ROBUT

**MIS**

**robut.java**

This module acts as an Artificial Intelligence to determine what move the computer should make. It is used by GameDisplay.java to give the computer’s next move based on a given board setup.

Interface Uses

-none

Return Type

-integer array describing where the computer should move

Access Programs

-public static int[] place(int[][] visibleTeams, int colour, int added) : A public method that, based on the setup of the board and the number of pieces on it, determines and returns a location on the board where a new piece should be placed.

-public static int[] move(int[][] visibleTeams, int colour) : A public method that, based on the setup of the board, determines and returns a set of locations on the board of what piece should be moved and where it should be moved to.

-public static int[] mill(int[][] visibleTeams, int colour, int added) : A public method that, based on the setup of the board determines and returns the location of a piece on the board that should be removed.

**GameData.java**

This module acts as the module in MVC framework, and stores and returns the information for the game currently in play. It is used by GameDisplay.java and Gameboard.java to return and update game data, and by FileIO.java return game data to be saved.

Interface Uses

-FileIO.java

Return Type

-information on the game currently being played

Access Programs

-public GameData() : A constructor for GameData class. Allows another class to create Game Data for a new game.

-public int[][] getVisibleTeams() : A getter method for the locations of the pieces on the game board.

-public void setVisibleTeams(int x, int y, int value) : A setter method for an individual value of visibleTeams. Allows a piece to be added to the gameboard.

-public void setVisibleTeams(int[][] newTeams) : A setter method for the whole of visibleTeams. Allows for an old game to be loaded to the game board.

-public void incrementPlay1count() : Increments the number of pieces played by player 1 by 1.

-public void decreasePlay1count() : Decrements the number of pieces played by player 1 by 1.

-public void setPlay1count(int count) : A setter method for the number of pieces played by player 1. Allows for an old game’s count of player 1’s pieces to be loaded.

-public int getPlay1count() : A getter method for the number of pieces played by player 1.

-public int getPlay1removed() : A getter method for the number of player 1’s pieces that were removed.

-public void incrementPlay2count() : Increments the number of pieces played by player 2 by 1.

-public void decreasePlay2count() : Decrements the number of pieces played by player 2 by 1.

-public void setPlay2count(int count) : A setter method for the number of pieces played by player 2. Allows for an old game’s count of player 2’s pieces to be loaded.

-public int getPlay2count() : A getter method for the number of pieces played by player 2.

-public int getPlay2removed() : A getter method for the number of player 2’s pieces that were removed.

-public void setRedTake(bolean value) : A setter method for whether or not red can take one of blue’s pieces.

-public Boolean getRedTake() : A getter method for whether or not red can take one of blue’s pieces.

-public void setBlueTake(bolean value) : A setter method for whether or not blue can take one of red’s pieces.

-public Boolean getBlueTake() : A getter method for whether or not blue can take one of red’s pieces.

-public void incrementTake() : A method that switches which player is active.

-public void setFirst(boolean bool) : A setter method that sets which player moves first.

-public boolean getFirst() : A getter method that returns which player moves first.

-public void setState(int state) : A setter method that sets the previous state to the current state and sets the current state to the new state.

-public int getCurrentState() : A getter method that returns the current state of the game.

-public int getPreviousState() : A getter method that returns the previous state of the game.

-public int getPreviousMoves() : A getter method that returns the number of moves previously made.

-public void resetPreviousMoves() : A method that sets the number of previous moves to 0.

-public void incrementPreviousMoves() : A method that increases the number of previous moves by 1.

-public void loadGame() : A method that loads a game state from a txt file.

**MID**

**robut.java**

Variables

-none

Access Programs

-public static int[] place(int[][] visibleTeams, int colour, int added) : A public method taking 3 inputs: visibleTeams, an integer array of the setup of the board; colour, an integer of what which team’s piece is being placed; added, an integer of the number of pieces added to the board. Determines and returns a location on the board where a new piece should be placed.

-public static int[] move(int[][] visibleTeams, int colour) : A public method taking 2 inputs: visibleTeams, an integer array of the setup of the board; colour, an integer of what which team’s piece is being placed. Determines and returns a set of locations on the board of what piece should be moved and where it should be moved to.

-public static int[] mill(int[][] visibleTeams, int colour, int added) : A public method taking 2 inputs: visibleTeams, an integer array of the setup of the board; colour, an integer of what which team’s piece is being placed. Determines and returns the location of a piece on the board that should be removed.

Private Programs

-private static int[] random(int[][] visibleTeams, int search) : A private method that searches for a random piece of a certain colour on the board and returns its location.

-private static in[][] nearMill(int[][] visibleTeams, int colour, int action) : A private method that searches for any near mills of a certain colour and either returns and array of integer arrays of either the empty spots of near mills to be completed or pieces in near mills to be removed.

-private static int nearMillRandom(int empty, int pieceA, int pieceB, int action) : A private method that, based on the value of action, either returns the location of the empty spot of a near mill or one of the two pieces in a near mill.

-private static int[][] checkAdjacent(int[][] visibleTeams, int adj, int i, int j) : A private method that, given a location and a type to search for (a piece of a certain colour or an empty spot), will return an array of integer arrays of the locations of the adjacent pieces of that type.

-private static int[] randomAdjacent(int[][] visibleTeams, int search) : A private method that returns the first open spot it finds adjacent to the first piece of a certain colour it finds.

-private static int[][] findMill(int[][] visibleTeams, int notColour) : A private method that searches for any mills that a player has and returns one piece from each mill to be removed.

-private static int findMillRandom(int pieceA, int pieceB, int pieceC) : A private method that, given a mill, randomly returns one of the three pieces to be removed.

-private static int[] nearbyPiece(int[][] visibleTeams, int[][] nearMills, int search) : A private method that, given an array of locations of near mills, will search for and return the location of a nearby piece of a certain colour to block or complete the near mill, if one exists.

**GameData.java**

Variables

-private int[][] visibleTeams : An integer array of the current state of each piece

-private boolean redTake : A boolean to mark whether or not red is the active player. If true, red is active. If not, red is not active.

-private boolean blueTake : A boolean to mark whether or not blue is the active player. If true, blue is active. If not, blue is not active.

-private boolean first : A boolean that represents which player goes first.

-private int currentState : An integer to mark which state the game is currently in.

-private int previousState : An integer to mark which state the game was in before it became its current state.

-private int play1count : The number of active pieces for player 1.

-private int play2count : The number of active pieces for player 2.

-private int play1removed : The number of pieces of player 1’s that have been removed.

-private int play2removed : The number of pieces of player 2’s that have been removed.

-private int previousMoves : The number of moves that have been made in the game.

-private final int allowable : The maximum number of pieces each player is allowed to have.

Access Programs

-public GameData() : A constructor for GameData class. Allows another class to create Game Data for a new game.

-public int[][] getVisibleTeams() : A getter method for the locations of the pieces on the game board.

-public void setVisibleTeams(int x, int y, int value) : A setter method for an individual value of visibleTeams. Allows a piece to be added to the gameboard.

-public void setVisibleTeams(int[][] newTeams) : A setter method for the whole of visibleTeams. Allows for an old game to be loaded to the game board.

-public void incrementPlay1count() : Increments the number of pieces played by player 1 by 1.

-public void decreasePlay1count() : Decrements the number of pieces played by player 1 by 1.

-public void setPlay1count(int count) : A setter method for the number of pieces played by player 1. Allows for an old game’s count of player 1’s pieces to be loaded.

-public int getPlay1count() : A getter method for the number of pieces played by player 1.

-public int getPlay1removed() : A getter method for the number of player 1’s pieces that were removed.

-public void incrementPlay2count() : Increments the number of pieces played by player 2 by 1.

-public void decreasePlay2count() : Decrements the number of pieces played by player 2 by 1.

-public void setPlay2count(int count) : A setter method for the number of pieces played by player 2. Allows for an old game’s count of player 2’s pieces to be loaded.

-public int getPlay2count() : A getter method for the number of pieces played by player 2.

-public int getPlay2removed() : A getter method for the number of player 2’s pieces that were removed.

-public void setRedTake(bolean value) : A setter method for whether or not red can take one of blue’s pieces.

-public Boolean getRedTake() : A getter method for whether or not red can take one of blue’s pieces.

-public void setBlueTake(bolean value) : A setter method for whether or not blue can take one of red’s pieces.

-public Boolean getBlueTake() : A getter method for whether or not blue can take one of red’s pieces.

-public void incrementTake() : A method that switches which player is active.

-public void setFirst(boolean bool) : A setter method that sets which player moves first.

-public boolean getFirst() : A getter method that returns which player moves first.

-public void setState(int state) : A setter method that sets the previous state to the current state and sets the current state to the new state.

-public int getCurrentState() : A getter method that returns the current state of the game.

-public int getPreviousState() : A getter method that returns the previous state of the game.

-public int getPreviousMoves() : A getter method that returns the number of moves previously made.

-public void resetPreviousMoves() : A method that sets the number of previous moves to 0.

-public void incrementPreviousMoves() : A method that increases the number of previous moves by 1.

-public void loadGame() : A method that loads a game state from a txt file.

**robut.java: The Computer Player**

The computer player was designed with the three states of gameplay in mind. Just as the player was implemented through parsing inputs in differing methods depending on the state of the game, the computer was implemented as an abstract class that returns inputs of where to move next depending on which state the game is in. For each state there is a corresponding function to determine what move the computer player should make: place() corresponds with state 1; move() corresponds with state 2; mill() corresponds with state 3.

Each of these functions works like a state machine—if a certain set of conditions are met a particular action is performed. For instance, place() begins by checking how many pieces are added to the board. If no pieces have been added, the computer places a piece on a random location on the board. Otherwise, the computer checks if it has a near mill (i.e. has two of three spots needed for a mill filled where the final spot is empty). If there is a near mill for the computer, it places a piece so that it has a mill. If not, it checks if the other player has a near mill to block. If there is one the computer will place a piece to block it. However, if there is no near mill to block, the computer will try to create a near mill by placing a piece next to one of its own. If this cannot be done, the computer will add a piece to a random location.

The function move() works in a similar fashion. It begins by checking if the computer has a near mill. If it has that it continues to check for whether there is one of the computer’s pieces adjacent to the empty spot of the near mill. If that is true, then the piece is moved to complete the mill. However, if any one of those are false move() checks if the other player has a near mill. If true, it continues to check for whether one of the computer’s pieces is adjacent to the empty spot of the opponent’s near mill. If that is also true, the piece is moved to block the other player from creating a mill. If all else fails, a random piece of the computer’s is moved to a random open adjacent location.

Finally, mill()’s implementation follows that of move() and place(). It begins by checking if the opposing player has a mill and, if it does, randomly removes one of the three pieces that make up the mill. If the opponent does not have a mill, the function checks if they have a near mill instead and removes one of the pieces that makes it up, if it exists. Otherwise, mill() randomly removes one of the opponent’s pieces.

**TEST**

|  |  |  |
| --- | --- | --- |
| What was Tested | What it Did | Comments |
| robut: place(), with input visibleTeams = {{0,0,2,0,0,0,1,1}, {0,0,0,0,0,0,0,0}} , 1 , 3 | Output location was 0,0 | Place() is functioning correctly |
| robut: place(), with input visibleTeams = {{0,0,2,0,0,0,1,1}, {0,0,0,0,0,0,0,0}} , 2 , 3 | Output location was 0,0 | Place() is functioning correctly |
| robut: place(), with input visibleTeams = {{0,0,2,0,0,0,0,0}, {0,0,0,0,0,2,1,2}} , 1 , 2 | The function got stuck in an infinite loop | Changed randomAdjacent() so it does not randomly select a piece but instead it uses a for loop to check for a piece with and adjacent empty space |
| robut: place(), with input visibleTeams = {{0,0,2,0,0,0,0,0}, {0,0,0,0,0,2,1,2}} , 1 , 2 | Array out of bound exception | Changed randomAdjacent() so it either returns an array of the location or it returns null |
| robut: place(), with input visibleTeams = {{0,0,2,0,0,0,0,0}, {0,0,0,0,0,2,1,2}} , 1 , 2 | Output location was {0,4} | Place() is functioning correctly |
| robut: move(), with input visibleTeams = {{0,1,2,1,0,2,2,2}, {0,1,1,2,0,0,0,0}} , 2 | Output piece and location were {0,7} and {0,0} | Move is not functioning correctly, added catch so pieces in near mills cannot move to try complete themselves |
| robut: move(), with input visibleTeams = {{0,1,2,1,0,2,2,2}, {0,1,1,2,0,0,0,0}} , 2 | Output piece and location were {1,3} and {1,4} | Move is functioning correctly |
| robut: move(), with input visibleTeams = {{0,1,2,1,0,2,2,2}, {0,1,1,2,0,0,0,0}} , 1 | Output piece and location were {0,1} and {0,0} | Move is functioning correctly |
| robut: mill(), with input visibleTeams = {{0,1,2,1,0,2,2,2}, {0,1,1,2,0,0,0,0}} , 1 | Output was {0,6} | Mill is functioning correctly |
| robut: mill(), with input visibleTeams = {{0,1,2,1,0,2,2,2}, {2,1,1,2,0,0,0,0}} , 1 | Output was {0,0} | Mill is functioning correctly |
|  | Below is general testing of game |  |
| Board was {{2,1,2,1,2,0,2,1}, {0,1,2,1,2,0,1,0}}, piece at 0,7 was clicked and was attempted to be moved to 1,7 | Piece could not be moved to 1,7. Random clicking led to the piece finally being moved to 1,0 | Game is not functioning correctly |
| Playing game with AI, Board was {{1,0,2,2,2,0,1,2} {1,1,2,0,1,1,2,1}}, player (2) has mill and tried to remove a 1 piece | Piece could not be removed, blue got an extra turn | Game is not functioning correctly |
| Playing game with AI, Board was {{1,0,2,2,0,2,1,2} {1,1,2,0,1,1,2,1}}, player (2) moved piece at 0,5 to 0,4 to create mill | Game did not recognize mill, instead game was ended in draw | Game is not functioning correctly |
| Playing game with AI, Board was {{0,0,1,2,1,1,1,0} {2,1,1,2,2,0,0,2}}, computer (1) has mill | AI did not remove piece, instead got an extra turn to move | Game is not functioning correctly |