THỰC HÀNH CÔNG CỤ VÀ MÔI TRƯỜNG VÀ LẬP TRÌNH 2

TS. Võ Phương Bình – Email: binhvp@dlu.edu.vn
Information Technology Faculty - Dalat University
Website: http://it.dlu.edu.vn/ivp-lab

LAB 7.2 (4 tiết): Lập trình GUI và xử lí sự kiện

A. Mục tiêu:

- Lập trình giao diện GUI, sử dụng các sự kiện chuột, bàn phím kết hợp các hàm vẽ tạo ra các hình ảnh trên môi trường Frame.
- Xử lý các sự kiện trên môi trường GUI.

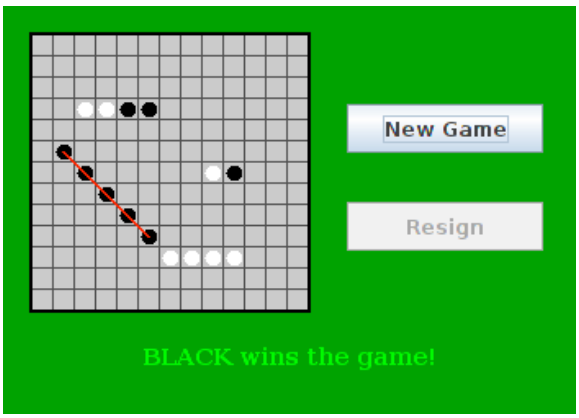
B. Kết quả sau khi hoàn thành:

- Sử dụng được các thành phần thiết kế để có thể tạo ra các giao diện, vẽ các hình ảnh và tô màu cho các hình ảnh đó.
- Xây dựng các ứng dụng có sự kiện chuột và bàn phím.

C. Yêu cầu:

The game of Go Moku (also known as Pente or Five Stones) is similar to Tic-Tac-Toe, except that it is played on a much larger board and the object is to get five squares in a row rather than three. Players take turns placing pieces on a board. A piece can be placed in any empty square. The first player to get five pieces in a row -- horizontally, vertically, or diagonally -- wins. If all squares are filled before either player wins, then the game is a draw. Write a program that lets two players play Go Moku against each other.

Here is a picture of the program, just after black has won the game.



The Solution

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This panel lets two users play Go Moku (a.k.a Pente) against each
 * other.  Black always starts the game.  When a player gets five-in-a-
row,
 * that player wins.  The game ends in a draw if the board is filled
 * before either player wins.
 *
 * The class has a main() routine that lets it be run as a stand-alone
 * application.  The application just opens a window that uses an object
 * of type GoMoku as its content pane.
 *
 */

public class GoMoku extends JPanel {

   /**
    * Main routine makes it possible to run GoMoku as a stand-alone
    * application.  Opens a window showing a GoMoku panel; the program
    * ends when the user closes the window.
    */
   public static void main(String[] args) {
      JFrame window = new JFrame("GoMoku");
      GoMoku content = new GoMoku();
      window.setContentPane(content);
      window.pack();
      Dimension screensize = Toolkit.getDefaultToolkit().getScreenSize();
      window.setLocation( (screensize.width - window.getWidth())/2,
            (screensize.height - window.getHeight())/2 );
      window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      window.setResizable(false);
      window.setVisible(true);
   }


   private JButton newGameButton;  // Button for starting a new game.

   private JButton resignButton;   // Button that a player can use to end
the
                                   //   game by resigning.

   private JLabel message;  // Label for displaying messages to the user.


   /**
    *  The constructor lays out the panel.  The work of
    *  the game is all done in the Board object.  A null layout
    *  is used, and all setup of sizes and positions is done here.
    */
   public GoMoku() {
```

```
      setLayout(null);  // I will do the layout myself.

      setPreferredSize( new Dimension(350,250) );

      setBackground(new Color(0,150,0));  // Dark green background.

      /* Create the components and add them to the panel. */

      Board board = new Board();  // Note: The constructor for the
                                  //   board also creates the buttons
                                  //   and label.
      add(board);
      add(newGameButton);
      add(resignButton);
      add(message);

      /* Set the position and size of each component by calling
         its setBounds() method. */

      board.setBounds(16,16,172,172); // Note:  size MUST be 172-by-172 !
      newGameButton.setBounds(210, 60, 120, 30);
      resignButton.setBounds(210, 120, 120, 30);
      message.setBounds(0, 200, 350, 30);
   }


   // ----------------------- Nested class ------------------------------
-----

   /**
    * This panel displays a 168-by-168 pixel checkerboard pattern with
    * a 2-pixel black border.  It is assumed that the size of the
    * canvas is set to exactly 172-by-172 pixels.  This class does
    * the work of letting the users play Go Moku, and it displays
    * the checkerboard.  In this program, the board has 13 rows and
    * columns of squares.
    */
   class Board extends JPanel implements ActionListener, MouseListener {

      int[][] board;   // The data for the board is kept here.  The
values
                       //   in this array are chosen from the following
constants.

      static final int EMPTY = 0,       // Represents an empty square.
                       WHITE = 1,       // A white piece.
                       BLACK = 2;       // A black piece.

      boolean gameInProgress; // Is a game currently in progress?

      int currentPlayer;      // Whose turn is it now?  The possible
values
                              //    are WHITE and BLACK.  (This is valid
only while
                              //    a game is in progress.)

      int win_r1, win_c1, win_r2, win_c2;  // When a player wins by
getting five or more
```

```
                                                // pieces in a row, the
squares at the
                                                // ends of the row are
(win_r1,win_c1)
                                                // and (win_r2,win_c2).  A red
line is
                                                // drawn between these
squares.  When there
                                                // are not five pieces in a
row, the value of
                                                // win_r1 is -1.  The values
are set in the
                                                // count() method.  The value
of win_r1 is
                                                // tested in the
paintComponent() method.

      /**
       * Constructor.  Create the buttons and label.  Listen for mouse
       * clicks and for clicks on the buttons.  Create the board and
       * start the first game.
       */
      public Board() {
         setBackground(Color.LIGHT_GRAY);
         addMouseListener(this);
         resignButton = new JButton("Resign");
         resignButton.addActionListener(this);
         newGameButton = new JButton("New Game");
         newGameButton.addActionListener(this);
         message = new JLabel("",JLabel.CENTER);
         message.setFont(new  Font("Serif", Font.BOLD, 14));
         message.setForeground(Color.GREEN);
         board = new int[13][13];
         doNewGame();
      }


      /**
       * Respond to user's click on one of the two buttons.
       */
      public void actionPerformed(ActionEvent evt) {
         Object src = evt.getSource();
         if (src == newGameButton)
            doNewGame();
         else if (src == resignButton)
            doResign();
      }


      /**
       * Begin a new game; this is called by the actionPerformed()
       * method when a user clicks the New Game button.
       */
      void doNewGame() {
         if (gameInProgress == true) {
               // This should not be possible because New Game button
               // is enabled only when it is legal to use it, but it
doesn't
               // hurt to check.
```

```
            message.setText("Finish the current game first!");
            return;
         }
         for (int row = 0; row < 13; row++)       // Fill the board with
EMPTYs
            for (int col = 0; col < 13; col++)
               board[row][col] = EMPTY;
         currentPlayer = BLACK;   // BLACK moves first.
         message.setText("BLACK:  Make your move.");
         gameInProgress = true;
         newGameButton.setEnabled(false);
         resignButton.setEnabled(true);
         win_r1 = -1;  // This value indicates that no red line is to be
drawn.
         repaint();
      }


      /**
       * Current player resigns; this is called by the actionPerformed()
       * method when a user clicks the Resign button.  Game ends, and
       * opponent wins.
       */
      void doResign() {
         if (gameInProgress == false) {
                // This should not be possible.
            message.setText("There is no game in progress!");
            return;
         }
         if (currentPlayer == WHITE)
            message.setText("WHITE resigns.  BLACK wins.");
         else
            message.setText("BLACK resigns.  WHITE wins.");
         newGameButton.setEnabled(true);
         resignButton.setEnabled(false);
         gameInProgress = false;
      }


      /**
       * This method is called whenever the game ends.  The parameter,
str,
       * is displayed as a message, and the buttons are enabled/disabled
       * to reflect the fact that a game is not currently in progress.
       */
      void gameOver(String str) {
         message.setText(str);
         newGameButton.setEnabled(true);
         resignButton.setEnabled(false);
         gameInProgress = false;
      }


      /**
       * This is called by mousePressed() when a player clicks on the
       * square in the specified row and col.  It has already been
checked
       * that a game is, in fact, in progress.
       */
```

```
    void doClickSquare(int row, int col) {

       /* Check that the user clicked an empty square.  If not, show an
        error message and exit. */

       if ( board[row][col] != EMPTY ) {
          if (currentPlayer == BLACK)
             message.setText("BLACK:  Please click an empty square.");
          else
             message.setText("WHITE:  Please click an empty square.");
          return;
       }

       /* Make the move.  Check if the board is full or if the move
        is a winning move.  If so, the game ends.  If not, then it's
        the other user's turn. */

       board[row][col] = currentPlayer;  // Make the move.
       repaint();

       if (winner(row,col)) {  // First, check for a winner.
          if (currentPlayer == WHITE)
             gameOver("WHITE wins the game!");
          else
             gameOver("BLACK wins the game!");
          return;
       }

       boolean emptySpace = false;     // Check if the board is full.
       for (int i = 0; i < 13; i++)
          for (int j = 0; j < 13; j++)
             if (board[i][j] == EMPTY)
                emptySpace = true;
       if (emptySpace == false) {
          gameOver("The game ends in a draw.");
          return;
       }

       /* Continue the game.  It's the other player's turn. */

       if (currentPlayer == BLACK) {
          currentPlayer = WHITE;
          message.setText("WHITE:  Make your move.");
       }
       else {
          currentPlayer = BLACK;
          message.setText("BLACK:  Make your move.");
       }

    }  // end doClickSquare()


    /**
     * This is called just after a piece has been played on the
     * square in the specified row and column.  It determines
     * whether that was a winning move by counting the number
     * of squares in a line in each of the four possible
     * directions from (row,col).  If there are 5 squares (or more)
     * in a row in any direction, then the game is won.
```

```
       */
      private boolean winner(int row, int col) {

         if (count( board[row][col], row, col, 1, 0 ) >= 5)
            return true;
         if (count( board[row][col], row, col, 0, 1 ) >= 5)
            return true;
         if (count( board[row][col], row, col, 1, -1 ) >= 5)
            return true;
         if (count( board[row][col], row, col, 1, 1 ) >= 5)
            return true;

         /* When we get to this point, we know that the game is not
          won.  The value of win_r1, which was changed in the count()
          method, has to be reset to -1, to avoid drawing a red line
          on the board. */

         win_r1 = -1;
         return false;

      }  // end winner()


      /**
       * Counts the number of the specified player's pieces starting at
       * square (row,col) and extending along the direction specified by
       * (dirX,dirY).  It is assumed that the player has a piece at
       * (row,col).  This method looks at the squares (row + dirX, col +
dirY),
       * (row + 2*dirX, col + 2*dirY), ... until it hits a square that is
       * off the board or is not occupied by one of the player's pieces.
       * It counts the squares that are occupied by the player's pieces.
       * Furthermore, it sets (win_r1,win_c1) to mark last position where
       * it saw one of the player's pieces.  Then, it looks in the
       * opposite direction, at squares (row - dirX, col-dirY),
       * (row - 2*dirX, col - 2*dirY), ... and does the same thing.
       * Except, this time it sets (win_r2,win_c2) to mark the last
piece.
       * Note:  The values of dirX and dirY must be 0, 1, or -1.  At
least
       * one of them must be non-zero.
       */
      private int count(int player, int row, int col, int dirX, int dirY)
{

         int ct = 1;  // Number of pieces in a row belonging to the
player.

         int r, c;    // A row and column to be examined

         r = row + dirX;  // Look at square in specified direction.
         c = col + dirY;
         while ( r >= 0 && r < 13 && c >= 0 && c < 13 && board[r][c] ==
player ) {
            // Square is on the board and contains one of the players's
pieces.
            ct++;
            r += dirX;  // Go on to next square in this direction.
            c += dirY;
```

```
         }

         win_r1 = r - dirX;  // The next-to-last square looked at.
         win_c1 = c - dirY;  //    (The LAST one looked at was off the
board or
         //    did not contain one of the player's pieces.

         r = row - dirX;  // Look in the opposite direction.
         c = col - dirY;
         while ( r >= 0 && r < 13 && c >= 0 && c < 13 && board[r][c] ==
player ) {
            // Square is on the board and contains one of the players's
pieces.
            ct++;
            r -= dirX;   // Go on to next square in this direction.
            c -= dirY;
         }

         win_r2 = r + dirX;
         win_c2 = c + dirY;

         // At this point, (win_r1,win_c1) and (win_r2,win_c2) mark the
endpoints
         // of the line of pieces belonging to the player.

         return ct;

      }  // end count()


      /**
       * Draws the board and the pieces on the board.  If the game has
       * been won by getting five or more pieces in a row, draws a red
line
       * through the pieces.
       */
      public void paintComponent(Graphics g) {

         super.paintComponent(g); // Fill with background color,
lightGray

         /* Draw a two-pixel black border around the edges of the canvas,
          and draw grid lines in darkGray.  */

         g.setColor(Color.DARK_GRAY);
         for (int i = 1; i < 13; i++) {
            g.drawLine(1 + 13*i, 0, 1 + 13*i, getSize().height);
            g.drawLine(0, 1 + 13*i, getSize().width, 1 + 13*i);
         }
         g.setColor(Color.BLACK);
         g.drawRect(0,0,getSize().width-1,getSize().height-1);
         g.drawRect(1,1,getSize().width-3,getSize().height-3);

         /* Draw the pieces that are on the board. */

         for (int row = 0; row < 13; row++)
            for (int col = 0; col < 13; col++)
               if (board[row][col] != EMPTY)
                  drawPiece(g, board[row][col], row, col);
```

```
         /* If the game has been won, then win_r1 >= 0.  Draw a line to
mark
          the five (or more) winning pieces. */

         if (win_r1 >= 0)
            drawWinLine(g);

      }  // end paintComponent()


      /**
       * Draw a piece in the square at (row,col).  The color is specified
       * by the piece parameter, which should be either BLACK or WHITE.
       */
      private void drawPiece(Graphics g, int piece, int row, int col) {
         if (piece == WHITE)
            g.setColor(Color.WHITE);
         else
            g.setColor(Color.BLACK);
         g.fillOval(3 + 13*col, 3 + 13*row, 10, 10);
      }


      /**
       * Draw a 2-pixel wide red line from the middle of the square at
       * (win_r1,win_c1) to the middle of the square at (win_r2,win_c2).
       * This routine is called to mark the pieces that won the game.
       * The values of the variables are set in the count() method.
       */
      private void drawWinLine(Graphics g) {
         g.setColor(Color.RED);
         g.drawLine( 8 + 13*win_c1, 8 + 13*win_r1, 8 + 13*win_c2, 8 +
13*win_r2 );
         if (win_r1 == win_r2)
            g.drawLine( 8 + 13*win_c1, 7 + 13*win_r1, 8 + 13*win_c2, 7 +
13*win_r2 );
         else
            g.drawLine( 7 + 13*win_c1, 8 + 13*win_r1, 7 + 13*win_c2, 8 +
13*win_r2 );
      }


      /**
       * Respond to a user click on the board.  If no game is
       * in progress, show an error message.  Otherwise, find
       * the row and column that the user clicked and call
       * doClickSquare() to handle it.
       */
      public void mousePressed(MouseEvent evt) {
         if (gameInProgress == false)
            message.setText("Click \"New Game\" to start a new game.");
         else {
            int col = (evt.getX() - 2) / 13;
            int row = (evt.getY() - 2) / 13;
            if (col >= 0 && col < 13 && row >= 0 && row < 13)
               doClickSquare(row,col);
         }
      }
```

```
        public void mouseReleased(MouseEvent evt) { }
        public void mouseClicked(MouseEvent evt) { }
        public void mouseEntered(MouseEvent evt) { }
        public void mouseExited(MouseEvent evt) { }


    }  // end nested class Board


} // end class GoMoku
```

D. Kết quả thực hành.

- Sinh viên thực hành ứng dụng trên GUI.
- Thời gian thực hành: 4 tiết.

E. Đánh giá:

- Kiểm tra lại chương trình, thử các kết quả.
- Bắt các lỗi bằng cách sử dụng các phần bắt lỗi: try – catch.

----------Hết----------