```java
package Pieces;

import Game.Board;
import Game.Colour;
import Game.Move;
import Game.Player;

/**
 * This class represents a chess piece and all it can do on the board.
 *
 * @author Ethan Palser, Param Jansari
 */
public abstract class Piece {

    public final PieceType piece; // what kind of piece it is
    public final Colour colour; // what colour is it
    public final int weight; // what's its value
    private Move bestMove; // the best move it can do from its current position

    /**
     * Creates Chess piece
     *
     * @param piece
     * @param colour
     * @param weight
     */
    public Piece(PieceType piece, Colour colour, int weight) {
        this.piece = piece;
        this.colour = colour;
        this.weight = weight;
    }

    /**
     * This method checks how good the position of the piece on the board is.
     *
     * @param board
     * @param row
     * @param column
     * @return heuristic value
     */
    public int heuristic(Board board, int row, int column) {
        int heurVal = 0;
        Piece toExamine = board.getBoard()[row][column];
        if (toExamine != null) {
            heurVal += toExamine.weight;
            heurVal += (threats(board, row, column)
                    - this.isThreatened(board, row, column));
        }
        return heurVal;
    }

    /**
     * This method checks how good the position of the piece on the board is.
     *
     * @param board
     * @param move
     * @return
     */
    public int heuristic(Board board, Move move) {
        if (move == null) {
            return -9999;
        }
        return heuristic(board, move.nextR, move.nextC);
    }

    /**
     * This method calculates the number of threats the piece has based on its
     * position
     *
```

```
70          * @param board
71          * @param row
72          * @param column
73          * @return
74          */
75         public abstract int threats(Board board, int row, int column);
76
77         /**
78          * This method calculates the number of attacks the piece can make based on
79          * its position
80          *
81          * @param board
82          * @param row
83          * @param column
84          * @return
85          */
86         public abstract int[][] attacks(Board board, int row, int column);
87
88         /**
89          * This method calculate all the valid positions the piece can move to based
90          * on its current position
91          *
92          * @param opponent
93          * @param board
94          * @param row
95          * @param column
96          * @return
97          */
98         public abstract boolean[][] validMoves(Player opponent, Board board, int row, int
           column);
99
100        /**
101         * This method checks if a piece has a special rule that applies to it Such
102         * a promotion for pawns, or castling for king/rook
103         *
104         * @return
105         */
106        public abstract boolean validSpecial();
107
108        /**
109         * This method executes the special rule of the piece, in the case it has a
110         * special move i.e. execute under condition a move confirms (pawn) or
111         * invalidates (king/rook) its special action
112         */
113        public abstract void modifySpecial();
114
115        /**
116         * This method prints piece to board
117         *
118         * @return
119         */
120        public abstract String printToBoard(); // prints piece board
121
122        /**
123         * This method prints piece to log
124         *
125         * @return
126         */
127        public abstract String printToLog();
128
129        /**
130         * This method calculates the number of pieces currently threaten it
131         *
132         * @param board
133         * @param row
134         * @param column
135         * @return
136         */
137        public int isThreatened(Board board, int row, int column) {
```

```
138             Piece[][] currentBoard = board.getBoard();
139             int threatCounter = 0; // increment if opponent, decrement if own
140             // Pawn - White
141             if (column >= 1) {
142                 if (row >= 1) {
143                     threatCounter += checkPawnWhite(
144                             currentBoard, row - 1, column - 1);
145                 }
146                 if (row <= 6) {
147                     threatCounter += checkPawnWhite(
148                             currentBoard, row + 1, column - 1);
149 
150                 }
151             }
152             // Pawn - Black
153             if (column <= 6 && column > 0) {
154                 if (row >= 1) {
155                     threatCounter += checkPawnBlack(
156                             currentBoard, row - 1, column - 1);
157                 }
158                 if (row <= 6) {
159                     threatCounter += checkPawnBlack(
160                             currentBoard, row + 1, column + 1);
161 
162                 }
163             }
164             // Queen + Rook
165             int result;
166             // check left
167             if (row > 0) {
168                 for (int x = row - 1; x >= 0; x--) {
169                     result = checkPiece(currentBoard, x, column,
170                             PieceType.Queen, PieceType.Rook);
171                     // piece encountered
172                     if (result != 0) {
173                         threatCounter += result;
174                         break;
175                     }
176                 }
177             }
178             // check right
179             if (row < 7) {
180                 for (int x = row + 1; x <= 7; x++) {
181                     result = checkPiece(currentBoard, x, column,
182                             PieceType.Queen, PieceType.Rook);
183                     // piece encountered
184                     if (result != 0) {
185                         threatCounter += result;
186                         break;
187                     }
188                 }
189             }
190             // check up
191             if (column > 0) {
192                 for (int y = column; y <= 0; y--) {
193                     result = checkPiece(currentBoard, row, y,
194                             PieceType.Queen, PieceType.Rook);
195                     // piece encountered
196                     if (result != 0) {
197                         threatCounter += result;
198                         break;
199                     }
200                 }
201             }
202             // check down
203             if (column < 7) {
204                 for (int y = column + 1; y <= 7; y++) {
205                     result = checkPiece(currentBoard, row, y,
206                             PieceType.Queen, PieceType.Rook);
```

```
                // piece encountered
                if (result != 0) {
                    threatCounter += result;
                    break;
                }
            }
        }
        // Queen + Bishop
        int posx = row;
        int posy = column;
        // diagonal top-left
        while (posx > 0 && posy > 0) {
            posx--;
            posy--;
            result = checkPiece(currentBoard, posx, posy,
                    PieceType.Queen, PieceType.Bishop);
            // piece encountered
            if (result != 0) {
                threatCounter += result;
                break;
            }
        }
        posx = row;
        posy = column;
        // diagonal top-right
        while (posx < 7 && posy > 0) {
            posx++;
            posy--;
            result = checkPiece(currentBoard, posx, posy,
                    PieceType.Queen, PieceType.Bishop);
            // piece encountered
            if (result != 0) {
                threatCounter += result;
                break;
            }
        }
        posx = row;
        posy = column;
        // diagonal bottom-left
        while (posx > 0 && posy < 7) {
            posx--;
            posy++;
            result = checkPiece(currentBoard, posx, posy,
                    PieceType.Queen, PieceType.Bishop);
            // piece encountered
            if (result != 0) {
                threatCounter += result;
                break;
            }
        }
        posx = row;
        posy = column;
        // diagonal bottom-right
        while (posx < 7 && posy < 7) {
            posx++;
            posy++;
            result = checkPiece(currentBoard, posx, posy,
                    PieceType.Queen, PieceType.Bishop);
            // piece encountered
            if (result != 0) {
                threatCounter += result;
                break;
            }
        }
        // Knight
        // top-left
        if (row >= 2 && column >= 1) {
            threatCounter += checkPiece(
                    currentBoard, row - 2, column - 1, PieceType.Knight);
```

```
276             }
277             if (row >= 1 && column >= 2) {
278                 threatCounter += checkPiece(
279                         currentBoard, row - 1, column - 2, PieceType.Knight);
280             }
281             // top-right
282             if (row >= 2 && column <= 6) {
283                 threatCounter += checkPiece(
284                         currentBoard, row - 2, column + 1, PieceType.Knight);
285             }
286             if (row >= 1 && column <= 5) {
287                 threatCounter += checkPiece(
288                         currentBoard, row - 1, column + 2, PieceType.Knight);
289             }
290             // bottom-left
291             if (row <= 5 && column >= 1) {
292                 threatCounter += checkPiece(
293                         currentBoard, row + 2, column - 1, PieceType.Knight);
294             }
295             if (row <= 6 && column >= 2) {
296                 threatCounter += checkPiece(
297                         currentBoard, row + 1, column - 2, PieceType.Knight);
298             }
299             // bottom-right
300             if (row <= 5 && column <= 6) {
301                 threatCounter += checkPiece(
302                         currentBoard, row + 2, column + 1, PieceType.Knight);
303             }
304             if (row <= 6 && column <= 5) {
305                 threatCounter += checkPiece(
306                         currentBoard, row + 1, column + 2, PieceType.Knight);
307             }
308             // King
309             if (row >= 1) {
310                 // top
311                 threatCounter += checkPiece(
312                         currentBoard, row - 1, column, PieceType.King);
313                 if (column >= 1) {
314                     // top-left
315                     threatCounter += checkPiece(
316                             currentBoard, row - 1, column - 1, PieceType.King);
317                 }
318                 if (column <= 6) {
319                     // top-right
320                     threatCounter += checkPiece(
321                             currentBoard, row - 1, column + 1, PieceType.King);
322                 }
323             }
324             if (row <= 6) {
325                 // bottom
326                 threatCounter += checkPiece(
327                         currentBoard, row + 1, column, PieceType.King);
328                 if (column >= 1) {
329                     // bottom-left
330                     threatCounter += checkPiece(
331                             currentBoard, row + 1, column - 1, PieceType.King);
332                 }
333                 if (column <= 6) {
334                     // bottom-right
335                     threatCounter += checkPiece(
336                             currentBoard, row + 1, column + 1, PieceType.King);
337                 }
338             }
339             if (column >= 1) {
340                 // left
341                 threatCounter += checkPiece(
342                         currentBoard, row, column - 1, PieceType.King);
343             }
344             if (column <= 6) {
```

```
345                 // right
346                 threatCounter += checkPiece(
347                         currentBoard, row, column + 1, PieceType.King);
348             }
349             return threatCounter;
350         }
351 
352         // isThreatened considers both of these worst cases
353         /*
354         public boolean isForked(Board board, int indexX, int indexY) {
355             return isThreatened(board, indexX, indexY) > 1;
356         }
357 
358         public boolean isPinned(Board board, int indexX, int indexY) {
359             return false;
360         }
361          */
362         /**
363          * This method is used to check if a piece is opposite to the current piece
364          *
365          * @param otherPiece
366          * @return
367          */
368         public boolean isOppositeColour(Piece otherPiece) {
369             if (this.colour == Colour.White) {
370                 return this.colour == Colour.Black;
371             } else {
372                 return this.colour == Colour.White;
373             }
374         }
375 
376         /**
377          * This method checks all horizonal, vertical and knight-shaped positions,
378          * and if that spot corresponds to the PieceType required
379          *
380          * @param board
381          * @param row
382          * @param column
383          * @param required
384          * @return
385          */
386         private int checkPiece(Piece[][] board, int row, int column,
387                 PieceType required) {
388             Piece toExamine = board[row][column];
389             if (toExamine != null) {
390                 if (toExamine.piece == required) {
391                     if (this.isOppositeColour(toExamine)) {
392                         return 1;
393                     } else {
394                         return -1;
395                     }
396                 }
397             }
398             return 0;
399         }
400 
401         /**
402          * This method checks all horizonal, vertical and knight-shaped positions,
403          * and if that spot corresponds to the PieceType required or required2
404          *
405          * @param board
406          * @param row
407          * @param column
408          * @param required1
409          * @param required2
410          * @return
411          */
412         private int checkPiece(Piece[][] board, int row, int column,
413                 PieceType required1, PieceType required2) {
```

```
414             Piece toExamine = board[row][column];
415             if (toExamine != null) {
416                 if (toExamine.piece == required1
417                         || toExamine.piece == required2) {
418                     if (this.isOppositeColour(toExamine)) {
419                         return 1;
420                     } else {
421                         return -1;
422                     }
423                 }
424             }
425             return 0;
426         }
427 
428         /**
429          * This method checks if white pawn is at location on board
430          *
431          * @param board
432          * @param row
433          * @param column
434          * @return
435          */
436         private int checkPawnWhite(Piece[][] board, int row, int column) {
437             Piece toExamine = board[row][column];
438             if (toExamine != null) {
439                 if (toExamine.piece == PieceType.Pawn) {
440                     if (this.colour == Colour.White
441                             && this.isOppositeColour(toExamine)) {
442                         return 1;
443                     } else if (this.colour == Colour.Black
444                             && !this.isOppositeColour(toExamine)) {
445                         return -1;
446                     }
447                 }
448             }
449             return 0;
450         }
451 
452         /**
453          * This method checks if black pawn is at location on board
454          *
455          * @param board
456          * @param row
457          * @param column
458          * @return
459          */
460         private int checkPawnBlack(Piece[][] board, int row, int column) {
461             // ensure that for the current piece the piece to check is previous
462             Piece toExamine = board[row][column];
463             if (toExamine != null) {
464                 if (toExamine.piece == PieceType.Pawn) {
465                     if (this.colour == Colour.Black
466                             && this.isOppositeColour(toExamine)) {
467                         return 1;
468                     } else if (this.colour == Colour.White
469                             && !this.isOppositeColour(toExamine)) {
470                         return -1;
471                     }
472                 }
473             }
474             return 0;
475         }
476 
477         /**
478          * This method calculate the best move the piece can make based on its
479          * current position
480          *
481          * @param opponent
482          * @param board
```

```
     * @param row
     * @param column
     * @return
     */
    public Move calcBestMove(Player opponent, Board board, int row, int column) {
        boolean[][] validMoves = this.validMoves(opponent, board, row, column);
        Move currentBest = null;
        for (int i = 0; i < validMoves.length; i++) {
            for (int j = 0; j < validMoves[0].length; j++) {
                if (validMoves[i][j]) {
                    Move move = new Move(row, column, i, j);
                    if (bestMove == null) {
                        currentBest = move;
                        bestMove = currentBest;
                    } else if (heuristic(board, move.nextR, move.nextC)
                            > heuristic(board, bestMove.nextR, bestMove.nextC)) {
                        currentBest = move;
                        bestMove = move;
                    }
                }
            }
        }
        return currentBest;
    }

    /**
     * This method returns the best move which the piece can perform
     *
     * @return
     */
    public Move getBestMove() {
        return bestMove;
    }

    /**
     * This method determines if the piece can move
     *
     * @return
     */
    public abstract boolean getCanMove();

    /**
     * This method compares this piece to another piece
     *
     * @param p
     * @return
     */
    public boolean equals(Piece p) {
        return this.colour == p.colour && this.piece == p.piece;
    }

    public void printValidMoves(boolean[][] positions) {
        System.out.println("+---+---+---+---+---+---+---+---+");
        for (int i = 0; i < positions.length; i++) {
            System.out.print("|");
            for (int j = 0; j < positions[i].length; j++) {
                if (positions[i][j] == false) {
                    System.out.print("   |");
                } else {
                    System.out.print(" T |");
                }
            }
            System.out.println("\n+---+---+---+---+---+---+---+---+");
        }
    }

}
```