```java
package Game;

import Pieces.Piece;
import Pieces.PieceType;
import java.util.ArrayList;

/**
 * This class represents the player who will be playing chess
 *
 * @author Ethan Palser, Param Jansari
 */
public class Player {

    private final Colour colour; // Black or White
    private int piecesCentred; // number of pieces in center of board
    private int repeatedMoves; // counts to 3 (draw), resets if either doesn't repeat
    private Piece lastMoved; // the last piece moved
    private int lastR, lastC; // previous postion of the last piece moved
    private int[][] attacks; // where player can attack
    private boolean lostGame; // if they lost the game

    public Player(Colour c) {
        colour = c;
        piecesCentred = 0;
        lastR = -1; // defaults that indicate no piece
        lastC = -1; // defaults that indicate no piece
        attacks = new int[8][8];
        lostGame = false;
    }

    /**
     * This method moves the piece on the board
     *
     * @param opponent
     * @param board
     * @param move
     * @param promotionTo
     * @return
     */
    public Board movePiece(Player opponent, Board board, Move move, Piece promotionTo) {
        return movePiece(opponent, board, move.startR, move.startC, move.nextR,
        move.nextC, promotionTo);
    }

    /**
     * This method moves the piece on the board
     *
     * @param opponent
     * @param board
     * @param startR
     * @param startC
     * @param nextR
     * @param nextC
     * @param promotionTo
     * @return
     */
    public Board movePiece(
            Player opponent,
            Board board,
            int startR,
            int startC,
            int nextR,
            int nextC,
            Piece promotionTo) {
        Board next = new Board(board);
        Piece toMove = next.getBoard()[startR][startC];
        if (toMove == null || toMove.colour != colour) {
            System.out.println("Failed to Move Piece");
            return board; // nothing happens to board state or player states
```

```
69          }
70          boolean[][] validPositions = toMove.validMoves(opponent, next, startR, startC);
71          if (validPositions[nextR][nextC] == false) {
72              System.out.println("Invalid Move");
73              return board; // invalid action
74          } else {
75              // May need to have repeated check before move is considered (maybe in board)
76              // valid action occurs
77              lastMoved = next.getBoard()[startR][startC];
78              lastR = nextR;
79              lastC = nextC;
80              // check if rook or king moves, if so castling not possible unless
                performed this move
81              if (toMove.piece == PieceType.Rook || toMove.piece == PieceType.King) {
82                  next.getBoard()[startR][startC].modifySpecial();
83              } // check if pawn moved and if it can be en passant after
84              else if (toMove.piece == PieceType.Pawn
85                      && (toMove.colour == Colour.White && nextR == startR - 2
86                      || toMove.colour == Colour.Black && nextR == startR + 2)) {
87                  next.getBoard()[startR][startC].modifySpecial();
88              }
89              // get a list of actions that will occur
90              ArrayList<Action> actions = actionTaken(opponent, next, startR, startC,
                nextR, nextC);
91              // update player state of attacks
92              this.updateAttacks(next, startR, startC, nextR, nextC);
93              // gets a copy of the board to modify
94              next.getBoard()[nextR][nextC] = next.getBoard()[startR][startC];
95              next.getBoard()[startR][startC] = null;
96              // output results to board
97              next.printToLog(toMove, nextR, nextC, actions, promotionTo);
98              System.out.println("Move Complete!");
99              return next; // returns new board state after applying move
100         }
101     }
102
103     /**
104      * This method determines what action did the player perform i.e. Castle, En
105      * Passant, Capture ... etc.
106      *
107      * @param opponent
108      * @param board
109      * @param move
110      * @return
111      */
112     public ArrayList<Action> actionTaken(Player opponent, Board board, Move move) {
113         return actionTaken(opponent, board, move.startR, move.startC, move.nextR,
            move.nextC);
114     }
115
116     /**
117      * This method determines what action did the player perform i.e. Castle, En
118      * Passant, Capture ... etc.
119      *
120      * @param opponent
121      * @param board
122      * @param startR
123      * @param startC
124      * @param nextR
125      * @param nextC
126      * @return
127      */
128     public ArrayList<Action> actionTaken(
129             Player opponent,
130             Board board,
131             int startR,
132             int startC,
133             int nextR,
134             int nextC) {
```

```
135             //System.out.println(startR + "  " + startC);
136             Piece pieceMoved = board.getBoard()[startR][startC];
137             //System.out.println(pieceMoved == null);
138             Piece pieceAt = board.getBoard()[nextR][nextC];
139             ArrayList<Action> actions = new ArrayList<>();
140             int backRow = colour == Colour.White ? 7 : 0;
141             //Move
142             if (pieceAt == null) {
143                 actions.add(Action.Move);
144                 //Castling (assume move has been validated already)
145                 if (pieceMoved.piece == PieceType.King) {
146                     // can castle
147                     if (pieceMoved.validSpecial()) {
148                         // check queen side
149                         if (nextR == backRow && nextC == 2) {
150                             // check if rook has moved
151                             if (board.getBoard()[backRow][0] != null
152                                     && board.getBoard()[backRow][0].validSpecial()) {
153                                 actions.add(Action.CastleQueenSide);
154                             }
155                         } // check king side
156                         else if (nextR == backRow && nextC == 6) {
157                             // check if rook has moved
158                             if (board.getBoard()[backRow][7] != null
159                                     && board.getBoard()[backRow][7].validSpecial()) {
160                                 actions.add(Action.CastleKingSide);
161                             }
162                         }
163                     }
164                 }
165             } else {
166                 //Capture
167                 if (pieceAt.colour != colour) {
168                     actions.add(Action.Capture);
169                     //Checkmate
170                     if (pieceAt.piece == PieceType.King) {
171                         actions.add(Action.Checkmate);
172                         opponent.setLoss();
173                     }
174                 } else {
175                     // Invalid action
176                 }
177             }
178             if (pieceMoved.piece == PieceType.Pawn) {
179                 //Promotion
180                 if (colour == Colour.White && nextR == 0 || colour == Colour.Black && nextR
                    == 7) {
181                     actions.add(Action.Promotion);
182                 }
183                 //check if opponent last moved pawn by two spaces
184                 Piece lastMoved = opponent.getLastMoved();
185                 if (lastMoved != null && lastMoved.piece == PieceType.Pawn &&
                    lastMoved.validSpecial()) {
186                     //checks if my pawn is in right position and moves to right space
187                     if (colour == Colour.White && startR == 3
188                             && (opponent.getLastC() == startC - 1
189                             || opponent.getLastC() == startC + 1)) {
190                         if (nextC == opponent.getLastC()) {
191                             actions.add(Action.EnPassant);
192                         }
193                     } else if (colour == Colour.Black && startR == 4
194                             && (opponent.getLastC() == startC - 1
195                             || opponent.getLastC() == startC + 1)) {
196                         if (nextC == opponent.getLastC()) {
197                             actions.add(Action.EnPassant);
198                         }
199                     }
200                 }
201             }
```

```
202             return actions;
203         }
204 
205         /**
206          * This method check if the Player has repeated the same move 3 times
207          *
208          * @param opponent
209          * @param board
210          * @param move
211          * @return
212          */
213         public boolean checkRepeat(Player opponent, Board board, Move move) {
214             return checkRepeat(opponent, board, move.startR, move.startC, move.nextR,
                 move.nextC);
215         }
216 
217         /**
218          * This method check if the Player has repeated the same move 3 times
219          *
220          * @param opponent
221          * @param board
222          * @param startR
223          * @param startC
224          * @param nextR
225          * @param nextC
226          * @return
227          */
228         public boolean checkRepeat(Player opponent, Board board, int startR, int startC,
             int nextR, int nextC) {
229             Piece toMove = board.getBoard()[startR][startC];
230             if (lastMoved.equals(toMove) && lastR == nextR && lastC == nextC) {
231                 this.updateRepeat(false);
232             } else {
233                 this.updateRepeat(true);
234                 opponent.updateRepeat(true);
235             }
236             return repeatedMoves == 3 && opponent.repeatedMoves == 3;
237         }
238 
239         /**
240          * This method updates the counter for the number of times the player has
241          * repeated a move
242          *
243          * @param reset
244          */
245         public void updateRepeat(boolean reset) {
246             if (reset) {
247                 repeatedMoves = 0;
248             } else {
249                 repeatedMoves++;
250             }
251         }
252 
253         /**
254          * This method calculates the number of pieces the player has in the center
255          * of the board
256          *
257          * @param board
258          * @return
259          */
260         public int piecesCentered(Board board) {
261             for (int i = 2; i < 6; i++) {
262                 for (int j = 2; j < 6; j++) {
263                     if (board.getBoard()[i][j].colour == colour) {
264                         piecesCentred++;
265                     }
266                 }
267             }
268             return piecesCentred;
```

```
269         }
270
271         /**
272          * This method determines where the player can attack
273          *
274          * @param board
275          * @param startR
276          * @param startC
277          */
278         public void setupAttacks(Board board, int startR, int startC) {
279             Piece toExamine = board.getBoard()[startR][startC];
280             int[][] examinedAttacks = toExamine.attacks(board, startR, startC);
281             for (int i = 0; i < 8; i++) {
282                 for (int j = 0; j < 8; j++) {
283                     // could have attacks be changed to boolean
284                     if (examinedAttacks[i][j] > 0) {
285                         attacks[i][j]++;
286                     }
287                 }
288             }
289         }
290
291         /**
292          * This method updates where the player can attack
293          *
294          * @param board
295          * @param move
296          */
297         private void updateAttacks(Board board, Move move) {
298             updateAttacks(board, move.startR, move.startC, move.nextR, move.nextC);
299         }
300
301         /**
302          * This method updates where the player can attack
303          *
304          * @param board
305          * @param startR
306          * @param startC
307          * @param nextR
308          * @param nextC
309          */
310         private void updateAttacks(Board board, int startR, int startC, int nextR, int
        nextC) {
311             Piece toExamine = board.getBoard()[startR][startC];
312             int[][] examinedAttacks = toExamine.attacks(board, startR, startC);
313             int[][] nextAttacks = toExamine.attacks(board, nextR, nextC);
314             for (int i = 0; i < 8; i++) {
315                 for (int j = 0; j < 8; j++) {
316                     // could have attacks be changed to boolean
317                     if (examinedAttacks[i][j] > 0) {
318                         attacks[i][j]--;
319                     }
320                     // could have attacks be changed to boolean
321                     if (nextAttacks[i][j] > 0) {
322                         attacks[i][j]++;
323                     }
324                 }
325             }
326         }
327
328         /**
329          * This method returns last row position of the last piece moved by player
330          *
331          * @return
332          */
333         public int getLastR() {
334             return lastR;
335         }
336
```

```
337         /**
338          * This method returns last column position of the last piece moved by
339          * player
340          *
341          * @return
342          */
343         public int getLastC() {
344             return lastC;
345         }
346 
347         /**
348          * This method returns the last piece moved by player
349          *
350          * @return
351          */
352         public Piece getLastMoved() {
353             return lastMoved;
354         }
355 
356         /**
357          * This method returns where the player can attack
358          *
359          * @return
360          */
361         public int[][] getAttacks() {
362             return attacks;
363         }
364 
365         /**
366          * This method gives the loss to the player
367          */
368         public void setLoss() {
369             lostGame = true;
370         }
371 
372         /**
373          * This method determines if the player has lost
374          *
375          * @return
376          */
377         public boolean getLoss() {
378             return lostGame;
379         }
380     }
381 
```