JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, No. 2, July-August 2002

Educator's Corner: An OO Application that introduces Heuristic Algorithm Design

Dr. Richard Wiener, University of Colorado, Colorado Springs, U.S.A.

I shall wear the hat of columnist from time-to-time. This series of columns is aimed at fellow educators both within and outside the University. Although the subject area addressed by the column may not be directly involved with object technology, an object-oriented approach to problem solving will be featured.

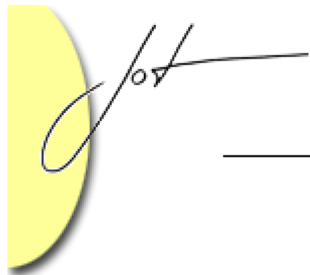
This first column demonstrates how one might wish to introduce the subject of heuristics in teaching algorithm design. To stimulate student interest in this subject, I have chosen a problem familiar to most students – the game of MasterMind™ (registered trademark of Pressman Toy Company). Master Mind was invented in 1970-71 by Mordecai Meirowitz, an Israeli Postmaster. Over 55 million games have been sold worldwide since its release in 1972.

The challenge is to design an algorithm that forms the basis of an application program that allows the computer to guess the human user's secret code with the fewest number of guesses. In this case the application program shall be constructed using Java 1.3.1 running on an Apple Macintosh under OS X. Since this is a new platform for me (most of my work has been done on a Windows 2000 platform) I shall comment, whenever appropriate, on the tools available under this platform.

1 A REVIEW OF MASTER-MIND

The user must first construct a secret code consisting of a sequence of four colors chosen from red, blue, yellow, green, white and orange. In our context, the application program (using the heuristic algorithm) must "guess" the user's code through a series of steps. At each step, the program produces a 4-tuple of colors. The user must then input a score associated with the program's 4-tuple. The scoring is performed as follows: for each color in the 4-tuple the program produces that is identical in color and position to the user's secret code, the user uses a red peg to score that hit. For each color the program produces that is the same as one of the user's colors but is not in the correct position, the user uses a white peg to score that hit. Once the user completes the scoring of the program's 4-tuple, the program produces a new 4-tuple which the user then scores. This process continues until the program produces a 4-tuple that exactly matches the user's

Cite this article as follows: Richard Wiener: Educator's Corner: An OO Application that introduces Heuristic Algorithm Design, in Journal of Object Technology, Vol. 1, No. 2, July-August 2002, pages 63-78. http://www.jot.fm/issues/issue_2002_07/column6



secret code in both colors and positions. In this final case, the user uses 4 red pegs to indicate that the application program has succeeded in "guessing" the secret code.

"Guessing" the user's secret code when playing the actual game by hand is non-trivial, since there are 1296 possible 4-tuples that can form the basis for the secret code. In 1993, Kenji Koyama and Tony W. Lai calculated that the best strategy uses an average of 4.340 moves.

2 THE HEURISTIC ALGORITHM

The heuristic algorithm that shall be used in the application program appears on the website of Radu Rosu (http://www.unc.edu/~radu/mm/MMS.html).

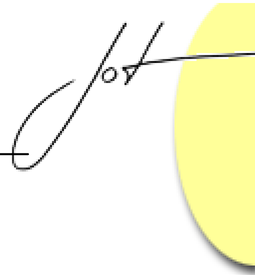
After an arbitrary initial guess, the algorithm produces a random series of 4-tuples until the first is found that is consistent with all the user's previous scores. So instead of utilizing deep analysis and mathematics, the algorithm uses total randomness. The simplicity of the algorithm makes it appealing. In addition, we shall see that this simple algorithm requires an average number of guesses close to 4.6, not too far away from the optimum strategy that requires an average of 4.340 guesses.

The Development Platform, relevant tools and porting Java software

As indicated earlier, the Java application that implements and demonstrates this heuristic approach to playing MasterMind™ was developed on an Apple Macintosh under OS X. JBuilder 6 Enterprise Version was used as the main development tool both for project management and to assist in the production of the GUI. As is known from my review of JBuilder 6 that appeared in the previous issue of JOT (co-authored with Dave Neuendorf), I believe that this Java development tool is of superb quality. It works exactly the same under OS X as under Windows.

To port an earlier version of this application from my PC to the Macintosh, I utilized an important tool, Dave 3.1.1 manufactured by Thursby Software Systems – www.thursby.com. This tool provides complete connectivity between the Apple Macintosh (the new kid on my block) and my existing PC local area network. Without this tool the value of the Macintosh would be significantly reduced.

Any claim that Java software runs "as-is" on all platforms is not true. After porting the Window's version to the Macintosh, it was quickly evident that many of the labels above and on buttons did not fit properly. The default fonts used on each platform are different. Fortunately, this is the only area that required fine tuning.



3 THE APPLICATION

The application is modeled using 5 classes:

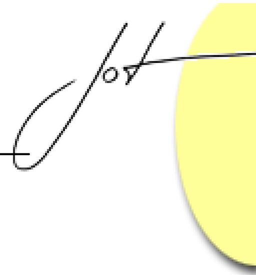
- *MasterMindApplication*: The usual "main driver" found in many GUI applications.
- *MasterMindPanel*: An extension of the standard JPanel component. This class is used to hold the game image and supports the graphical images of pegs for scoring and playing.
- *Row*: This class forms the model of a 4-tuple of color objects. Random row objects can
  be created and their scores computed.
- *MasterMindUI*: The usual user interface class that contains a *MasterMindPanel* object as well as the game control buttons and output messages.
- *Global*: Contains and supports a globally accessible random number object.

Listing 1 contains the details of class *Row*.

Listing 1 – Class Row

```
import java.util.*;
import java.awt.*;

// Models a row in Mastermind
public class Row {

    // Fields

    private Color [] elements = new Color[4];
    private int exactMatches, colorMatches;
    private final Color [] colors =
                  {Color.red, Color.blue, Color.yellow,
          Color.white, Color.green, Color.orange};

    // Constructors
    public Row (Color p1, Color p2, Color p3, Color p4) {
        elements[0] = p1;
        elements[1] = p2;
        elements[2] = p3;
        elements[3] = p4;
        // Helps create a statistically sound random sequence
        for (int i = 0; i < 20000; i++) {
            Global.rnd.nextDouble();
        }
    }

    public Row () {}
```

```
// Commands

   /** Sets values of fields exactMatches and colorMatches */
public void computeScore (Row anotherRow) {
    this.resetScore();

    // Create two local arrays
    Color [] receiver = new Color[4];
    Color [] parameter = new Color[4];
    for (int index = 0; index < 4; index++) {
        receiver[index] = elements[index];
        parameter[index] = anotherRow.elements[index];
    }

    // Find matchups
    for (int index = 0; index < 4; index++) {
        if (receiver[index] == parameter[index]) {
            exactMatches++;
            // Remove match from receiver and parameter
            receiver[index] = null;
            parameter[index] = null;
        }
    }

    // Find the same color with no matchup
    for (int index = 0; index < 4; index++) {
        if (receiver[index] != null) {
            // Is receiver[index] in parameter array
            boolean found = false;
            for (int i = 0; !found && i < 4; i++) {
                if (receiver[index] == parameter[i]) {
                    parameter[i] = null;
                    colorMatches++;
                    found = true;
                }
            }
        }
    }
}

public void generateRandomRow () {
    for (int index = 0; index < 4; index++) {
        int color = Global.rnd.nextInt(6);
        elements[index] = colors[color];
    }
}

public void setMatches (int matches) {
    exactMatches = matches;
}
```



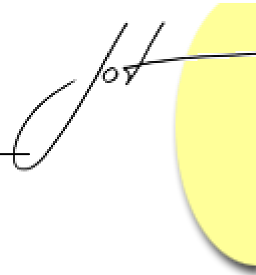
```
    public void setSameColor (int sameColor) {
        colorMatches = sameColor;
    }

    // Queries
    public int matches () {
        return exactMatches;
    }

    public int sameColor () {
        return colorMatches;
    }

    public boolean sameScore (Row other) {
        return exactMatches == other.exactMatches &&
               colorMatches == other.colorMatches;
    }

    public Color p1 () {
        return elements[0];
    }


    public Color p2 () {
        return elements[1];
    }


    public Color p3 () {
        return elements[2];
    }

    public Color p4 () {
        return elements[3];
    }

    public String toString () {
        return p1().toString() + p2().toString() + p3().toString() +
               p4().toString();
    }

    private void resetScore () {
        exactMatches = 0;
        colorMatches = 0;
    }
}
```

Listing 2 presents some of the details of class *MasterMindPanel*.

Listing 2 – Class MasterMindPanel

```
import java.awt.*;
```

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

public class MasterMindPanel extends JPanel {

    // Fields

    // Image of Master Mind Game - Digital photo of game
    private Image masterMindGameImage;

    // Coordinates of peg holes
    private Point [] peg =    {new Point(0, 0),
                               new Point(74, 512),
                               // … details omitted
                               new Point(178, 157),
                               new Point(222, 160)};

    // Coordinates of pin holes for scoring
    private Point [] pin = {new Point(0, 0),
                            new Point(238, 529),
                            new Point(258, 532),
                            // … details omitted
                            new Point(259, 156),
                            new Point(278, 159)};


    // Used by paintComponent for rendering peg or pin
    private Color [] pinSelected = new Color[33];
    private Color [] pegSelected = new Color[33];

    private int pegIndex;
    private int pinIndex;


    // Constructor
    public MasterMindPanel () {
        masterMindGameImage =
            Toolkit.getDefaultToolkit().getImage("MasterMind.gif");
        MediaTracker tracker = new MediaTracker(this);
        tracker.addImage(masterMindGameImage, 0);
        try {
            tracker.waitForID(0);
        } catch (InterruptedException ex) {
            System.out.println(ex);
        }
    }

    public void paintComponent (Graphics g) {
        super.paintComponent(g);
        g.drawImage(masterMindGameImage, 0, 0, this);
        int diameter = 23;
```



```
        for (int j = 1; j <= 32; j++) {
            if (pegSelected[j] != null) {
                g.drawArc(peg[j].x, peg[j].y, diameter, diameter, 0,
                          360);
                Color pegColor = pegSelected[j];
                g.setColor(pegColor);
                g.fillArc(peg[j].x, peg[j].y, diameter, diameter, 0,
                          360);
            }
        }
        diameter = 11;
        for (int j = 1; j <= 32; j++) {
            if (pinSelected[j] != null) {
                g.drawArc(pin[j].x, pin[j].y, diameter, diameter, 0,
                360);
                Color pinColor = pinSelected[j];
                g.setColor(pinColor);
                g.fillArc(pin[j].x, pin[j].y, diameter, diameter, 0,
                          360);
            }
        }
    }

    public void drawPeg (Color color, int row, int col) {
        pegIndex = (row - 1) * 4 + col;
    }

    public void drawPin(Color color, int row, int col) {
        pinIndex = (row - 1) * 2 + col;
    }

    public void setPinColor(Color color, int index) {
        pinSelected[index] = color;
    }

    public void setPegColor(Color color, int row, int col) {
        pegSelected[(row - 1) * 4 + col] = color;
    }


    public boolean pinColor (int index) {
        return pinSelected[index] != null;
    }

    public Color [] pinsSelected () {
        return pinSelected;
    }
}
```

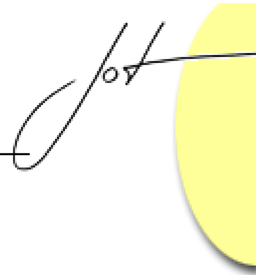
The constructor handles the task of downloading the image from a *.gif* file. This *.gif* file was produced by taking a digital photo of the real game. The points defined in the *peg*

and *pin* arrays were obtained tediously by hand since the digital image was off-center. The *paintComponent* method is automatically activated whenever the GUI requires refreshing such as in response to a resize event or re-validate event. As is typical in Java, graphics-based messages are transmitted through a *Graphics* object .

Listing 3 presents some of the details of class *MasterMindUI*. All the event handlers are present in this class including the logic for the decision about which pin hole the user has selected using either a left or right mouse click.

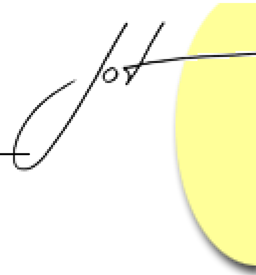
Listing 3 – Class MasterMindUI

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

public class MasterMindUI extends JFrame {

    // Fields
    private Row [] board = new Row[10]; // Holds the game board
    private int rowNumber = 0;
    private Row code = new Row();
    private Row newRow;
    private MasterMindPanel imagePanel;
    private Color [] userCode = new Color[4]; // For scoring
                                              // verification

    // Coordinates of scoring pegs
    private Point [] scoringPeg =
        {new Point(0, 0), new Point(248, 557), new Point(266, 557),
         // … Details omitted
         new Point(284, 204), new Point(265, 182), new Point(284,185)};

    // Assorted graphical component objects, not shown

    // Constructor and initialization method not shown


    private void this_mouseClicked (MouseEvent e) {
        int error = 3;
        for (int i = 1; i <= 32; i++) {
            if ((e.getX() >= scoringPeg[i].x - error &&
                    e.getX() <= scoringPeg[i].x + error) &&
                    e.getY() >= scoringPeg[i].y - error &&
                    e.getY() <= scoringPeg[i].y + error) {
                int row = (i - 1) / 2 + 1;
                int col = i % 2 == 1 ? 1 : 2;
                Color pinColor;
                if (imagePanel.pinColor(i)) {
                    pinColor = null;
```



```
                } else {
                    pinColor =
                        SwingUtilities.isLeftMouseButton(e) ? Color.
                        red : Color.white;
                }
                imagePanel.drawPin(pinColor, row, col);
                imagePanel.repaint();
                imagePanel.setPinColor(pinColor, i);
            }
        }
    }

    private void newGame () {
        // Advance the random number generator
        for (int i = 0; i < 50000; i++) {
            Global.rnd.nextDouble();
        }

        rowNumber = 0;

        // Clear all pegs and pins
        for (int j = 1; j <= 32; j++) {
            imagePanel.setPinColor(null, j);
        }
        for (int r = 1; r <= 8; r++) {
            for (int c = 1; c <= 4; c++) {
                imagePanel.setPegColor(null, r, c);
            }
        }
        imagePanel.repaint();
        imagePanel.setPegColor(Color.red, 1, 1);
        imagePanel.setPegColor(Color.red, 1, 2);
        imagePanel.setPegColor(Color.blue, 1, 3);
        imagePanel.setPegColor(Color.blue, 1, 4);
        imagePanel.repaint();
              // Choose from among 6 random starting configurations
        int choice = Global.rnd.nextInt(6) + 1;
        // Details not shown

        board[0] = newRow;
    }

       private void enterScore () {
        // Get range of pinsSelected to query
        int row = rowNumber + 1;
        int endIndex = row * 4;
        int startIndex = endIndex - 3;
        int matches = 0;
        int same = 0;
        Color [] pinsSelected = imagePanel.pinsSelected();
        for (int i = startIndex; i <= endIndex; i++) {
            if (pinsSelected[i] == Color.red) {
```

```
                matches++;
            } else if (pinsSelected[i] == Color.white) {
                same++;
            }
        }
        // Verify that user has entered correct score
        newRow.computeScore(new Row(userCode[0], userCode[1],
                                    userCode[2], userCode[3]));
        if (matches != newRow.matches() ||
                same != newRow.sameColor()) {
            JOptionPane.showMessageDialog(this,
                  "Incorrect score entered.");
            return;
        }
        if (newRow == null) {
            return;
        }
        newRow.setMatches(matches);
        newRow.setSameColor(same);
        board[rowNumber] = newRow;
        if (matches == 4) {
            JOptionPane.showMessageDialog(this,
                                "It took the computer " + (rowNumber + 1)
+
                   " guesses.");
            codeField1.setText("");
            codeField2.setText("");
            codeField3.setText("");
            codeField4.setText("");
            return;
        }

        /* Generate a random row that satisfies all the previous board
           scores entered by the user.
        */
        int counter = 0;
        boolean allRowsMatchScore = true;
        do {
            counter++;
            newRow = new Row();
            newRow.generateRandomRow();
            allRowsMatchScore = true;
            for (int r = 0; allRowsMatchScore &&
                    r <= rowNumber; r++) {
                /* Value of internal fields exactMatches and
                   colorMatches set for newRow object based on board[r]
                */
                newRow.computeScore(board[r]);
                /* Returns true if existing scores for board[r] are the
                   same as those just computed for newRow based on
                   board[r].
                */
                allRowsMatchScore = board[r].sameScore(newRow);
```



```
            }
        } while (!allRowsMatchScore && counter <= 50000 &&
                rowNumber <= 7);
        if (rowNumber >= 8 || counter >= 50000) {
            JOptionPane.showMessageDialog(this,
                  "You have an illegal entry. Game will be stopped.");
            newGame();
            return;
        }
        rowNumber++;
        Color color1 = newRow.p1();
        Color color2 = newRow.p2();
        Color color3 = newRow.p3();
        Color color4 = newRow.p4();
        row++;
        imagePanel.setPegColor(color1, row, 1);
        imagePanel.setPegColor(color2, row, 2);
        imagePanel.setPegColor(color3, row, 3);
        imagePanel.setPegColor(color4, row, 4);
        imagePanel.repaint();
    }

       // Several event handler methods not shown

}
```

The *this_mouseClicked* event handler method considers the user to have hit a scoring hole if the mouse click is within plus or minus 3 pixels of the center.

The heuristic algorithm that is at the heart of the application is shown in the private *enterScore* method. The pertinent code is shown in boldface.

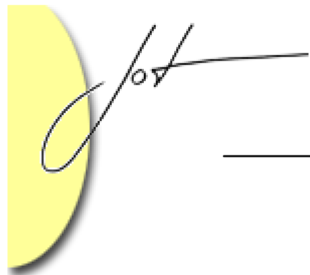
Listing 4 shows the details of class Global that makes a random number object globally accessible.

Listing 4 – Class Global

```
import java.util.*;

public class Global {
    public static Random rnd = new Random();
}
```

4 RUNNING THE APPLICATION ON A MACINTOSH

The OS X environment on the Macintosh makes it easy to create "clickable" applications. A tool called *MRJAppBuilder* may be used to create a native Macintosh application. Its



wizard walks the user through the appropriate steps. Another approach, and one that I prefer, is to create an application *.jar* file that is clickable. To make the *.jar* file clickable, one needs to create a *manifest.txt* file and use it while constructing the *.jar file* for the application.

The manifest.txt file needed to make the application clickable contains the one line,

Main-Class: MasterMindApplication

From an OS X shell opened to the sub-directory containing the application files (the ability for a Macintosh programmer to have access to a standard command shell is a relatively new event in the history of Apple Computers – one that is long overdue and greatly appreciated), the application is compiled using the usual

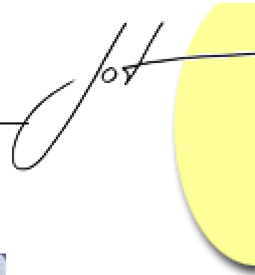
*javac *.java* command

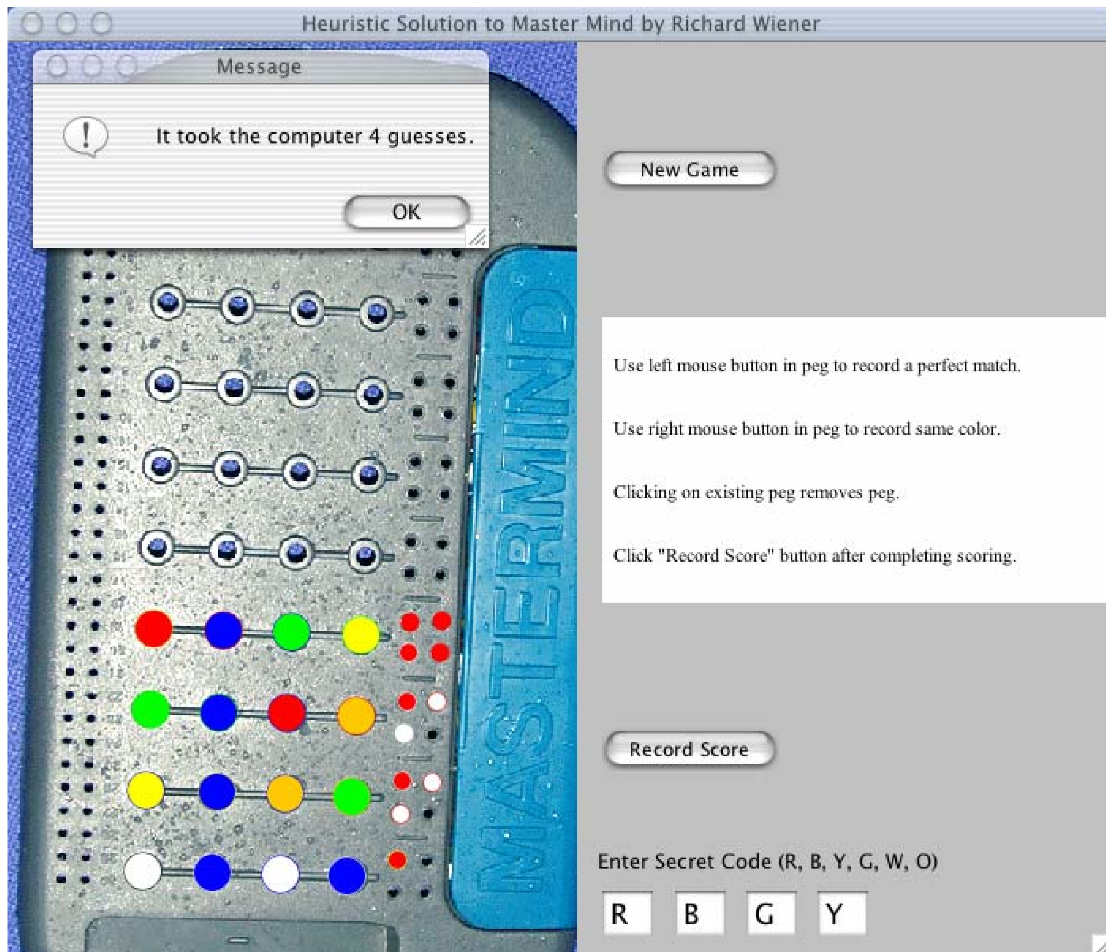
Next the clickable *.jar* file is produced from the command:

*jar –cvfm MasterMind.jar manifest.txt *.class*

The *MasterMind.jar* file can then be renamed simply *MasterMind* and double clicked to launch it.

A screen shot of the application while running is:





5 SOME STATISTICS ON THE HEURISTIC ALGORITHM

It is interesting to examine the efficiency and relative performance of this heuristic algorithm by producing statistics that output:

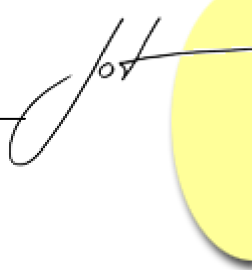
1. The average number of random 4-tuples required as a function of board position before one is accepted.
2. The average number of "guesses" that the algorithm requires.

To accomplish this we generate all possible $6^4 = 1296$ 4-tuples as secret codes and for each determine the number of guesses required for each board position and the total number of rows required (guesses required) before a solution is reached. We output the average results over the 1296 games that are simulated.

Listing 5 presents the details of class *MasterMindStats*.

Listing 5 – Class MasterMindStats

```
import java.util.*;
import java.text.*;

import java.awt.*;

/**
 * Determine the average number of guesses and the average number
 * of 4-tuples that need to be generated as a function of row number.
 */
public class MasterMindStats {

    // Fields
    private Row [] board = new Row[10];    // Holds the game board
    private int rowNumber = 0;
    private Row code;
    public static Random rnd = new Random();
    private int totalNumberGuesses = 0;
    private int [] numberGenerated = new int[10]; // Number of 4-tuples
    private Color [] colors = { Color.red, Color.blue, Color.yellow,
                                Color.white, Color.green, Color.orange
                              };

    public static void main (String[] args) {
        MasterMindStats app = new MasterMindStats();
        DecimalFormat df = new DecimalFormat("0.###");

        TimeInterval t = new TimeInterval();
        t.startTiming();
              // Generate all possible secret codes
        for (int color1 = 0; color1 < 6; color1++) {
            for (int color2 = 0; color2 < 6; color2++) {
                for (int color3 = 0; color3 < 6; color3++) {
                    for (int color4 = 0; color4 < 6; color4++) {
                        app.code = new Row(app.colors[color1],
                                           app.colors[color2],
                                           app.colors[color3],
                                           app.colors[color4]);
                        // Always use the same initial  4-tuple
                        Row newRow = new Row(Color.red, Color.red,
                                             Color.blue, Color.blue);

                        app.rowNumber = 0;
                        app.board[app.rowNumber] = newRow;
                        newRow.computeScore(app.code);

                        /* Generate a random row that satisfies all the
                           previous board scores.
```



```
                                */
                                while (newRow.matches() != 4) {
                                    boolean allRowsMatchScore = true;
                                    int numberGenerated = 0;

                                    do {
                                        newRow = new Row();
                                        newRow.generateRandomRow();
                                        allRowsMatchScore = true;
                                        for (int r = 0; allRowsMatchScore &&
                                                  r <= app.rowNumber; r++) {
                                            newRow.computeScore(app.board[r]);
                                            allRowsMatchScore =
                                                app.board[r].sameScore(newRow);
                                        }
                                        numberGenerated++;
                                    } while (!allRowsMatchScore);
                                    app.rowNumber++;
                                    app.board[app.rowNumber] = newRow;
                                    newRow.computeScore(app.code);
                                    app.numberGenerated[app.rowNumber] +=
                                        numberGenerated;
                                }
                                app.totalNumberGuesses += app.rowNumber + 1;
                            }
                        }
                    }
                }
                t.endTiming();
                System.out.println();
                System.out.println("Elapsed time: " + t.elapsedTime() +
                                   " seconds.");
                System.out.println();
                System.out.println("Average number of guesses: "
                                   + df.format(app.totalNumberGuesses /
                                     1296.0));
                for (int i = 1; i <= 8; i++) {
                    System.out.println("Average Number 4-tuples Generated[" +
                                       i + "] = " +
                                       df.format(app.numberGenerated[i] /
                                       1296.0));
                }
                System.out.println();
            }
        }
```

Listing 6 shows the support class *TimeInterval*.

Listing 6 – Class TimeInterval

```
/**
   * A timing utility class useful for timing code segments
*/
```

```
public class TimeInterval {

    private long startTime, endTime;
    private long elapsedTime; // Time interval in milliseconds

    // Commands
    public void startTiming() {
        elapsedTime = 0;
        startTime = System.currentTimeMillis();
    }

    public void endTiming() {
        endTime = System.currentTimeMillis();
        elapsedTime = endTime - startTime;
    }

    // Queries
    public double elapsedTime() {
        return (double) elapsedTime / 1000.0;
    }
}
```

The output for a typical run is:

Elapsed time: 12.858 seconds.

Average number of guesses: 4.639

Average Number 4-tuples Generated[1] = 11.434

Average Number 4-tuples Generated[2] = 87.522

Average Number 4-tuples Generated[3] = 430.57

Average Number 4-tuples Generated[4] = 530.5

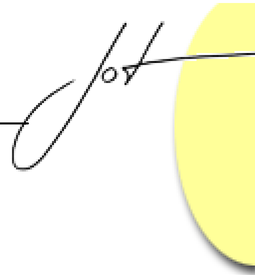
Average Number 4-tuples Generated[5] = 158.288

Average Number 4-tuples Generated[6] = 12.807

Average Number 4-tuples Generated[7] = 0

Average Number 4-tuples Generated[8] = 0

Complete Java sources are available for download at
http://www.jot.fm/issues/issue_2002_07/column6/Mastermind.zip



About the author

Richard Wiener is Associate Professor of Computer Science at the University of Colorado at Colorado Springs. He is also the Editor-in-Chief of JOT and former Editor-in-Chief of the Journal of Object Oriented Programming. In addition to University work, Dr. Wiener has authored or co-authored 21 books and works actively as a consultant and software contractor whenever the possibility arises.