```java
package Project3;

import java.util.ArrayList;

/**********************************************************************
 * CIS 163 Section 01
 * Project 3: Chess Game
 * ChessModel Class
 *
 * This class handles the model for the chess game.  Contains methods
 * for changing and updating the game state.  Implements the
 * IChessModel Class
 *
 * @author George Fayette
 * @version 3/23/2019
 **********************************************************************/
public class ChessModel implements IChessModel {
    /**
     * Private JButton array representing the chess board.
     */
    private IChessPiece[][] board;

    /**
     * Private JButton array representing the chess board.
     */
    private Player player;

    /**
     * Private JButton array representing the chess board.
     */
    private GUIcodes GUIcode;

    /**
     * Private JButton array representing the chess board.
     */
    private int numMoves;

    /**
     * Private JButton array representing the chess board.
     */
    private final int SIZE = 8;

    /**
     * Private JButton array representing the chess board.
     */
    private ArrayList<Move> moves;

    /**
     * Private JButton array representing the chess board.
     */
    private ArrayList<IChessPiece> capturedPieces, movedPieces;

    /**
     * Private JButton array representing the chess board.
     */
    private boolean upgradePawn, enPassant, castle;

    /**
     * Private JButton array representing the chess board.
     */
```

```
 61      private ArrayList<Boolean> EpHappened, castleHappened,
 62              firstPawnMoves, firstRookMoves, firstKingMoves;
 63
 64      /**************************************************************
 65       * Public constructor sets game state to initial values.
 66       **************************************************************/
 67      public ChessModel() {
 68          board = new IChessPiece[8][8];
 69          player = Player.WHITE;
 70          GUIcode = GUIcodes.NO_MESSAGE;
 71          numMoves = 0;
 72
 73          moves = new ArrayList<>();
 74          capturedPieces = new ArrayList<>();
 75          movedPieces = new ArrayList<>();
 76
 77          upgradePawn = false;
 78          enPassant = false;
 79          castle = false;
 80
 81          EpHappened = new ArrayList<>();
 82          castleHappened = new ArrayList<>();
 83          firstPawnMoves = new ArrayList<>();
 84          firstRookMoves = new ArrayList<>();
 85          firstKingMoves = new ArrayList<>();
 86
 87
 88          for (int i = 0; i < SIZE; ++i) {
 89              board[6][i] = new Pawn(Player.WHITE);
 90          }
 91          board[7][0] = new Rook(Player.WHITE);
 92          board[7][1] = new Knight(Player.WHITE);
 93          board[7][2] = new Bishop(Player.WHITE);
 94          board[7][3] = new Queen(Player.WHITE);
 95          board[7][4] = new King(Player.WHITE);
 96          board[7][5] = new Bishop(Player.WHITE);
 97          board[7][6] = new Knight(Player.WHITE);
 98          board[7][7] = new Rook(Player.WHITE);
 99
100
101          for (int i = 0; i < SIZE; ++i) {
102              board[1][i] = new Pawn(Player.BLACK);
103          }
104          board[0][0] = new Rook(Player.BLACK);
105          board[0][1] = new Knight(Player.BLACK);
106          board[0][2] = new Bishop(Player.BLACK);
107          board[0][3] = new Queen(Player.BLACK);
108          board[0][4] = new King(Player.BLACK);
109          board[0][5] = new Bishop(Player.BLACK);
110          board[0][6] = new Knight(Player.BLACK);
111          board[0][7] = new Rook(Player.BLACK);
112
113      }
114
115      /**************************************************************
116       * Public boolean, returns true if the move is valid.
117       * @param move The move that is being checked.
118       * @return True if the move is valid.
119       **************************************************************/
120      public boolean isValidMove(Move move) {
```

```
121         IChessPiece moveFrom = board[move.fromRow][move.fromColumn];
122
123         if (board[move.fromRow][move.fromColumn] == null) {
124             enPassant = false;
125             castle = false;
126             return false;
127         }
128
129         if (board[move.fromRow][move.fromColumn]
130                 .isValidMove(move, board)) {
131             enPassant = false;
132             castle = false;
133             return true;
134         }
135
136         if (numMoves > 0 && moveFrom.type().equals("Pawn") &&
137                 isValidEnPassant(move)) {
138             enPassant = true;
139             castle = false;
140             return true;
141         }
142
143         if (moveFrom.type().equals("King") && isValidCastle(move)) {
144             enPassant = false;
145             castle = true;
146             return true;
147         }
148
149         enPassant = false;
150         castle = false;
151         return false;
152     }
153
154     /*****************************************************************
155      * Public boolean, returns true if the move has been made and
156      * does not put the player into check.
157      * @param move The move that is being tried.
158      * @return True if the move has been made and does not put the
159      * player into check.
160      *****************************************************************/
161     public boolean tryMove(Move move) {
162         if (isValidMove(move)) {
163             move(move);
164             if (!inCheck(board[move.toRow][move.toColumn].player())) {
165                 return true;
166             }
167             undo();
168         }
169         return false;
170     }
171
172     // Checks to see if a move is a valid En Passant.  Returns true
173     // if valid.
174     private boolean isValidEnPassant(Move move) {
175         Move lastMove = moves.get(numMoves - 1);
176         if (board[lastMove.toRow][lastMove.toColumn].type()
177                 .equals("Pawn")) {
178
179             if (lastMove.fromRow - lastMove.toRow == 2) {
180
```

```
181                 board[lastMove.fromRow - 1][lastMove.toColumn] =
182                         new Pawn(player.next());
183
184                 if (board[move.fromRow][move.fromColumn]
185                         .isValidMove(move, board)) {
186                     board[lastMove.fromRow - 1][lastMove.toColumn] =
187                             null;
188                     return true;
189                 }
190                 board[lastMove.fromRow - 1][lastMove.toColumn] = null;
191             }
192
193             if (lastMove.fromRow - lastMove.toRow == -2) {
194
195                 board[lastMove.fromRow + 1][lastMove.toColumn] =
196                         new Pawn(player.next());
197
198                 if (board[move.fromRow][move.fromColumn]
199                         .isValidMove(move, board)) {
200                     board[lastMove.fromRow + 1][lastMove.toColumn] =
201                             null;
202                     return true;
203                 }
204                 board[lastMove.fromRow + 1][lastMove.toColumn] = null;
205             }
206         }
207         return false;
208     }
209
210     // Checks to see if a move is a valid Castle.  Returns true if
211     // valid.
212     private boolean isValidCastle(Move move) {
213         if (move.fromRow == move.toRow &&
214                 ((King) board[move.fromRow][move.fromColumn]).firstMove) {
215             if (move.fromColumn - move.toColumn == 2) {
216                 for (int i = move.fromColumn - 1; i >= 1; --i) {
217                     if (board[move.fromRow][i] != null) {
218                         return false;
219                     }
220                 }
221                 if (board[move.fromRow][0] != null &&
222                         board[move.fromRow][0].type().equals("Rook")) {
223                     if (((Rook) board[move.fromRow][0]).firstMove) {
224                         return true;
225                     }
226                 }
227             }
228
229             if (move.fromColumn - move.toColumn == -2) {
230                 for (int i = move.fromColumn + 1; i <= 6; ++i) {
231                     if (board[move.fromRow][i] != null) {
232                         return false;
233                     }
234                 }
235                 if (board[move.fromRow][7] != null &&
236                         board[move.fromRow][7].type().equals("Rook")) {
237                     if (((Rook) board[move.fromRow][7]).firstMove) {
238                         return true;
239                     }
240                 }
```

```
241             }
242         }
243         return false;
244     }
245 
246     /*****************************************************************
247      * This method executes a given move and stores values for
248      * undoing the move.
249      * @param move The move that is executed.
250      *****************************************************************/
251     public void move(Move move) {
252         IChessPiece moveFrom = board[move.fromRow][move.fromColumn];
253         IChessPiece moveTo = board[move.toRow][move.toColumn];
254         isValidMove(move);
255 
256         if (moveFrom.type().equals("Pawn")) {
257 
258             if (((Pawn) moveFrom).firstMove) {
259                 firstPawnMoves.add(true);
260                 ((Pawn) moveFrom).firstMove = false;
261             } else {
262                 firstPawnMoves.add(false);
263             }
264 
265 
266             if (move.toRow == 0 || move.toRow == 7) {
267                 upgradePawn = true;
268             } else {
269                 upgradePawn = false;
270             }
271 
272 
273         } else {
274             firstPawnMoves.add(false);
275             upgradePawn = false;
276         }
277 
278         if (moveFrom.type().equals("Rook")) {
279             if (((Rook) moveFrom).firstMove) {
280                 firstRookMoves.add(true);
281                 ((Rook) moveFrom).firstMove = false;
282             } else {
283                 firstRookMoves.add(false);
284             }
285         } else {
286             firstRookMoves.add(false);
287         }
288 
289         if (moveFrom.type().equals("King")) {
290             if (((King) moveFrom).firstMove) {
291                 firstKingMoves.add(true);
292                 ((King) moveFrom).firstMove = false;
293             } else {
294                 firstKingMoves.add(false);
295             }
296         } else {
297             firstKingMoves.add(false);
298         }
299 
300         if (enPassant) {
```

```
301             board[move.fromRow][move.toColumn] = null;
302             EpHappened.add(true);
303         } else {
304             EpHappened.add(false);
305         }
306 
307         if (castle) {
308             if (move.fromColumn > move.toColumn) {
309                 board[move.toRow][move.toColumn + 1] =
310                         board[move.toRow][0];
311                 board[move.toRow][0] = null;
312             } else {
313                 board[move.toRow][move.toColumn - 1] =
314                         board[move.toRow][7];
315                 board[move.toRow][7] = null;
316             }
317             castleHappened.add(true);
318         } else {
319             castleHappened.add(false);
320         }
321 
322         moves.add(numMoves, move);
323         capturedPieces.add(numMoves, moveTo);
324         movedPieces.add(numMoves, moveFrom);
325 
326         ++numMoves;
327         board[move.toRow][move.toColumn] =
328                 board[move.fromRow][move.fromColumn];
329         board[move.fromRow][move.fromColumn] = null;
330         setNextPlayer();
331     }
332 
333     /**************************************************************
334      * This method updates GUIcode to reflect the current game status.
335      *************************************************************/
336     public void updateStatus() {
337         if (upgradePawn) {
338             GUIcode = GUIcodes.UPGRADE;
339         } else if (isCheckmate()) {
340             GUIcode = GUIcodes.CHECKMATE;
341         } else if (isDraw()) {
342             GUIcode = GUIcodes.DRAW;
343         } else if (inCheck(player)) {
344             GUIcode = GUIcodes.IN_CHECK;
345         } else {
346             GUIcode = GUIcodes.NO_MESSAGE;
347         }
348     }
349 
350     /**************************************************************
351      * This method promotes a pawn to the piece type in the String
352      * parameter.
353      * @param piece The type of piece that the pawn should be
354      *              promoted to.
355      *************************************************************/
356     public void upgradePawn(String piece) {
357         int r = moves.get(numMoves - 1).toRow;
358         int c = moves.get(numMoves - 1).toColumn;
359         if (piece.equals("Rook")) {
360             board[r][c] = new Rook(player.next());
```

```
361             ((Rook) board[r][c]).firstMove = false;
362         } else if (piece.equals("Knight")) {
363             board[r][c] = new Knight(player.next());
364         } else if (piece.equals("Bishop")) {
365             board[r][c] = new Bishop(player.next());
366         } else if (piece.equals("Queen")) {
367             board[r][c] = new Queen(player.next());
368         }
369         upgradePawn = false;
370     }
371 
372     /**************************************************************
373      * This method undoes the previous move.
374      **************************************************************/
375     public void undo() {
376         if (numMoves > 0) {
377             Move lastMove = moves.get(numMoves - 1);
378             board[lastMove.fromRow][lastMove.fromColumn] =
379                     movedPieces.get(numMoves - 1);
380             board[lastMove.toRow][lastMove.toColumn] =
381                     capturedPieces.get(numMoves - 1);
382             IChessPiece moveFrom =
383                     board[lastMove.fromRow][lastMove.fromColumn];
384 
385             if (moveFrom.type().equals("Pawn") &&
386                     firstPawnMoves.get(numMoves - 1)) {
387                 ((Pawn) moveFrom).firstMove = true;
388             }
389 
390             if (moveFrom.type().equals("Rook") &&
391                     firstRookMoves.get(numMoves - 1)) {
392                 ((Rook) moveFrom).firstMove = true;
393             }
394 
395             if (moveFrom.type().equals("King") &&
396                     firstKingMoves.get(numMoves - 1)) {
397                 ((King) moveFrom).firstMove = true;
398             }
399 
400             if (EpHappened.get(numMoves - 1)) {
401                 board[lastMove.fromRow][lastMove.toColumn] =
402                         new Pawn(player);
403                 ((Pawn) board[lastMove.fromRow][lastMove.toColumn]).firstMove =
404                         false;
405             }
406 
407             if (castleHappened.get(numMoves - 1)) {
408                 if (lastMove.fromColumn > lastMove.toColumn) {
409                     board[lastMove.toRow][0] =
410                             board[lastMove.toRow][lastMove.toColumn +
411                                     1];
412                     board[lastMove.toRow][lastMove.toColumn + 1] = null;
413                 } else {
414                     board[lastMove.toRow][7] =
415                             board[lastMove.toRow][lastMove.toColumn -
416                                     1];
417                     board[lastMove.toRow][lastMove.toColumn - 1] = null;
418                 }
419             }
420 
```

```
421             --numMoves;
422             moves.remove(numMoves);
423             capturedPieces.remove(numMoves);
424             movedPieces.remove(numMoves);
425             firstPawnMoves.remove(numMoves);
426             firstRookMoves.remove(numMoves);
427             firstKingMoves.remove(numMoves);
428             EpHappened.remove(numMoves);
429             castleHappened.remove(numMoves);
430             numMoves = moves.size();
431             setNextPlayer();
432         }
433     }
434 
435     /**************************************************************
436      * This method checks to see if there is a checkmate.
437      * @return True if game status is checkmate.
438      **************************************************************/
439     public boolean isCheckmate() {
440         if (inCheck(player) && isComplete()) {
441             return true;
442         }
443         return false;
444     }
445 
446     /**************************************************************
447      * This method checks to see if there is a draw.
448      * @return True if game status is draw.
449      **************************************************************/
450     public boolean isDraw() {
451         if (!inCheck(player) && isComplete()) {
452             return true;
453         }
454         return false;
455     }
456 
457     /**************************************************************
458      * This method checks to see if the game is over.
459      * @return True if the game is over.
460      **************************************************************/
461     public boolean isComplete() {
462         for (int r = 0; r < SIZE; ++r) {
463             for (int c = 0; c < SIZE; ++c) {
464                 if (board[r][c] != null &&
465                         board[r][c].player() == player) {
466                     Move m = new Move(r, c, 0, 0);
467                     if (outOfCheckMove(m) != null) {
468                         return false;
469                     }
470                 }
471             }
472         }
473         return true;
474     }
475 
476     //tries all possible moves for a given piece to see if that move
477     // will take the player out of check.
478     private Move outOfCheckMove(Move m) {
479         for (int r = 0; r < SIZE; ++r) {
480             m.toRow = r;
```

```
481             for (int c = 0; c < SIZE; ++c) {
482                 m.toColumn = c;
483                 if (tryMove(m)) {
484                     undo();
485                     return m;
486                 }
487             }
488         }
489         return null;
490     }
491 
492     /**************************************************************
493      * This method checks to see if a player is in check.
494      * @param p The player that is possibly in check.
495      * @return True if the player is in check.
496      *************************************************************/
497     public boolean inCheck(Player p) {
498         boolean inCheck = false;
499         int rKing = 0;
500         int cKing = 0;
501         for (int r = 0; r < SIZE; ++r) {
502             for (int c = 0; c < SIZE; ++c) {
503                 if (board[r][c] != null && board[r][c].player() == p &&
504                         board[r][c].type().equals("King")) {
505                     rKing = r;
506                     cKing = c;
507                 }
508             }
509         }
510 
511         for (int r = 0; r < SIZE; ++r) {
512             for (int c = 0; c < SIZE; ++c) {
513                 if (board[r][c] != null &&
514                         board[r][c].player() == p.next()) {
515                     Move capMove = new Move(r, c, rKing, cKing);
516                     if (isValidMove(capMove)) {
517                         inCheck = true;
518                     }
519 
520                 }
521             }
522         }
523         return inCheck;
524     }
525 
526     /**************************************************************
527      * This method returns the number of moves for the current game.
528      * @return The number of moves for the game.
529      *************************************************************/
530     public int numMoves() {
531         return numMoves;
532     }
533 
534     /**************************************************************
535      * This method returns the GUIcode for the game.
536      * @return The GUIcode for the game.
537      *************************************************************/
538     public GUIcodes GUIcode() {
539         return GUIcode;
540     }
```

```
541
542     /**************************************************************
543      * This method returns the current player in the game.
544      * @return The current player in the game.
545      **************************************************************/
546     public Player currentPlayer() {
547         return player;
548     }
549
550     /**************************************************************
551      * This method returns the number of rows on the board.
552      * @return The number of rows on the board.
553      **************************************************************/
554     public int numRows() {
555         return SIZE;
556     }
557
558     /**************************************************************
559      * This method returns the number of columns on the board.
560      * @return The number of columns on the board.
561      **************************************************************/
562     public int numColumns() {
563         return SIZE;
564     }
565
566     /**************************************************************
567      * This method returns the IChessPiece at the requested location.
568      * @param row The row of the IChessPiece
569      * @param column The column of the IChessPiece
570      * @return The number of moves for the game.
571      **************************************************************/
572     public IChessPiece pieceAt(int row, int column) {
573         return board[row][column];
574     }
575
576     /**************************************************************
577      * This method sets the next player for the game.
578      **************************************************************/
579     public void setNextPlayer() {
580         player = player.next();
581     }
582
583     /**************************************************************
584      * This method places a piece at a given location on the board.
585      * @param row The row that the piece is being placed.
586      * @param column The Column that the piece is being placed.
587      * @param piece The IChessPiece that is being placed.
588      **************************************************************/
589     public void setPiece(int row, int column, IChessPiece piece) {
590         board[row][column] = piece;
591     }
592
593     /**************************************************************
594      * This method clears the game board by setting all tiles to null.
595      **************************************************************/
596     public void clearBoard() {
597         for (int r = 0; r < SIZE; ++r) {
598             for (int c = 0; c < SIZE; ++c) {
599                 board[r][c] = null;
600             }
```

```
601         }
602     }
603
604     // This method attempts to find a checkmate move.
605     private boolean moveCheckmate() {
606         for (int r = 0; r < SIZE; ++r) {
607             for (int c = 0; c < SIZE; ++c) {
608                 if (board[r][c] != null &&
609                         board[r][c].player() == player) {
610                     for (int r2 = 0; r2 < SIZE; ++r2) {
611                         for (int c2 = 0; c2 < SIZE; ++c2) {
612                             Move m = new Move(r, c, r2, c2);
613                             if (tryMove(m)) {
614                                 if (isCheckmate()) {
615                                     return true;
616                                 } else {
617                                     undo();
618                                 }
619                             }
620                         }
621                     }
622                 }
623             }
624         }
625         return false;
626     }
627
628     // This method attempts to put the opponent into check.
629     private boolean moveCheck() {
630         for (int r = 0; r < SIZE; ++r) {
631             for (int c = 0; c < SIZE; ++c) {
632                 if (board[r][c] != null &&
633                         board[r][c].player() == player) {
634                     for (int r2 = 0; r2 < SIZE; ++r2) {
635                         for (int c2 = 0; c2 < SIZE; ++c2) {
636                             Move m = new Move(r, c, r2, c2);
637                             if (tryMove(m)) {
638                                 if (inCheck(player) &&
639                                         !pieceInDanger(r2, c2,
640                                                 player.next())) {
641                                     return true;
642                                 } else {
643                                     undo();
644                                 }
645                             }
646                         }
647                     }
648                 }
649             }
650         }
651         return false;
652     }
653
654     // This method checks to see if a piece is in danger of being
655     // captured.
656     private boolean pieceInDanger(int row, int col, Player p) {
657         for (int r = 0; r < SIZE; ++r) {
658             for (int c = 0; c < SIZE; ++c) {
659                 if (board[r][c] != null &&
660                         board[r][c].player() == p.next()) {
```

```
661                     Move m = new Move(r, c, row, col);
662                     if (tryMove(m)) {
663                         undo();
664                         return true;
665                     }
666
667                 }
668             }
669         }
670         return false;
671     }
672
673
674     // This method looks for a move that will take the piece out of
675     // danger.
676     private boolean moveOutOfDanger(int row, int col) {
677         for (int r = 0; r < SIZE; ++r) {
678             for (int c = 0; c < SIZE; ++c) {
679                 Move m = new Move(row, col, r, c);
680                 if (tryMove(m)) {
681                     if (!pieceInDanger(r, c, player.next())) {
682                         return true;
683                     } else {
684                         undo();
685                     }
686                 }
687             }
688         }
689         return false;
690     }
691
692     // This method looks for a capture move that will take the piece out
693     // of danger.
694     private boolean moveOutOfDangerCapture(int row, int col) {
695         for (int r = 0; r < SIZE; ++r) {
696             for (int c = 0; c < SIZE; ++c) {
697                 if (board[r][c] != null &&
698                         board[r][c].player() == player.next()) {
699                     Move m = new Move(row, col, r, c);
700                     if (tryMove(m)) {
701                         if (!pieceInDanger(r, c, player.next())) {
702                             return true;
703                         } else {
704                             undo();
705                         }
706                     }
707                 }
708             }
709         }
710         return false;
711     }
712
713     // This method attempts to protect a piece that is in danger of
714     // being captured.
715     private boolean protectPiece() {
716         for (int r = 0; r < SIZE; ++r) {
717             for (int c = 0; c < SIZE; ++c) {
718                 if (board[r][c] != null &&
719                         board[r][c].player() == player) {
720                     if (pieceInDanger(r, c, player)) {
```

```
721                         if (moveOutOfDanger(r, c)) {
722                             return true;
723                         }
724                     }
725                 }
726             }
727         }
728         return false;
729     }
730
731     // This method attempts to protect a piece that is in danger of
732     // being captured by capturing an opponent's piece.
733     private boolean protectPieceCapture() {
734         for (int r = 0; r < SIZE; ++r) {
735             for (int c = 0; c < SIZE; ++c) {
736                 if (board[r][c] != null &&
737                         board[r][c].player() == player) {
738                     if (pieceInDanger(r, c, player)) {
739                         if (moveOutOfDangerCapture(r, c)) {
740                             return true;
741                         }
742                     }
743                 }
744             }
745         }
746         return false;
747     }
748
749     // This method looks for a move that will not place the piece in
750     // danger.
751     private boolean safeMove(int r, int c) {
752         for (int r2 = 0; r2 < SIZE; ++r2) {
753             for (int c2 = 0; c2 < SIZE; ++c2) {
754                 Move m = new Move(r, c, r2, c2);
755                 if (tryMove(m)) {
756                     if (!pieceInDanger(r2, c2, player.next())) {
757                         return true;
758                     } else {
759                         undo();
760                     }
761                 }
762             }
763         }
764         return false;
765     }
766
767     // This method looks for a capture move that will not place the
768     // piece in danger.
769     private boolean safeCaptureMove(int r, int c) {
770         for (int r2 = 0; r2 < SIZE; ++r2) {
771             for (int c2 = 0; c2 < SIZE; ++c2) {
772                 if (board[r2][c2] != null &&
773                         board[r2][c2].player() == player.next()) {
774                     Move m = new Move(r, c, r2, c2);
775                     if (tryMove(m)) {
776                         if (!pieceInDanger(r2, c2, player.next())) {
777                             return true;
778                         } else {
779                             undo();
780                         }
```

```
781                     }
782                 }
783             }
784         }
785         return false;
786     }
787
788     // This method attempts to make a safe move.
789     private boolean movePieceSafe() {
790         for (int r = 0; r < SIZE; ++r) {
791             for (int c = 0; c < SIZE; ++c) {
792                 if (board[r][c] != null &&
793                         board[r][c].player() == player) {
794                     if (safeMove(r, c)) {
795                         return true;
796                     }
797                 }
798             }
799         }
800         return false;
801     }
802
803     // This method attempts to make a safe capture move.
804     private boolean movePieceSafeCapture() {
805         for (int r = 0; r < SIZE; ++r) {
806             for (int c = 0; c < SIZE; ++c) {
807                 if (board[r][c] != null &&
808                         board[r][c].player() == player) {
809                     if (safeCaptureMove(r, c)) {
810                         return true;
811                     }
812                 }
813             }
814         }
815         return false;
816     }
817
818     // This method attempts to make a safe pawn move.
819     private boolean movePawnSafe() {
820         for (int r = 0; r < SIZE; ++r) {
821             for (int c = 0; c < SIZE; ++c) {
822                 if (board[r][c] != null &&
823                         board[r][c].player() == player &&
824                         board[r][c].type().equals("Pawn")) {
825                     if (safeMove(r, c)) {
826                         return true;
827                     }
828                 }
829             }
830         }
831         return false;
832     }
833
834     // This method attempts to make a safe capture move for a pawn.
835     private boolean movePawnSafeCapture() {
836         for (int r = 0; r < SIZE; ++r) {
837             for (int c = 0; c < SIZE; ++c) {
838                 if (board[r][c] != null &&
839                         board[r][c].player() == player &&
840                         board[r][c].type().equals("Pawn")) {
```

```
841                     if (safeCaptureMove(r, c)) {
842                         return true;
843                     }
844                 }
845             }
846         }
847         return false;
848     }
849 
850     // This method attempts find a valid move.  May place the moved
851     // piece in danger.
852     private boolean movePieceUnsafe() {
853         for (int r = 0; r < SIZE; ++r) {
854             for (int c = 0; c < SIZE; ++c) {
855                 if (board[r][c] != null &&
856                         board[r][c].player() == player) {
857                     for (int r2 = 0; r2 < SIZE; ++r2) {
858                         for (int c2 = 0; c2 < SIZE; ++c2) {
859                             Move m = new Move(r, c, r2, c2);
860                             if (tryMove(m)) {
861                                 return true;
862                             }
863                         }
864                     }
865                 }
866 
867             }
868         }
869         return false;
870     }
871 
872 
873     /*****************************************************************
874      * This method preforms an AI move for the game.
875      *****************************************************************/
876     public void AI() {
877         if (GUIcode != GUIcodes.CHECKMATE && GUIcode != GUIcodes.DRAW) {
878             if (moveCheckmate()) {
879                 System.out.println("move checkmate");
880             } else if (moveCheck()) {
881                 System.out.println("move check");
882             } else if (protectPieceCapture()) {
883                 System.out.println("protect piece capture");
884             } else if (protectPiece()) {
885                 System.out.println("protect piece");
886             } else if (movePawnSafeCapture()) {
887                 System.out.println("move pawn safe capture");
888             } else if (movePieceSafeCapture()) {
889                 System.out.println("move piece safe capture");
890             } else if (movePawnSafe()) {
891                 System.out.println("move pawn safe");
892             } else if (movePieceSafe()) {
893                 System.out.println("move piece safe");
894             } else if (movePieceUnsafe()) {
895                 System.out.println("move piece unsafe");
896             }
897         }
898     }
899 }
```