPiece(new King(getBoard(), kingPos, PlayerColor.WHITE));
435      pawnAroundPos(kingPos, PlayerColor.BLACK);
436  }
437
438  /**
439   * Permet de rapidement position des pièces afin de tester le fonctionnement
440   * du roque.
441   */
442  private void testCastle() {
443      getBoard().addPiece(new King(getBoard(), new Cell(4, 0), PlayerColor.WHITE));
444      getBoard().addPiece(new Rook(getBoard(), new Cell(7, 0), PlayerColor.WHITE));
445      getBoard().addPiece(new Rook(getBoard(), new Cell(0, 0), PlayerColor.WHITE));
446      getBoard().addPiece(new Queen(getBoard(), new Cell(3, 7), PlayerColor.BLACK));
447  }
448
449  /**
450   * Permet de rapidement position des pièces afin de tester le fonctionnement
451   * de la mise en échec.
452   */
453  private void testCheck() {
454      getBoard().addPiece(new King(getBoard(), new Cell(3, 5), PlayerColor.BLACK));
455      getBoard().addPiece(new Rook(getBoard(), new Cell(4, 3), PlayerColor.WHITE));
456      getBoard().addPiece(new Queen(getBoard(), new Cell(5, 3), PlayerColor.WHITE));
457      getBoard().addPiece(new Bishop(getBoard(), new Cell(6, 3),
458          PlayerColor.WHITE));
459      getBoard().addPiece(new Knight(getBoard(), new Cell(7, 3),
460          PlayerColor.WHITE));
461      getBoard().addPiece(new King(getBoard(), new Cell(1, 3), PlayerColor.WHITE));
462      getBoard().addPiece(new Pawn(getBoard(), new Cell(2, 3), PlayerColor.WHITE));
463  }
464  }
465
466  /**
467   * Permet de rapidement position des pièces afin de tester le fonctionnement
468   * de l'echec et mat.
469   */
470  private void testCheckMate() {
471      getBoard().addPiece(new King(getBoard(), new Cell(0, 3), PlayerColor.BLACK));
472      getBoard().addPiece(new King(getBoard(), new Cell(2, 3), PlayerColor.WHITE));
473      getBoard().addPiece(new Queen(getBoard(), new Cell(3, 5), PlayerColor.WHITE));
474  }
475  }
476
477  // -----
478
479
480
481
482
483

```

```

484 package engine;
485
486 import chess.PieceType;
487 import chess.PlayerColor;
488 import engine.pieces.*;
489 import engine.utils.Cell;
490
491 import java.util.ArrayList;
492 import java.util.List;
493 import java.util.Objects;
494
495 /**
496  * Classe modélisant un plateau virtuel du jeu d'échecs.
497  * Elle s'occupe notamment de stocker et modifier les positions des différentes
498  * pièces.
499  *
500  * @author Jonathan Friedli
501  * @author Valentin Kaelin
502  */
503 public class Board {
504     public interface PieceListener {
505         void action(Piece piece, Cell cell);
506     }
507
508     public interface PromotionListener {
509         void action(Piece piece);
510     }
511
512     public static final int BOARD_SIZE = 8;
513     private int turn;
514     private final Piece[][] pieces;
515     private final List<King> kings;
516     private Piece lastPiecePlayed;
517
518     private PieceListener onAddPiece;
519     private PieceListener onRemovePiece;
520     private PromotionListener onPromotion;
521
522     /**
523      * Constructeur de base initialisant les différentes structures
524      */
525     public Board() {
526         pieces = new Piece[BOARD_SIZE][BOARD_SIZE];
527         kings = new ArrayList<>();
528     }
529
530     /**
531      * Remet le plateau à son état initial
532      */
533     public void resetBoard() {
534         // On vide le plateau pour éviter de recréer un tableau
535         for (int i = 0; i < BOARD_SIZE; i++)
536             for (int j = 0; j < BOARD_SIZE; j++)
537                 removePiece(new Cell(i, j));
538
539         kings.clear();
540         lastPiecePlayed = null;
541         turn = 0;
542     }
543
544
545
546
547
548
549
550
551
552

```



```

553 /**
554  * Remplit le tableau avec la position habituelle des différentes pièces
555  * Commence par les pièces blanches puis les noires
556  */
557 public void fillBoard() {
558     PlayerColor color = PlayerColor.WHITE;
559     int line = 0, pawnLine = 1;
560     for (int i = 0; i < 2; i++) {
561         addPiece(new Rook(this, new Cell(0, line), color));
562         addPiece(new Knight(this, new Cell(1, line), color));
563         addPiece(new Bishop(this, new Cell(2, line), color));
564         addPiece(new Queen(this, new Cell(3, line), color));
565         addPiece(new King(this, new Cell(4, line), color));
566         addPiece(new Bishop(this, new Cell(5, line), color));
567         addPiece(new Knight(this, new Cell(6, line), color));
568         addPiece(new Rook(this, new Cell(7, line), color));
569
570         // Pions
571         for (int xPawn = 0; xPawn < BOARD_SIZE; xPawn++)
572             addPiece(new Pawn(this, new Cell(xPawn, pawnLine), color));
573
574         color = PlayerColor.BLACK;
575         line = 7;
576         pawnLine = 6;
577     }
578 }
579
580 /**
581  * Vérifie que les coordonnées de la case sont valides
582  *
583  * @param cell : case à vérifier
584  * @throws RuntimeException si la case est invalide
585  */
586 private void checkCoordsOnBoard(Cell cell) {
587     if (cell == null || cell.getX() >= BOARD_SIZE || cell.getX() < 0 ||
588         cell.getY() >= BOARD_SIZE || cell.getY() < 0)
589         throw new RuntimeException("Coordonnées de la pièce invalides.");
590 }
591
592 /**
593  * @return le tour actuel
594  */
595 public int getTurn() {
596     return turn;
597 }
598
599 /**
600  * @return la dernière pièce jouée
601  */
602 public Piece getLastPiecePlayed() {
603     return lastPiecePlayed;
604 }
605
606 /**
607  * @param cell : case souhaitée
608  * @return la pièce à la case souhaitée ou null
609  * @throws RuntimeException si la case est invalide
610  */
611 public Piece getPiece(Cell cell) {
612     checkCoordsOnBoard(cell);
613     return pieces[cell.getX()][cell.getY()];
614 }
615
616
617
618
619
620
621

```

```

622     /**
623     * Ajoute la pièce à la case souhaitée
624     *
625     * @param piece : pièce à ajouter
626     * @param cell : case souhaitée
627     * @throws RuntimeException si la case est invalide
628     */
629     public void setPiece(Piece piece, Cell cell) {
630         Objects.requireNonNull(piece, "Pièce invalide");
631         checkCoordsOnBoard(cell);
632
633         pieces[cell.getX()][cell.getY()] = piece;
634         piece.setCell(cell);
635
636         if (piece.getType() == PieceType.KING)
637             kings.add((King) piece);
638
639         if (onAddPiece != null)
640             onAddPiece.action(piece, cell);
641     }
642
643     /**
644     * Petite fonction helper permettant d'ajouter une pièce à sa case actuelle
645     *
646     * @param piece : pièce à ajouter
647     * @throws RuntimeException si la case est invalide
648     */
649     public void addPiece(Piece piece) {
650         setPiece(piece, piece.getCell());
651     }
652
653     /**
654     * Supprime une pièce du tableau
655     *
656     * @param cell : case de la pièce à supprimer
657     * @throws RuntimeException si la case est invalide
658     */
659     public void removePiece(Cell cell) {
660         Piece piece = getPiece(cell);
661         pieces[cell.getX()][cell.getY()] = null;
662
663         if (piece != null) {
664             if (piece.getType() == PieceType.KING)
665                 kings.remove((King) piece);
666
667             if (onRemovePiece != null)
668                 onRemovePiece.action(piece, cell);
669         }
670     }
671
672     /**
673     * @return le joueur à qui c'est le tour de jouer
674     */
675     public PlayerColor currentPlayer() {
676         return turn % 2 == 0 ? PlayerColor.WHITE : PlayerColor.BLACK;
677     }
678
679
680
681
682
683
684
685
686
687
688
689
690

```

```

691  /**
692  * Vérifie que le déplacement d'une pièce peut se faire. Si c'est le cas,
693  * il est réalisé.
694  *
695  * @param from : case de départ
696  * @param to   : case d'arrivée
697  * @return true si le mouvement a pu être fait, false sinon
698  */
699  public boolean move(Cell from, Cell to) {
700      Piece p;
701      try {
702          p = getPiece(from);
703          checkCoordsOnBoard(to);
704      } catch (RuntimeException e) {
705          return false;
706      }
707
708      if (p == null || p.getColor() != currentPlayer())
709          return false;
710
711      if (p.checkMove(to) && p.applyMove(to)) {
712          postUpdate(p);
713          return true;
714      }
715
716      return false;
717  }
718
719  /**
720  * Applique le mouvement d'une pièce à une destination
721  *
722  * @param piece : pièce à déplacer
723  * @param to    : case d'arrivée
724  */
725  public void applyMove(Piece piece, Cell to) {
726      Objects.requireNonNull(piece, "Pièce invalide");
727      removePiece(piece.getCell());
728      removePiece(to);
729      setPiece(piece, to);
730  }
731
732  /**
733  * Vérifie si une pièce est actuellement menacée/attaquée
734  *
735  * @param color : couleur de la pièce à vérifier
736  * @param cell  : case de la pièce à vérifier
737  * @return true si la pièce est attaquée, false sinon
738  */
739  public boolean isAttacked(PlayerColor color, Cell cell) {
740      Objects.requireNonNull(color, "Couleur invalide");
741      Objects.requireNonNull(cell, "Case invalide");
742
743      for (Piece[] row : pieces)
744          for (Piece piece : row)
745              if (piece != null && piece.getColor() != color && piece.checkMove(cell))
746                  return true;
747
748      return false;
749  }
750
751
752
753
754
755
756
757
758
759

```

```

760 /**
761  * Vérifie si un joueur est actuellement en échec
762  *
763  * @param color : la couleur du joueur
764  * @return true si le joueur est en échec, false sinon
765  */
766 public boolean isCheck(PlayerColor color) {
767     Objects.requireNonNull(color, "Couleur invalide");
768
769     King king = kings.stream()
770         .filter(k -> k.getColor() == color)
771         .findAny()
772         .orElse(null);
773
774     if (king == null)
775         return false;
776
777     return isAttacked(color, king.getCell());
778 }
779
780 /**
781  * Vérifie si un roi déjà en échec et échec et mat ou non
782  *
783  * @param color : couleur du roi en échec
784  * @return true si le roi est échec et mat, false sinon
785  */
786 public boolean isCheckMate(PlayerColor color) {
787     // Boucle sur toutes les cases
788     for (int i = 0; i < BOARD_SIZE; i++) {
789         for (int j = 0; j < BOARD_SIZE; j++) {
790             Cell destination = new Cell(i, j);
791             Piece eaten = getPiece(destination);
792             // Vérifie qu'une pièce alliée peut bouger
793             for (Piece[] row : pieces) {
794                 for (Piece piece : row) {
795                     if (piece == null || piece.getColor() != color)
796                         continue;
797                     Cell oldPos = piece.getCell();
798                     if (piece.checkMove(destination) && piece.applyMove(destination)) {
799                         // On annule le déplacement qui a bien été effectué
800                         applyMove(piece, oldPos);
801                         if (eaten != null)
802                             applyMove(eaten, destination);
803                         return false;
804                     }
805                 }
806             }
807         }
808     }
809     return true;
810 }
811
812 /**
813  * Applique les changements nécessaires à la fin d'un tour
814  *
815  * @param piece : pièce jouée
816  */
817 private void postUpdate(Piece piece) {
818     Objects.requireNonNull(piece, "Pièce invalide");
819     lastPiecePlayed = piece;
820     piece.postUpdate();
821     turn++;
822 }
823
824
825
826
827
828

```

```

829     /**
830      * Définit le listener appelé lors de l'ajout d'une pièce
831      *
832      * @param onAddPiece : listener à exécuter
833      */
834     public void setAddPieceListener(PieceListener onAddPiece) {
835         Objects.requireNonNull(onAddPiece, "Listener invalide");
836         this.onAddPiece = onAddPiece;
837     }
838
839     /**
840      * Définit le listener appelé lors de la suppression d'une pièce
841      *
842      * @param onRemovePiece : listener à exécuter
843      */
844     public void setRemovePieceListener(PieceListener onRemovePiece) {
845         Objects.requireNonNull(onRemovePiece, "Listener invalide");
846         this.onRemovePiece = onRemovePiece;
847     }
848
849     /**
850      * Définit le listener appelé lors de la promotion d'une pièce
851      *
852      * @param onPromotion : listener à exécuter
853      */
854     public void setPromotionListener(PromotionListener onPromotion) {
855         Objects.requireNonNull(onPromotion, "Listener invalide");
856         this.onPromotion = onPromotion;
857     }
858
859     /**
860      * @return le listener appelé lors d'une promotion
861      */
862     public PromotionListener getOnPromotion() {
863         return onPromotion;
864     }
865 }
866
867 // -----
868
869 package engine.utils;
870
871 import java.util.Objects;
872
873 /**
874  * Classe représentant une case de l'échiquier
875  *
876  * @author Jonathan Friedli
877  * @author Valentin Kaelin
878  */
879 public class Cell {
880     private final int x;
881     private final int y;
882
883     /**
884      * @param x : coordonnée x de la case
885      * @param y : coordonnée y de la case
886      */
887     public Cell(int x, int y) {
888         this.x = x;
889         this.y = y;
890     }
891
892     /**
893      * @return la coordonnée X de la case
894      */
895     public int getX() {
896         return x;
897     }

```

```

898
899 /**
900  * @return la coordonnée Y de la case
901  */
902 public int getY() {
903     return y;
904 }
905
906 /**
907  * Additionne une seconde case
908  *
909  * @param cell : la case à ajouter
910  * @return le résultat de l'addition via une nouvelle case
911  * @throws RuntimeException si la case à additionner est invalide
912  */
913 public Cell add(Cell cell) {
914     if (cell == null)
915         throw new RuntimeException("Addition d'une case invalide");
916
917     return new Cell(x + cell.x, y + cell.y);
918 }
919
920 /**
921  * Soustrait une seconde case
922  *
923  * @param cell : la case à soustraire
924  * @return le résultat de la soustraction via une nouvelle case
925  * @throws RuntimeException si la case à soustraire est invalide
926  */
927 public Cell subtract(Cell cell) {
928     if (cell == null)
929         throw new RuntimeException("Soustraction d'une case invalide");
930
931     return new Cell(x - cell.x, y - cell.y);
932 }
933
934
935 /**
936  * Multiplie la case par un scalaire
937  *
938  * @param n : scalaire
939  * @return le résultat de la multiplication via une nouvelle case
940  */
941 public Cell multiply(int n) {
942     return new Cell(n * x, n * y);
943 }
944
945 /**
946  * Vérifie qu'une case peut être atteinte depuis une autre
947  *
948  * @param cell : case de potentielle arrivée
949  * @return true si la case est atteignable, false sinon
950  */
951 public boolean reachable(Cell cell) {
952     return cell != null && x * cell.y == y * cell.x;
953 }
954
955 /**
956  * Vérifie que deux cases ont les mêmes signes sur leurs deux coordonnées
957  *
958  * @param cell : la seconde case
959  * @return true si les signes sont les mêmes, false sinon
960  */
961 public boolean sameDirection(Cell cell) {
962     return cell != null && (x < 0 == cell.getX() < 0) && (y < 0 == cell.getY() < 0);
963 }
964
965
966

```

```

967     /**
968     * Retourne la distance jusqu'à une case.
969     * Ne vérifie pas si la case est accessible.
970     *
971     * @param to : case d'arrivée
972     * @return la distance entre les deux cases
973     * @throws RuntimeException si la case d'arrivée est invalide
974     */
975     public int getDistance(Cell to) {
976         Objects.requireNonNull(to, "Case invalide");
977         Cell fromTo = to.subtract(this);
978         return Math.max(Math.abs(fromTo.getX()), Math.abs(fromTo.getY()));
979     }
980
981     @Override
982     public int hashCode() {
983         return Objects.hash(x, y);
984     }
985
986     @Override
987     public boolean equals(Object o) {
988         if (this == o) return true;
989         if (o == null || getClass() != o.getClass()) return false;
990         Cell cell = (Cell) o;
991         return x == cell.x && y == cell.y;
992     }
993 }
994
995 // -----
996
997
998 package engine.utils;
999
1000 /**
1001  * Énumération permettant de modéliser des déplacements dans une certaine direction
1002  * Les directions gauches et droites ne sont pas utilisées, mais sont implémentée
1003  * dans un souci d'harmonisation.
1004  *
1005  * @author Jonathan Friedli
1006  * @author Valentin Kaelin
1007  */
1008 public enum Direction {
1009     UP(0, 1), DOWN(0, -1), LEFT(-1, 0), RIGHT(1, 0);
1010     private final Cell value;
1011
1012     private Direction(int x, int y) {
1013         this.value = new Cell(x, y);
1014     }
1015
1016     /**
1017     * @return la valeur de la direction sous forme d'une case
1018     */
1019     public Cell getValue() {
1020         return value;
1021     }
1022
1023     /**
1024     * @return la valeur de la direction sous forme d'un nombre
1025     */
1026     public int intValue() {
1027         return value.getX() == 0 ? value.getY() : value.getX();
1028     }
1029 }
1030
1031 // -----
1032
1033
1034
1035

```

```

1036 package engine.pieces;
1037
1038 import chess.ChessView;
1039 import chess.PieceType;
1040 import chess.PlayerColor;
1041 import engine.Board;
1042 import engine.moves.Move;
1043 import engine.utils.Cell;
1044
1045 import java.util.ArrayList;
1046 import java.util.List;
1047
1048 /**
1049  * Classe abstraite permettant de définir la base de toutes les pièces du jeu
1050  * d'échecs.
1051  *
1052  * @author Jonathan Friedli
1053  * @author Valentin Kaelin
1054  */
1055 public abstract class Piece implements ChessView.UserChoice {
1056     private final Board board;
1057     private final PlayerColor color;
1058     private Cell cell;
1059     protected List<Move> moves;
1060
1061     /**
1062      * Crée une nouvelle pièce
1063      *
1064      * @param board : plateau de la pièce
1065      * @param cell : case de la pièce
1066      * @param color : couleur de la pièce
1067      * @throws RuntimeException s'il manque un paramètre
1068      */
1069     public Piece(Board board, Cell cell, PlayerColor color) {
1070         if (board == null || cell == null || color == null)
1071             throw new RuntimeException("Construction de la pièce invalide");
1072
1073         this.board = board;
1074         this.cell = cell;
1075         this.color = color;
1076         moves = new ArrayList<>();
1077     }
1078
1079     /**
1080      * @return le type de la pièce
1081      */
1082     public abstract PieceType getType();
1083
1084     /**
1085      * @return le texte en français représentant la pièce
1086      */
1087     public abstract String textValue();
1088
1089     /**
1090      * @return le plateau de la pièce
1091      */
1092     public Board getBoard() {
1093         return board;
1094     }
1095
1096     /**
1097      * @return la couleur de la pièce
1098      */
1099     public PlayerColor getColor() {
1100         return color;
1101     }
1102
1103
1104

```



```

1105     @Override
1106     public String toString() {
1107         return textValue();
1108     }
1109
1110     /**
1111      * @return la case de la pièce
1112      */
1113     public Cell getCell() {
1114         return cell;
1115     }
1116
1117     /**
1118      * Change la case de la pièce
1119      *
1120      * @param cell : nouvelle case
1121      * @throws RuntimeException si le case est inexistante
1122      */
1123     public void setCell(Cell cell) {
1124         if (cell == null)
1125             throw new RuntimeException("Case de la pièce invalide.");
1126
1127         this.cell = cell;
1128     }
1129
1130
1131
1132     /**
1133      * Vérifie qu'un mouvement peut-être réalisé par la pièce
1134      *
1135      * @param to : case de destination souhaitée
1136      * @return true si le mouvement peut être fait, false sinon
1137      */
1138     public boolean checkMove(Cell to) {
1139         // Si la case de destination est occupée par une pièce de même couleur
1140         if (to == null || (board.getPiece(to) != null &&
1141             board.getPiece(to).getColor() == color))
1142             return false;
1143
1144         for (Move move : moves) {
1145             if (move.canMove(cell, to))
1146                 return true;
1147         }
1148
1149         return false;
1150     }
1151
1152     /**
1153      * Vérifie qu'un mouvement (légal) peut être appliqué
1154      *
1155      * @param to : case de destination souhaitée
1156      * @return true s'il peut être appliqué, false s'il met le roi du joueur en échec
1157      */
1158     public boolean applyMove(Cell to) {
1159         Cell oldCell = getCell();
1160         Piece eaten = board.getPiece(to);
1161
1162         board.applyMove(this, to);
1163
1164         // En échec : on annule le move
1165         if (board.isCheck(color)) {
1166             board.applyMove(this, oldCell);
1167             if (eaten != null)
1168                 board.setPiece(eaten, to);
1169             return false;
1170         }
1171
1172         return true;
1173     }

```

```

1174
1175     /**
1176     * Méthode à implémenter dans les pièces devant réaliser des actions après un
1177     * tour.
1178     */
1179     public void postUpdate() {
1180     }
1181 }
1182
1183 // -----
1184
1185 package engine.pieces;
1186
1187 import chess.PlayerColor;
1188 import engine.Board;
1189 import engine.utils.Cell;
1190
1191 /**
1192 * Classe abstraite permettant d'ajouter la gestion de premier coup spécifique à
1193 * certaines pièces.
1194 *
1195 * @author Jonathan Friedli
1196 * @author Valentin Kaelin
1197 */
1198 public abstract class FirstMoveSpecificPiece extends Piece {
1199     private boolean hasMoved;
1200
1201     public FirstMoveSpecificPiece(Board board, Cell cell, PlayerColor color) {
1202         super(board, cell, color);
1203         hasMoved = false;
1204     }
1205
1206     /**
1207     * @return true si la pièce a déjà bougé, false sinon
1208     */
1209     public boolean hasMoved() {
1210         return hasMoved;
1211     }
1212
1213     /**
1214     * Indique à la fin du tour que la pièce a déjà bougé
1215     */
1216     public void postUpdate() {
1217         hasMoved = true;
1218     }
1219 }
1220
1221 // -----
1222
1223 package engine.pieces;
1224
1225 import chess.PieceType;
1226 import chess.PlayerColor;
1227 import engine.Board;
1228 import engine.moves.LinearMove;
1229 import engine.utils.Cell;
1230
1231 /**
1232 * Classe représentant un fou
1233 *
1234 * @author Jonathan Friedli
1235 * @author Valentin Kaelin
1236 */
1237 public class Bishop extends Piece {
1238     public Bishop(Board board, Cell cell, PlayerColor color) {
1239         super(board, cell, color);
1240         moves.add(new LinearMove(this, new Cell(1, 1)));
1241         moves.add(new LinearMove(this, new Cell(1, -1)));
1242     }

```

```

1243
1244     @Override
1245     public PieceType getType() {
1246         return PieceType.BISHOP;
1247     }
1248
1249     @Override
1250     public String textValue() {
1251         return "Fou";
1252     }
1253 }
1254
1255 // -----
1256
1257 package engine.pieces;
1258
1259 import chess.PieceType;
1260 import chess.PlayerColor;
1261 import engine.Board;
1262 import engine.moves.LinearMove;
1263 import engine.utils.Cell;
1264
1265 /**
1266  * Classe représentant un roi
1267  *
1268  * @author Jonathan Friedli
1269  * @author Valentin Kaelin
1270  */
1271 public class King extends FirstMoveSpecificPiece {
1272     private static final int CASTLE_DISTANCE = 2;
1273
1274     public King(Board board, Cell cell, PlayerColor color) {
1275         super(board, cell, color);
1276         moves.add(new LinearMove(this, new Cell(0, 1), 1));
1277         moves.add(new LinearMove(this, new Cell(1, 0), 1));
1278         moves.add(new LinearMove(this, new Cell(1, 1), 1));
1279         moves.add(new LinearMove(this, new Cell(1, -1), 1));
1280     }
1281
1282     @Override
1283     public PieceType getType() {
1284         return PieceType.KING;
1285     }
1286
1287     @Override
1288     public String textValue() {
1289         return "Roi";
1290     }
1291
1292     @Override
1293     public boolean checkMove(Cell to) {
1294         return super.checkMove(to) || castle(to);
1295     }
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311

```

```

1312  /**
1313  * Vérifie si le déplacement est un roque légal
1314  *
1315  * @param to : case de destination
1316  * @return true si le roque a bien été effectué, false sinon
1317  */
1318  private boolean castle(Cell to) {
1319      if (to == null)
1320          return false;
1321      int deltaY = to.getY() - getCell().getY();
1322      int deltaX = to.getX() - getCell().getX();
1323      if (hasMoved() || Math.abs(deltaX) != CASTLE_DISTANCE || deltaY != 0)
1324          return false;
1325
1326      boolean leftSide = deltaX < 0;
1327      Cell direction = new Cell(leftSide ? -1 : 1, 0);
1328      Cell rookCell = new Cell(leftSide ? 0 : Board.BOARD_SIZE - 1, getCell().getY());
1329      Piece rook = getBoard().getPiece(rookCell);
1330      Cell rookDestination = getCell().add(direction);
1331
1332      // Vérification de la tour et que le chemin est libre
1333      if (rook == null || rook.getType() != PieceType.ROOK ||
1334          ((Rook) rook).hasMoved() || !rook.checkMove(rookDestination))
1335          return false;
1336
1337      // Vérification que le chemin ne met pas le roi en échec
1338      Cell initialPosition = getCell();
1339      for (int i = 0; i <= CASTLE_DISTANCE; i++) {
1340          Cell position = initialPosition.add(direction.multiply(i));
1341          getBoard().setPiece(this, position);
1342
1343          boolean isAttacked = getBoard().isAttacked(getColor(), position);
1344          getBoard().removePiece(position);
1345          if (isAttacked) {
1346              getBoard().setPiece(this, initialPosition);
1347              return false;
1348          }
1349      }
1350
1351      // Roque appliqué
1352      getBoard().applyMove(rook, rookDestination);
1353      rook.postUpdate();
1354
1355      return true;
1356  }
1357  }

```

```

1358
1359
1360  // -----
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380

```

```

1381 package engine.pieces;
1382
1383 import chess.PieceType;
1384 import chess.PlayerColor;
1385 import engine.Board;
1386 import engine.moves.LinearMove;
1387 import engine.utils.Cell;
1388
1389 /**
1390  * Classe représentant un cavalier
1391  *
1392  * @author Jonathan Friedli
1393  * @author Valentin Kaelin
1394  */
1395 public class Knight extends Piece {
1396     public Knight(Board board, Cell cell, PlayerColor color) {
1397         super(board, cell, color);
1398         moves.add(new LinearMove(this, new Cell(1, 2), 2, true));
1399         moves.add(new LinearMove(this, new Cell(1, -2), 2, true));
1400         moves.add(new LinearMove(this, new Cell(2, 1), 2, true));
1401         moves.add(new LinearMove(this, new Cell(2, -1), 2, true));
1402     }
1403
1404     @Override
1405     public PieceType getType() {
1406         return PieceType.KNIGHT;
1407     }
1408
1409     @Override
1410     public String textValue() {
1411         return "Cavalier";
1412     }
1413 }
1414
1415 // -----
1416
1417 package engine.pieces;
1418
1419 import chess.PieceType;
1420 import chess.PlayerColor;
1421 import engine.Board;
1422 import engine.utils.Direction;
1423 import engine.moves.OneDirectionMove;
1424 import engine.utils.Cell;
1425
1426 /**
1427  * Classe représentant un pion
1428  *
1429  * @author Jonathan Friedli
1430  * @author Valentin Kaelin
1431  */
1432 public class Pawn extends FirstMoveSpecificPiece {
1433     private final Direction direction;
1434     private int doubleMoveTurn;
1435
1436     public Pawn(Board board, Cell cell, PlayerColor color) {
1437         super(board, cell, color);
1438         direction = color == PlayerColor.WHITE ? Direction.UP : Direction.DOWN;
1439         moves.add(new OneDirectionMove(this, 1, direction));
1440         moves.add(new OneDirectionMove(this, 2, direction, true));
1441     }
1442
1443     @Override
1444     public PieceType getType() {
1445         return PieceType.PAWN;
1446     }
1447
1448
1449

```

```

1450     @Override
1451     public String textValue() {
1452         return "Pion";
1453     }
1454
1455     @Override
1456     public boolean checkMove(Cell to) {
1457         if (super.checkMove(to)) {
1458             // On stocke le tour actuel si le pion s'est déplacé de deux cases
1459             if (Math.abs(to.getY() - getCell().getY()) == 2)
1460                 doubleMoveTurn = getBoard().getTurn();
1461             return true;
1462         }
1463
1464         if (to == null)
1465             return false;
1466
1467         int deltaX = to.getX() - getCell().getX();
1468         int deltaY = to.getY() - getCell().getY();
1469
1470         // Manger en diagonale
1471         if (Math.abs(deltaX) == 1 && deltaY == direction.intValue()) {
1472             if (getBoard().getPiece(to) != null)
1473                 return true;
1474         }
1475         // En passant
1476         return enPassant(new Cell(to.getX(), getCell().getY()));
1477     }
1478
1479     @Override
1480     public boolean applyMove(Cell to) {
1481         if (to == null)
1482             return false;
1483
1484         Cell oldCell = getCell();
1485         Piece piece = getBoard().getLastPiecePlayed();
1486
1487         // Vérification de la mise en échec du en-passant
1488         if (enPassant(new Cell(to.getX(), oldCell.getY()))) {
1489             getBoard().applyMove(this, to);
1490             getBoard().removePiece(piece.getCell());
1491
1492             // En échec : on annule les moves
1493             if (getBoard().isCheck(getColor())) {
1494                 getBoard().applyMove(this, oldCell);
1495                 getBoard().setPiece(piece, piece.getCell());
1496                 return false;
1497             }
1498             return true;
1499         }
1500
1501         return super.applyMove(to);
1502     }
1503
1504     @Override
1505     public void postUpdate() {
1506         super.postUpdate();
1507
1508         // Gestion de la promotion
1509         if (canBePromoted() && getBoard().getOnPromotion() != null)
1510             getBoard().getOnPromotion().action(this);
1511     }
1512
1513
1514
1515
1516
1517
1518

```

```

1519     /**
1520     * @return true si le pion peut être promu, false sinon
1521     */
1522     public boolean canBePromoted() {
1523         return direction == Direction.UP ?
1524             getCell().getY() == Board.BOARD_SIZE - 1 :
1525             getCell().getY() == 0;
1526     }
1527
1528     /**
1529     * Vérifie si le move en-passant peut être réalisé
1530     *
1531     * @param cell : case de destination
1532     * @return true si le move est légal, false sinon
1533     */
1534     public boolean enPassant(Cell cell) {
1535         Piece piece = getBoard().getLastPiecePlayed();
1536         int lastTurn = getBoard().getTurn() - 1;
1537         return piece != null && piece != this && piece.getColor() != getColor() &&
1538             piece.getType() == PieceType.PAWN &&
1539             ((Pawn) piece).doubleMoveTurn == lastTurn &&
1540             piece.getCell().equals(cell);
1541     }
1542 }
1543
1544 // -----
1545
1546 package engine.pieces;
1547
1548 import chess.PieceType;
1549 import chess.PlayerColor;
1550 import engine.Board;
1551 import engine.moves.LinearMove;
1552 import engine.utils.Cell;
1553
1554 /**
1555  * Classe représentant une reine
1556  *
1557  * @author Jonathan Friedli
1558  * @author Valentin Kaelin
1559  */
1560 public class Queen extends Piece {
1561     public Queen(Board board, Cell cell, PlayerColor color) {
1562         super(board, cell, color);
1563         moves.add(new LinearMove(this, new Cell(0, 1)));
1564         moves.add(new LinearMove(this, new Cell(1, 0)));
1565         moves.add(new LinearMove(this, new Cell(1, 1)));
1566         moves.add(new LinearMove(this, new Cell(1, -1)));
1567     }
1568
1569     @Override
1570     public PieceType getType() {
1571         return PieceType.QUEEN;
1572     }
1573
1574     @Override
1575     public String textValue() {
1576         return "Reine";
1577     }
1578 }
1579
1580 // -----
1581
1582
1583
1584
1585
1586
1587

```

```

1588 package engine.pieces;
1589
1590 import chess.PieceType;
1591 import chess.PlayerColor;
1592 import engine.Board;
1593 import engine.moves.LinearMove;
1594 import engine.utils.Cell;
1595
1596 /**
1597  * Classe représentant une tour
1598  *
1599  * @author Jonathan Friedli
1600  * @author Valentin Kaelin
1601  */
1602 public class Rook extends FirstMoveSpecificPiece {
1603     public Rook(Board board, Cell cell, PlayerColor color) {
1604         super(board, cell, color);
1605         moves.add(new LinearMove(this, new Cell(0, 1)));
1606         moves.add(new LinearMove(this, new Cell(1, 0)));
1607     }
1608
1609     @Override
1610     public PieceType getType() {
1611         return PieceType.ROOK;
1612     }
1613
1614     @Override
1615     public String textValue() {
1616         return "Tour";
1617     }
1618 }
1619
1620 // -----
1621
1622 package engine.moves;
1623
1624 import engine.Board;
1625 import engine.pieces.Piece;
1626 import engine.utils.Cell;
1627
1628 /**
1629  * Classe abstraite modélisant la base des divers déplacements.
1630  * Le mouvement peut être limité à un nombre de cases.
1631  *
1632  * @author Jonathan Friedli
1633  * @author Valentin Kaelin
1634  */
1635 public abstract class Move {
1636     private final Piece piece;
1637     private final int maxDistance;
1638
1639     /**
1640      * Crée un déplacement
1641      *
1642      * @param piece : pièce concernée
1643      * @param maxDistance : potentielle distance maximale
1644      * @throws RuntimeException si les arguments sont invalides
1645      */
1646     public Move(Piece piece, int maxDistance) {
1647         if (piece == null || maxDistance < 0)
1648             throw new RuntimeException("Création du Move invalide");
1649
1650         this.piece = piece;
1651         this.maxDistance = maxDistance;
1652     }
1653
1654
1655
1656

```



```

1657     /**
1658     * Vérifie qu'une case peut être atteinte grâce au déplacement
1659     *
1660     * @param from : case de départ
1661     * @param to   : case d'arrivée
1662     * @return true si la case est atteignable, false sinon
1663     */
1664     public abstract boolean canMove(Cell from, Cell to);
1665
1666     /**
1667     * @return la pièce du déplacement
1668     */
1669     public Piece getPiece() {
1670         return piece;
1671     }
1672
1673     /**
1674     * Helper permettant de récupérer plus facilement le plateau du déplacement
1675     *
1676     * @return le plateau de la pièce du déplacement
1677     */
1678     public Board getBoard() {
1679         return piece.getBoard();
1680     }
1681
1682     /**
1683     * @return la distance maximale du déplacement
1684     */
1685     public int getMaxDistance() {
1686         return maxDistance;
1687     }
1688 }
1689
1690 // -----
1691
1692 package engine.moves;
1693
1694 import engine.pieces.Piece;
1695 import engine.utils.Cell;
1696
1697 /**
1698 * Classe représentant un déplacement linéaire dans un plan 2D.
1699 * La gestion des collisions est potentiellement gérée.
1700 *
1701 * @author Jonathan Friedli
1702 * @author Valentin Kaelin
1703 */
1704 public class LinearMove extends Move {
1705     protected final Cell direction;
1706     private final boolean flyOver;
1707
1708     /**
1709     * Crée un déplacement linéaire
1710     *
1711     * @param piece      : pièce concernée
1712     * @param direction  : direction du déplacement
1713     * @param maxDistance : potentielle distance maximale
1714     * @param flyOver    : indique si la pièce prend en compte les collisions ou pas
1715     * @throws RuntimeException si les arguments sont invalides
1716     */
1717     public LinearMove(Piece piece, Cell direction, int maxDistance, boolean flyOver) {
1718         super(piece, maxDistance);
1719
1720         if (direction == null)
1721             throw new RuntimeException("Création du LinearMove invalide");
1722
1723         this.direction = direction;
1724         this.flyOver = flyOver;
1725     }

```

```

1726
1727 /**
1728  * Crée un déplacement linéaire
1729  *
1730  * @param piece      : pièce concernée
1731  * @param direction  : direction du déplacement
1732  * @param maxDistance : potentielle distance maximale
1733  */
1734 public LinearMove(Piece piece, Cell direction, int maxDistance) {
1735     this(piece, direction, maxDistance, false);
1736 }
1737
1738 /**
1739  * Crée un déplacement linéaire
1740  *
1741  * @param piece      : pièce concernée
1742  * @param direction  : direction du déplacement
1743  */
1744 public LinearMove(Piece piece, Cell direction) {
1745     this(piece, direction, Integer.MAX_VALUE, false);
1746 }
1747
1748
1749 @Override
1750 public boolean canMove(Cell from, Cell to) {
1751     if (from == null || to == null)
1752         return false;
1753
1754     Cell fromTo = to.subtract(from);
1755     int distance = direction.reachable(fromTo) ? from.getDistance(to) : 0;
1756     int sign = direction.sameDirection(fromTo) ? 1 : -1;
1757
1758     if (distance == 0 || distance > getMaxDistance())
1759         return false;
1760
1761     // Gestion des collisions
1762     if (!flyOver) {
1763         for (int i = 1; i < distance; ++i) {
1764             Cell position = from.add(direction.multiply(i * sign));
1765             // Si une case sur le chemin est occupée
1766             if (getBoard().getPiece(position) != null)
1767                 return false;
1768         }
1769     }
1770
1771     return true;
1772 }
1773
1774
1775 // -----
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794

```

```

1795 package engine.moves;
1796
1797 import engine.pieces.FirstMoveSpecificPiece;
1798 import engine.pieces.Piece;
1799 import engine.utils.Cell;
1800 import engine.utils.Direction;
1801
1802 /**
1803  * Classe représentant un déplacement réduit à une seule direction.
1804  * Le déplacement peut potentiellement être à usage unique.
1805  * La gestion des collisions est également gérée.
1806  *
1807  * @author Jonathan Friedli
1808  * @author Valentin Kaelin
1809  */
1810 public class OneDirectionMove extends Move {
1811     private final Direction boundToDirection;
1812     private final boolean oneTimeMove;
1813
1814     /**
1815      * Crée un déplacement à une direction
1816      *
1817      * @param piece          : pièce concernée
1818      * @param maxDistance    : potentielle distance maximale
1819      * @param boundToDirection : unique direction possible
1820      * @param oneTimeMove    : true si le déplacement est à usage unique
1821      */
1822     public OneDirectionMove(Piece piece, int maxDistance, Direction boundToDirection,
1823                             boolean oneTimeMove) {
1824         super(piece, maxDistance);
1825
1826         if (boundToDirection == null)
1827             throw new RuntimeException("Création du OneDirectionMove invalide");
1828
1829         this.boundToDirection = boundToDirection;
1830         this.oneTimeMove = oneTimeMove;
1831     }
1832
1833     /**
1834      * Crée un déplacement à une direction
1835      *
1836      * @param piece          : pièce concernée
1837      * @param maxDistance    : potentielle distance maximale
1838      * @param boundToDirection : unique direction possible
1839      */
1840     public OneDirectionMove(Piece piece, int maxDistance, Direction boundToDirection) {
1841         this(piece, maxDistance, boundToDirection, false);
1842     }
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863

```

```

1864 @Override
1865 public boolean canMove(Cell from, Cell to) {
1866     if (from == null || to == null)
1867         return false;
1868
1869     // Vérification si le déplacement est à usage unique
1870     if (oneTimeMove && (!(getPiece() instanceof FirstMoveSpecificPiece) ||
1871         ((FirstMoveSpecificPiece) getPiece()).hasMoved()))
1872         return false;
1873
1874     Cell calculatedTo = from.add(
1875         boundToDirection.getValue().multiply(getMaxDistance())
1876     );
1877
1878     for (int i = 1; i < getMaxDistance(); ++i) {
1879         Cell position = from.add(boundToDirection.getValue().multiply(i));
1880         // Si une case sur le chemin est occupée
1881         if (getBoard().getPiece(position) != null)
1882             return false;
1883     }
1884
1885     return getBoard().getPiece(to) == null && to.equals(calculatedTo);
1886 }
1887 }
1888

```