Mehran Sahami
CS 106A

Handout #39
November 13, 2017

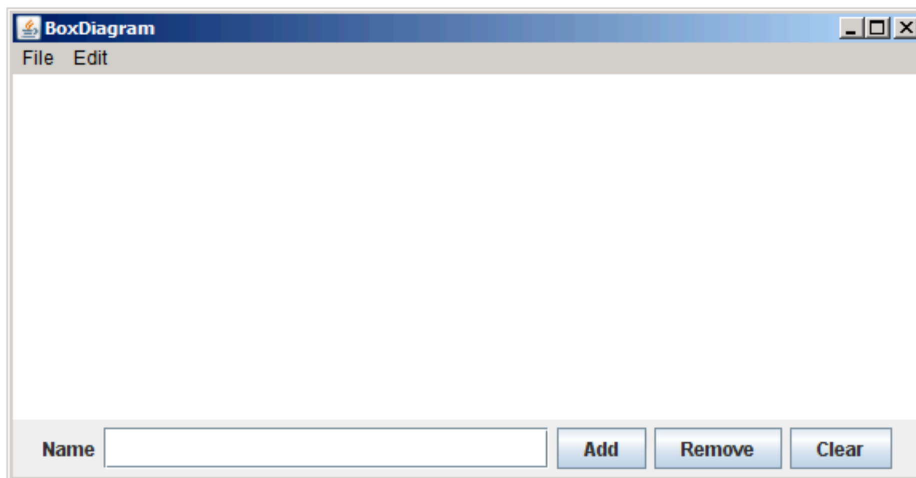
# Section Handout #7: Using Interactors and the Debugger

Based on a handout by Eric Roberts

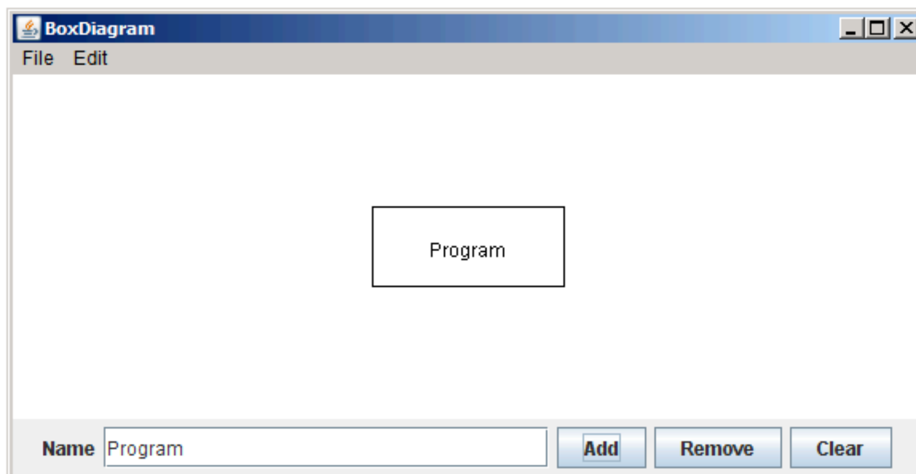
## 1. Using Interactors

The purpose of this problem is to give you some practice using the kind of interactors you need for the NameSurfer project in Assignment #6.

The specific example is to build an interactive design tool that allows the user to arrange labeled boxes on the window. The starting configuration for the program presents an empty graphics canvas and a control bar containing a **`JLabel`**, a **`JTextField`**, and three **`JButton`**s. The window as a whole looks like this:




The most important operation in the program is to be able to add a new box to the screen, which you do by typing the name of the box in the **`JTextField`** and clicking the **Add** button, or pressing the ENTER key. Doing so creates a new labeled box with that name in the center of the window. For example, if you entered the string **Program** in the **`JTextField`** and clicked **Add**, you would see the following result:

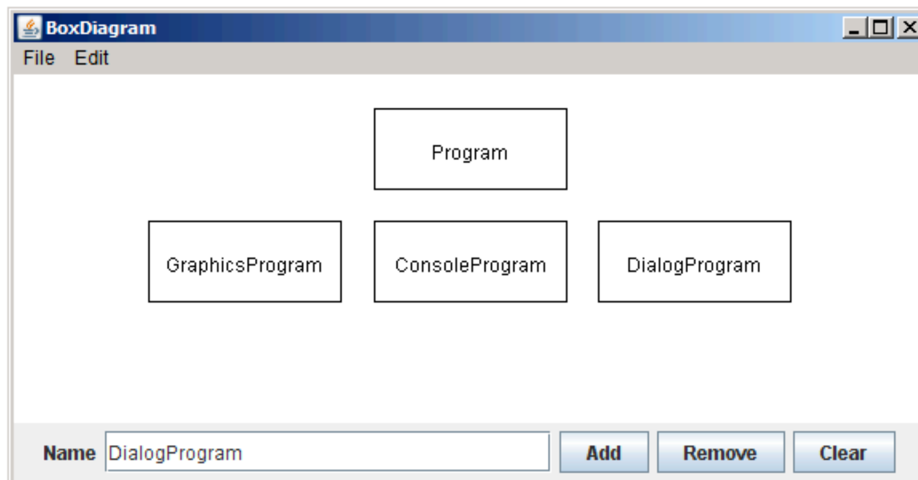



For this simple version of the application, you may assume that the box always has constant dimensions, as specified by the following constant definitions:

```
private static final double BOX_WIDTH = 120;
private static final double BOX_HEIGHT = 50;
```

Once you have created a labeled box, your program should allow you to move the box as a unit by dragging it with the mouse. Because the outline and the label must move together, it makes sense to combine the **GRect** and **GLabel** into a **GCompound** and then use code similar to that in Figure 10-4 in the book to implement the dragging operation.

The ability to create new boxes and drag them to new positions makes it possible to draw box diagrams containing an arbitrary number of labeled boxes. For example, you could add three more boxes with names **GraphicsProgram**, **ConsoleProgram**, and **DialogProgram** respectively, and drag them around to create this diagram of the **Program** class hierarchy:




The other two buttons in the control strip are **Remove** and **Clear**. The **Remove** button should delete the box whose name appears in the **JTextField**; the **Clear** button should remove all of the boxes. Note that you do not need to worry about the user trying to create multiple boxes with the same name–you can assume that all box names are unique.

While the operations in this program are conceptually simple, they influence the design in the following ways:

- The fact that you may need to remove a box by name forces you to keep track of the objects that appear in the window in some way that allows you to look up a labeled box given the name that appears inside it. You therefore need some structure—and there is an obvious choice in the Java Collection Framework—that keeps track of all the boxes on the screen.
- If the only objects in the window were labeled boxes, you could implement the **Clear** button by removing everything from the **GCanvas**. While that would work for this assignment, you might want to extend the program so if there were other objects on the screen that were part of the application itself, they would remain on the screen. In that case, you would want to implement **Clear** by iterating through the set of boxes on the screen and removing each one. Remember, you can assume all box names are unique.

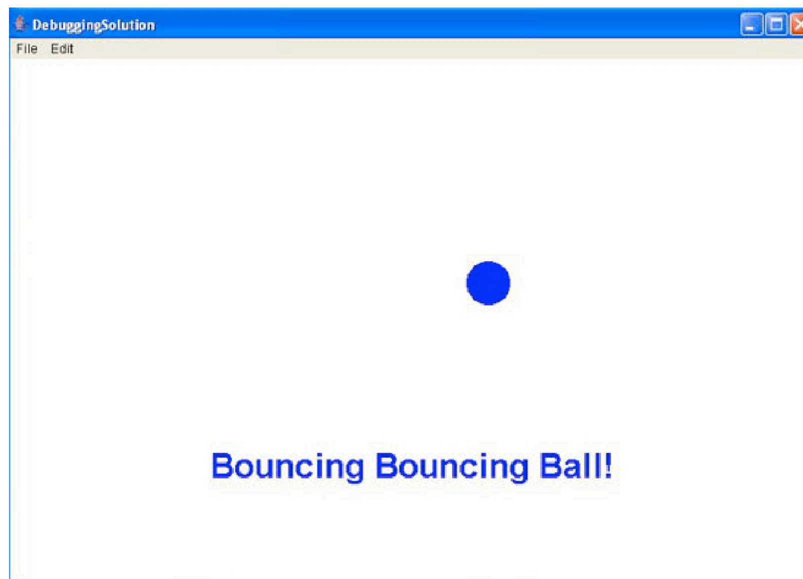
## 2. Using the Debugger

Debugging is an essential programming skill. Sure, you can often find bugs by re-reading your code looking for silly mistakes. But for large, complex programs (and sometimes even small ones) it helps tremendously to see what the computer is actually doing behind the scenes.

Eclipse, like most modern programming environments, comes equipped with a debugger. A debugger is a handy interface for monitoring programs as they run. With the Eclipse debugger, for example, you can follow along with your program line by line.

In this problem, your section leader will demonstrate how to use the debugger. It might look intimidating at first, but we hope to convince you that the debugger really is easy to use.

### Bouncing Bouncing Ball




Provided below is a program that is *supposed* to animate a ball that bounces off the four walls. The ball should change color whenever it hits a wall. The program is supposed to display a horizontally centered label "Bouncing Bouncing Ball!" Finally, the label color should always match the color of the ball, and both should initially be colored red.

Use the Eclipse debugger to figure out why the program fails, and fix the errors.

```
/*
 * File: DebuggingExample.java
 * ----------------------------
 * This program is supposed to animate a ball that bounces off the
 * four walls.  The ball should change color whenever it hits a wall.
 * The program should also display a big "Bouncing Bouncing Ball!"
 * label, centered horizontally.  The label color should always match
 * the color of the ball, and both should initially be colored red.
 *
 * However, the program has a few errors.  Use the Eclipse debugger
 * to find and fix them!
 */

import acm.program.*;
import acm.graphics.*;
import acm.util.*;

import java.awt.*;

public class DebuggingExample extends GraphicsProgram {

   private RandomGenerator rgen = RandomGenerator.getInstance();

   private static final int BALL_RADIUS = 20;

   private static final double MIN_DX = 2.0;
   private static final double MAX_DX = 3.0;
   private static final double MIN_DY = 2.0;
   private static final double MAX_DY = 3.0;

   private static final double PAUSE = 15.0;
   private static final int TEXT_HEIGHT = 100;

   private static final int NUM_COLORS = 5;

   private GLabel text;

   public void run() {
      GOval ball = null;

      double dx = rgen.nextDouble(MIN_DX, MAX_DX);
      double dy = rgen.nextDouble(MIN_DY, MAX_DY);

      setupLabel();
      setupBall(ball);

      while (true) {
         ball.move(dx, dy);
         checkForCollisions(ball, dx, dy);
         pause(PAUSE);
      }
   }

   private void setupLabel() {
      text = new GLabel("Bouncing  Bouncing  Ball!");
      double x = (getWidth() / 2.0) - (text.getWidth() / 2.0);
      double y = getHeight() - TEXT_HEIGHT;
      text.setLocation(x,y);
      text.setFont(new Font("Arial", Font.BOLD, 32));
      text.setColor(Color.RED);
      add(text);
   }
```

*Continued on next page…*

```
   private void setupBall(GOval ball) {
      ball = new GOval(BALL_RADIUS * 2, BALL_RADIUS * 2);
      ball.setFilled(true);
      ball.setColor(Color.RED);
      double x = rgen.nextDouble(0, getWidth() - (BALL_RADIUS * 2));
      double y = rgen.nextDouble(0, getHeight() - (BALL_RADIUS * 2));
      ball.setLocation(x, y);
      add(ball);
   }

   /*
    * If the ball collides with one of the walls, change the
    * corresponding direction and change the color.  Note that it's
    * possible to bounce off two walls (e.g. the corner) simultaneously.
    */
   private void checkForCollisions(GOval ball, double dx, double dy) {
      if (ball.getX() <= 0 || ball.getX() >= getWidth() - (BALL_RADIUS * 2)) {
         dx = -dx;
         ball.move(dx, 0); // Move the ball out of the wall
         Color nextColor = getRandomNewColor(ball.getColor());
         ball.setColor(nextColor);
         text.setColor(nextColor);
      }

      if (ball.getY() <= 0 || ball.getY() >= getHeight() - (BALL_RADIUS * 2)) {
         dy = -dy;
         ball.move(0, dy); // Move the ball out of the wall
         Color nextColor = getRandomNewColor(ball.getColor());
         ball.setColor(nextColor);
         text.setColor(nextColor);
      }
   }

   /*
    * Choose a random color that is different than "prevColor"
    */
   private Color getRandomNewColor(Color prevColor) {
      while (true) {
         Color newColor;
         int color = rgen.nextInt(0,  NUM_COLORS - 1);

         switch(color) {
            case 0:
               newColor = Color.RED;
            case 1:
               newColor = Color.ORANGE;
            case 2:
               newColor = Color.YELLOW;
            case 3:
               newColor = Color.GREEN;
            case 4:
               newColor = Color.BLUE;
            default:
               newColor = Color.BLACK;
         }

         // If the new color is different than the previous color,
         // we're done.  Otherwise, choose again.
         if (!newColor.equals(prevColor)) return newColor;
      }
   }
}
```