```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package fourplay;

import javax.swing.Timer;
import java.util.Observer;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Random;
/**
 *
 * @author User
 */
public class CPUPlayer implements ActionListener {

    private int playerNumber;
    private FPModel gameModel;
    private FPController gameController;
    private Timer moveTimer;

    private static final int TIME_INTERVAL = 500;

    public CPUPlayer(FPModel gameModel,FPController gameController, int playerNumber){
        this.gameModel=gameModel;
        this.gameController=gameController;
        this.playerNumber=playerNumber;

        moveTimer = new Timer(TIME_INTERVAL,this);
        moveTimer.setInitialDelay(TIME_INTERVAL);

    }

    public void setController(FPController gameController){
        this.gameController=gameController;
    }

    /**
     * This method starts a timer which once complete calls
     * requestMove().
     */
    public void doMove(){
        moveTimer.start();
    }

    /**
     * This method is call to actually perform a move. If the
     * board is empty it sets a random piece. If the board is not empty
     * it calls the bestMove() method to determine what the best move would
     * be. It then performs this move.
     */
    private void requestMove(){
```

```
        //If board is empty place a random piece
        if(gameModel.boardIsEmpty()){
            Random randomGenerator = new Random();
            int randomCol = randomGenerator.nextInt(gameModel.getNoOfColomns()-1);
            gameModel.setPiece(randomCol, playerNumber);
            gameController.togglePlayer();
            return;
        }

        //Determine which colomn click would produce the best move
        int bestMove = bestMove(playerNumber);
        if(bestMove>=0){
            gameModel.setPiece(bestMove, playerNumber);
        }
        gameController.checkForWinner();
        gameController.togglePlayer();
    }


    /**
     * This method determines which colomn click would result
     * in the best move. It does this by determining the heuristic
     * result for each possible move and picking the move with the
     * highest heuristic value.
     *
     * @param player    The player to determine the best move for.
     * @return          The best colomn to click on (0-gameModel.getNoOfColomns
     */
    public int bestMove(int player){
        int bestSoFar=0;
        int bestCol=-1;
        int currentValue;
        //Check the heuristic value of each possible move
        for(int i=0; i <gameModel.getNoOfColomns();i++){
            if(gameModel.validMove(gameModel.getNoOfRows()-1,i)){
                currentValue=heuristicResult(i,player);
                if(currentValue >=bestSoFar){
                    bestSoFar=currentValue;
                    bestCol=i;
                }
            }
        }
        return bestCol;
    }

    /**
     * Determines the value of a given move for a given player
     *
     * @param col       The colomn the piece would fall in
     * @param player    The player the piece belongs to
     * @return          A value depending of the actual value of a move,
     *                  the larger the value the better.
     */
    public int  heuristicResult(int col,int player){
        int row=0;
        int orthoResult,diagResult;
        int totalResult = 0;
```

```
        //Copys board status to do hypothetical moves
        int[][]hypoBoardStatus = new int[gameModel.getNoOfRows()][gameModel.getNoOfColomns()];
        for(int i = 0; i < gameModel.getNoOfRows(); i++){
            System.arraycopy(gameModel.getChipStatus()[i], 0, hypoBoardStatus[i], 0,
            gameModel.getNoOfColomns());
        }

        //Decides where the hypothetical chip would land for a give colomn so derives a row.
        //Also actually places the piece on hypoBoardStatus[][]
        for(int k =0 ; k < gameModel.getNoOfRows(); k++){
            if(hypoBoardStatus[k][col]==0){
                hypoBoardStatus[k][col]=player;
                row=k;
                break;
            }
        }
        //Checks the value of that move in terms of orthogonal lines
        orthoResult = heuristicOrtho(hypoBoardStatus,false,row,col,player)+heuristicOrtho(
        hypoBoardStatus,true,row,col,player);
        //Checks the value of that move in terms of diagonal lines
        diagResult = heuristicDiag(hypoBoardStatus,true,row,col,player)+heuristicDiag(
        hypoBoardStatus,false,row,col,player);

        totalResult = orthoResult+diagResult;
        return totalResult;
    }

    /**
     * This method determine the value in terms of game play for a given move. It splits
     * a row or colomn into blocks of 4 and determines the value of each block of 4. It
     * then sums the blocks together. The value of each block is determined by the method
     * blockScore().
     *
     * @param hypoBoardStatus   An int[][] representing the current state of the board
     *                          + the hypothetical move.
     * @param horizontal        Boolean, if true the method works in rows else colomns
     * @param row               Row of the hypothetical piece
     * @param col               Col of the hypothetical piece
     * @param player            Player of the hypothetical move
     * @return                  Value of the hypothetical move
     */
    public int heuristicOrtho(int[][] hypoBoardStatus, boolean horizontal,int row,int col,int
     player){

        int opponentPlayer;
        int noOfBlocks;

        if(horizontal)
            noOfBlocks=(gameModel.getNoOfColomns()-4)+1;
        else
            noOfBlocks=(gameModel.getNoOfRows()-4)+1;

        int[][] blocks = new int[noOfBlocks][4];
        int result=0;

        if(player==1){
```

```
            opponentPlayer=2;
        }else{
            opponentPlayer=1;
        }

        for(int i = 0; i < noOfBlocks; i++){
            for(int j =0; j<4;j++){
                if(horizontal){

                    blocks[i][j]=hypoBoardStatus[row][i+j];
                }else{
                    blocks[i][j]=hypoBoardStatus[i+j][col];
                }
            }
        }

        for(int k = 0; k < noOfBlocks; k++){
            if(containsOpponent(blocks[k],opponentPlayer))
                result=result+0;
            else
                result=result+blockScore(blocks[k],player);
        }
        return result;
    }

    /**
     * This method determine the value in terms of game play for a given move. It splits
     * a diagonal line into blocks of 4 and determines the value of each block of 4. It
     * then sums the blocks together. The value of each block is determined by the method
     * blockScore().
     *
     * @param hypoBoardStatus   An int[][] representing the current state of the board
     *                          + the hypothetical move.
     * @param negGrad           Boolean, true to check for negative gradient diagonals,
     *                          false to check for positive gradient diagonals.
     * @param row               Row of the hypothetical piece
     * @param col               Col of the hypothetical piece
     * @param player            Player of the hypothetical move
     * @return                  Value of the hypothetical move
     */
    public int heuristicDiag(int[][] hypoBoardStatus,boolean negGrad,int row,int col,int
    player){

        int firstDiagCoordRow, firstDiagCoordCol,lastDiagCoordRow,lastDiagCoordCol;
        int rowRes,colRes,length,opponentPlayer;
        int result = 0;

        if(player==1){
            opponentPlayer=2;
        }else{
            opponentPlayer=1;
        }

        if(negGrad){
            rowRes=gameModel.getNoOfRows()-(row+1);
            if(rowRes < col){
                firstDiagCoordRow=row+rowRes;
```

```
                firstDiagCoordCol=col-rowRes;
            }else{
                firstDiagCoordRow=row+col;
                firstDiagCoordCol=col-col;
            }

            colRes=gameModel.getNoOfColomns()-(firstDiagCoordCol+1);
            if(colRes < firstDiagCoordRow){
                lastDiagCoordRow=firstDiagCoordRow-colRes;
                lastDiagCoordCol=firstDiagCoordCol+colRes;
            }else{
                lastDiagCoordRow=firstDiagCoordRow-firstDiagCoordRow;
                lastDiagCoordCol=firstDiagCoordCol+firstDiagCoordRow;
            }
        }else{

            if(col < row){
                firstDiagCoordRow=row-col;
                firstDiagCoordCol=col-col;
            }else{
                firstDiagCoordRow=row-row;
                firstDiagCoordCol=col-row;
            }
            colRes=gameModel.getNoOfColomns()-(col+1);
            rowRes=gameModel.getNoOfRows()-(row+1);
            if(colRes < rowRes){
                lastDiagCoordRow=row+colRes;
                lastDiagCoordCol=col+colRes;
            }else{
                lastDiagCoordRow=row+rowRes;
                lastDiagCoordCol=col+rowRes;
            }

        }
        if(negGrad)
            length = (firstDiagCoordRow-lastDiagCoordRow)+1;
        else
            length = (lastDiagCoordRow-firstDiagCoordRow)+1;

        int noOfBlocks = length-3;
        if(noOfBlocks < 0)
            noOfBlocks=0;

        int[][] blocks = new int[noOfBlocks][4];

        for(int i = 0; i < noOfBlocks; i++){
            for(int j =0; j<4;j++){
                if(negGrad){
                    blocks[i][j]=hypoBoardStatus[firstDiagCoordRow-(j+i)][firstDiagCoordCol+(
                    j+i)];
                }else{
                    blocks[i][j]=hypoBoardStatus[firstDiagCoordRow+(j+i)][firstDiagCoordCol+(
                    j+i)];
                }
            }
        }
        for(int k = 0; k < noOfBlocks; k++){
```

```
            if(containsOpponent(blocks[k],opponentPlayer))
                result=result+0;
            else
                result=result+blockScore(blocks[k],player);
        }
        return result;
    }
    /**
     * This method returns a score for a block of pieces where:
     * 1 piece = score of 1
     * 2 consecutive pieces = score of 4
     * 3 consecutive pieces = score of 32
     * 4 consecutive pieces = score of 128 (this is a win)
     *
     * @param subject   The block to examine
     * @param player    The player in question
     * @return          The total score for the block
     */
    public int blockScore(int[] subject, int player){
        int counter=0;
        int result=0;

        for(int i = 0; i < subject.length; i++){
            if(subject[i]==player){
                switch (counter){
                    case 0: counter++;
                        break;
                    case 1: counter = counter +3;
                        break;
                    case 4: counter = counter +28;
                        break;
                    case 32: counter = counter +96;
                        break;
                    default: counter = -999;
                        break;
                }

            }else{
                result = result+counter;
                counter = 0;
            }
        }
        return result+counter;
    }


    /**
     * This method checks a block of pieces and if there is a piece
     * from the opponent it returns true, else false.
     *
     * @param subject   The block to examine
     * @param opponent  The opponent player
     * @return          True if the opponents chip is found
     */
    public boolean containsOpponent(int[] subject, int opponent){
        for(int i= 0; i < subject.length; i++){
            if(subject[i]== opponent)
```

```
                return true;
            }
            return false;
        }

        public void actionPerformed(ActionEvent event){
            if(event.getSource()==moveTimer){
                requestMove();
                moveTimer.stop();
            }
        }
}
```