Five-Minute Review

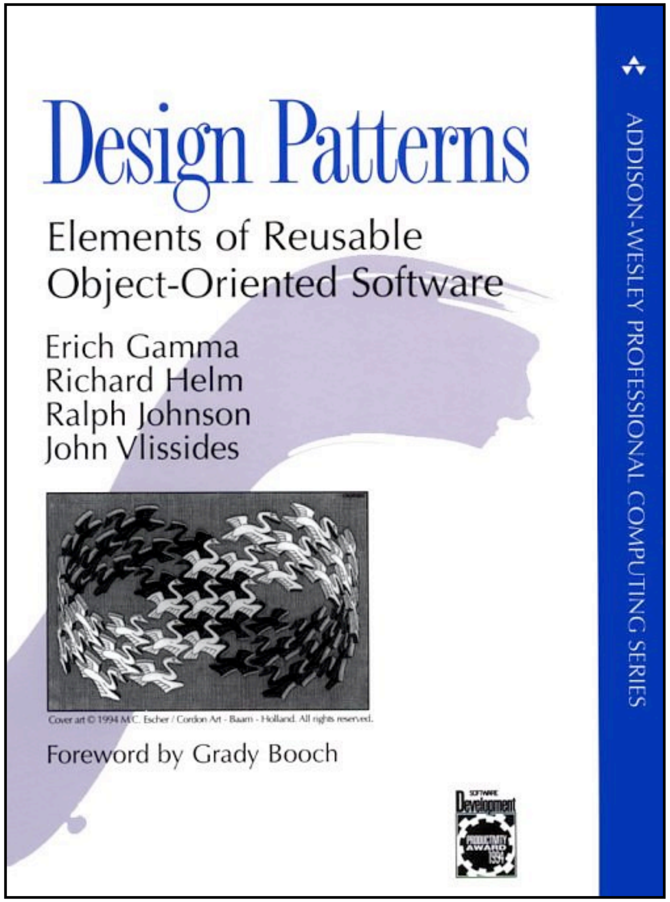
1. What are *interactive programs*?
2. What types of *events* do we distinguish?
3. What is an example of an *event listener*?
4. Which are the *GUI-control strips*?
5. What is the MVC pattern?

Programming – Lecture 14

Looking Ahead (Chapter 14)

- Programming patterns, MVC
- Concurrency
- Race Conditions

Model – View – Controller (MVC)



Design Patterns

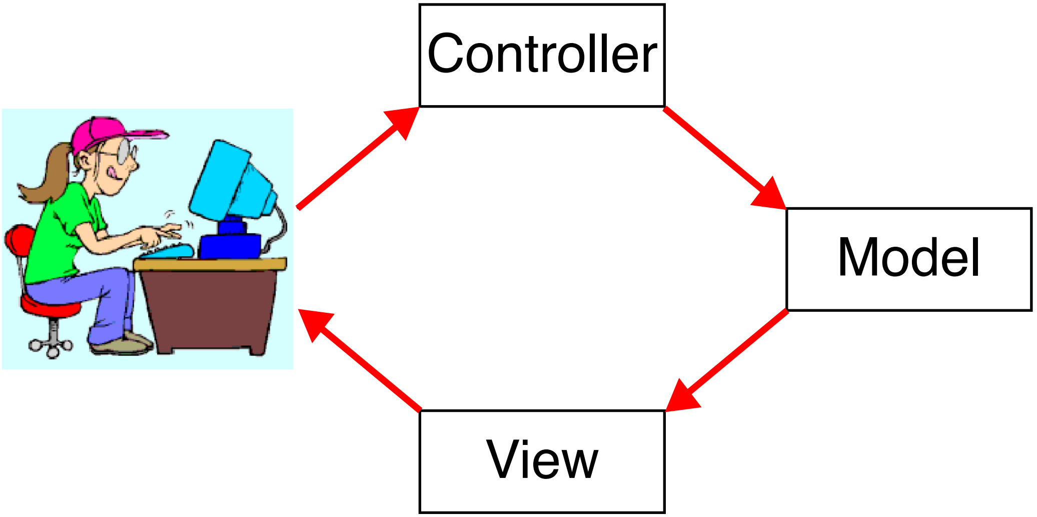
Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



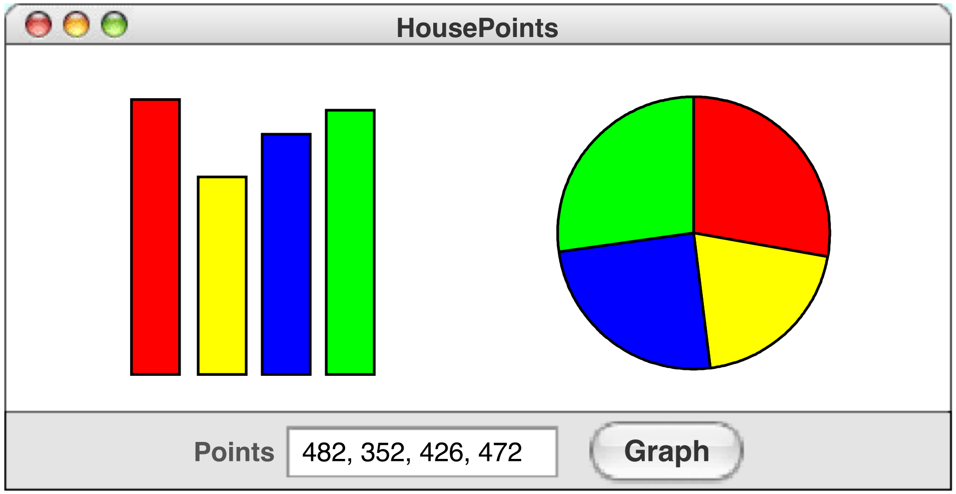
Controller

Model

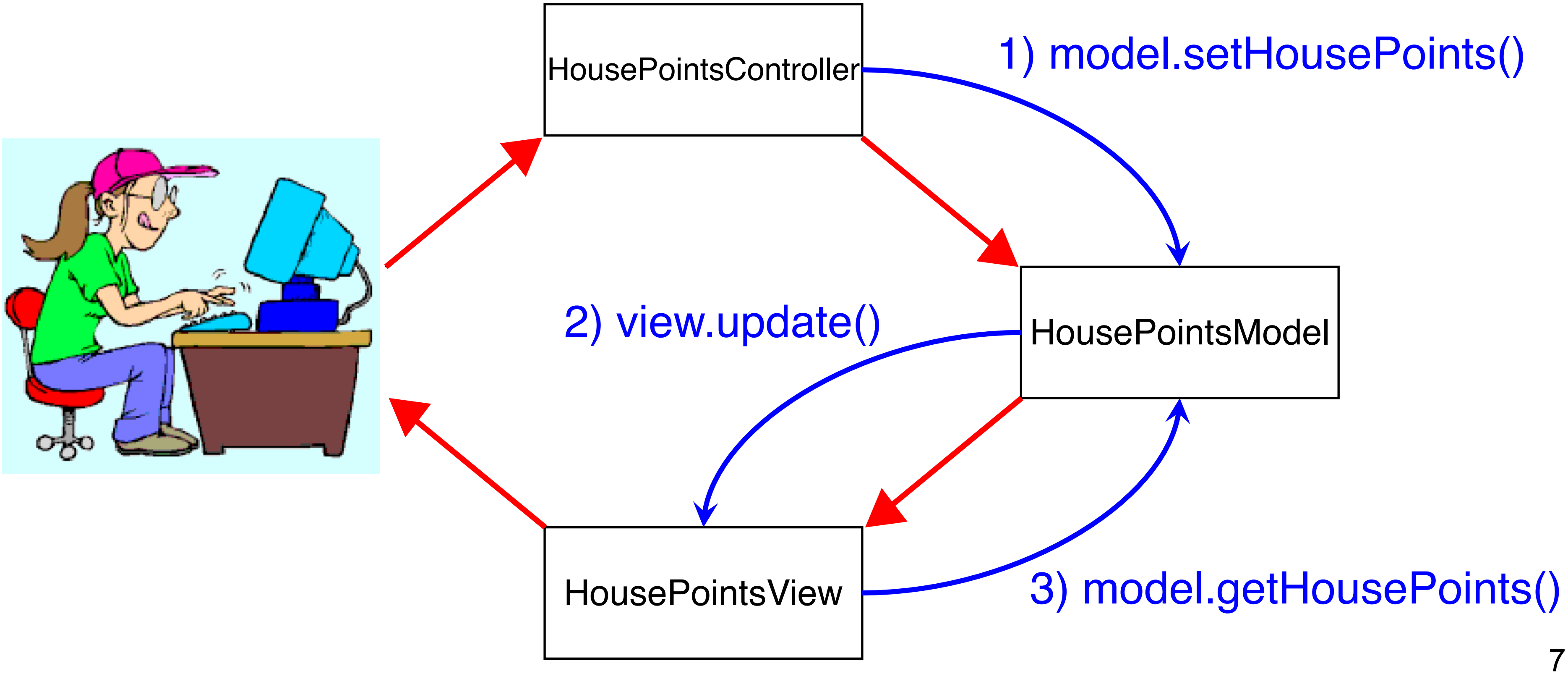
View

- Exact definition of MVC varies.
- **Main point:** think about *some* useful separation of concerns.
- That separation should be reflected in class structure

MVC in HousePoints



Points 482, 352, 426, 472



HousePointsController

HousePointsModel

HousePointsView

1) model.setHousePoints()

2) view.update()

3) model.getHousePoints()

```
public class HousePoints extends GraphicsProgram {
  // Constants that define the size of the views
  private double GRAPHWIDTH = 400;
  private double GRAPHHEIGHT = 400;

  public void init() {
    // Create a model
    HousePointsModel model = new HousePointsModel();

    // Create bar graph view
    BarGraphView barview = new BarGraphView(GRAPHWIDTH, GRAPHHEIGHT);
    model.addView(barview);
    add(barview);

    // Create pie chart view
    PieChartView pieview = new PieChartView(GRAPHWIDTH, GRAPHHEIGHT);
    model.addView(pieview);
    add(pieview, GRAPHWIDTH, 0);

    // The panel where the controller places its interactors
    JPanel controllerPanel = getRegionPanel(SOUTH);

    // Create controller
    HousePointsController controller = new HousePointsController(model, controllerPanel);
  }
}
```

```
package HousePoints;
/*
 * File: HousePointsModel.java
 * ---------------------------
 * This class keeps track of the data in the array but is not
 * responsible for the actual display.  Whenever the controller
 * resets the data array, the model notifies all registered views.
 */

import java.util.*;

public class HousePointsModel {

  /** Private instance variables */
  private int[] housePoints;
  private ArrayList<HousePointsView> views;

  /** Creates a new HousePointsModel with no views */
  public HousePointsModel() {
    housePoints = new int[0];
    views = new ArrayList<HousePointsView>();
  }

  /** Adds a view to the list of views for this model */
  public void addView(HousePointsView view) {
    views.add(view);
  }
/** Sets the house points data to the contents of the integer array */
  public void setHousePoints(int[] points) {
    housePoints = new int[points.length];
    for (int i = 0; i < points.length; i++) {
      housePoints[i] = points[i];
    }
    notifyViews();
  }

  /** Returns a copy of the internal house points data */
  public int[] getHousePoints() {
    int[] points = new int[housePoints.length];
    for (int i = 0; i < points.length; i++) {
      points[i] = housePoints[i];
    }
    return points;
  }

  /** Calls update(this) on every view to reconstruct their displays */
  private void notifyViews() {
    for (HousePointsView view : views) {
      view.update(this);
    }
  }
}
```

```
package HousePoints;
/*
 * File: HousePointsView.java
 * --------------------------
 * This abstract class defines the operations that any specific
 * view class must support.  Each HousePointsView is a GCompound
 * that responds to update messages from the model.
 */

import acm.graphics.*;
import java.awt.*;

public abstract class HousePointsView extends GCompound {

  /* Private constants */
  private static final Color[] COLORS = { Color.RED, Color.YELLOW, Color.BLUE,
Color.GREEN, Color.PINK, Color.CYAN,
      Color.MAGENTA, Color.ORANGE };

  /* Private instance variables */
  private GRect background;
/** Creates a new HousePointsView with a given model and size */
  public HousePointsView(double width, double height) {
    background = new GRect(width, height);
    background.setFilled(true);
    background.setColor(Color.WHITE);
  }

  /** Each subclass must define a method to create the graph */
  public abstract void createGraph(int[] data);

  /** Updates the display image from the model */
  public void update(HousePointsModel model) {
    removeAll();
    add(background);
    createGraph(model.getHousePoints());
  }

  /** Returns a color to use for the kth data value */
  public Color getColorForIndex(int k) {
    return COLORS[k % COLORS.length];
  }
}
```

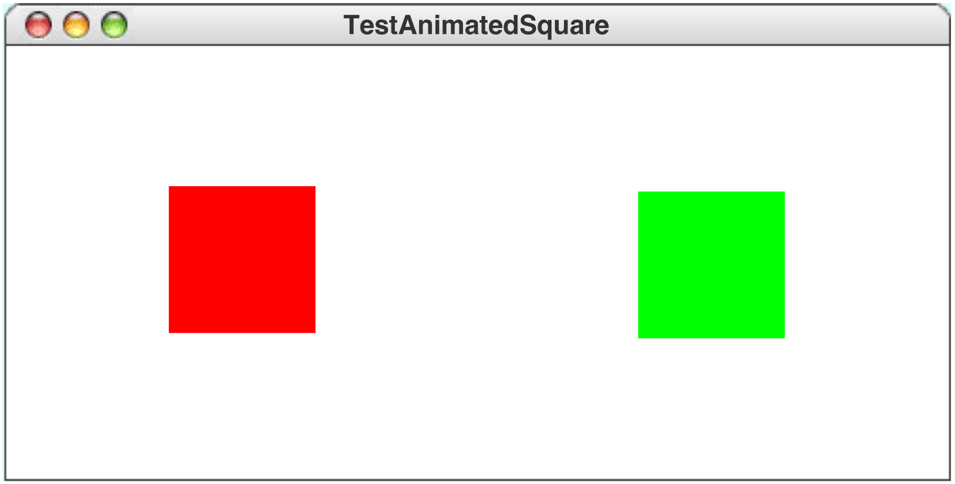
```
package HousePoints;
/*
 * File: BarGraphView.java
 * -----------------------
 * This class represents a concrete implementation of the
 * HousePointsView class that builds a bar chart.  The chart is
 * scaled so that the maximum value fills the vertical space.
 */

import acm.graphics.*;

public class BarGraphView extends HousePointsView {

  /* Private constants */
  private static final double BAR_WIDTH = 20;


  /** Creates a new BarGraphView */
  public BarGraphView(double width, double height) {
    super(width, height);
  }

  /** Arranges the data as a set of bars */
  public void createGraph(int[] data) {
    int n = data.length;
    double max = maxIntArray(data);
    if (max == 0)
      return;
    double sep = (getWidth() - n * BAR_WIDTH) / (n + 1);
    for (int i = 0; i < n; i++) {
      double height = data[i] / max * getHeight();
      double x = i * (BAR_WIDTH + sep);
      double y = getHeight() - height;
      GRect bar = new GRect(x, y, BAR_WIDTH, height);
      bar.setFilled(true);
      bar.setFillColor(getColorForIndex(i));
      add(bar);
    }
  }

  /* Returns the maximum value of an integer array (or 0 if empty) */
  private int maxIntArray(int[] array) {
    if (array.length == 0)
      return 0;
    int largest = array[0];
    for (int val : array) {
      largest = Math.max(largest, val);
    }
    return largest;
  }
}
```

```
package HousePoints;
/*
 * File: HousePointsController.java
 * ----------------------
 * The controller part of the HousePoints program.
 */

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.StringTokenizer;
import javax.swing.*;

public class HousePointsController implements ActionListener {

  /**
   * These are ivars because they are shared between the methods.
   *
   */
  private JTextField intsField;
  private HousePointsModel model;
```

```
/** Constructor */
public HousePointsController(HousePointsModel model, JPanel panel) {
  this.model = model;

  // The text field interactor where the user enters points
  panel.add(new JLabel("Points: "));
  intsField = new JTextField(20);
  panel.add(intsField);
  intsField.addActionListener(this);
  // Hitting enter in the text field also triggers an update
  intsField.setActionCommand("Graph");

  // The button that triggers an update of the model
  JButton graphButton = new JButton("Graph");
  panel.add(graphButton);
  graphButton.addActionListener(this);
 }

/** The action performed when hitting enter or the button. */
 public void actionPerformed(ActionEvent e) {
  if (e.getActionCommand().equals("Graph")) {
    String line = intsField.getText();
    int[] housePoints = parseInts(line);
    model.setHousePoints(housePoints);
  }
 }
```

```
  /** Parse four integers. Default values are 0. */
  private int[] parseInts(String line) {
    int[] housePoints = new int[4];
    StringTokenizer tokenizer = new StringTokenizer(line, ", ");
    for (int i = 0; (i < 4) && tokenizer.hasMoreTokens(); i++) {
      housePoints[i] = Integer.parseInt(tokenizer.nextToken());
    }
    return housePoints;
  }
}
```

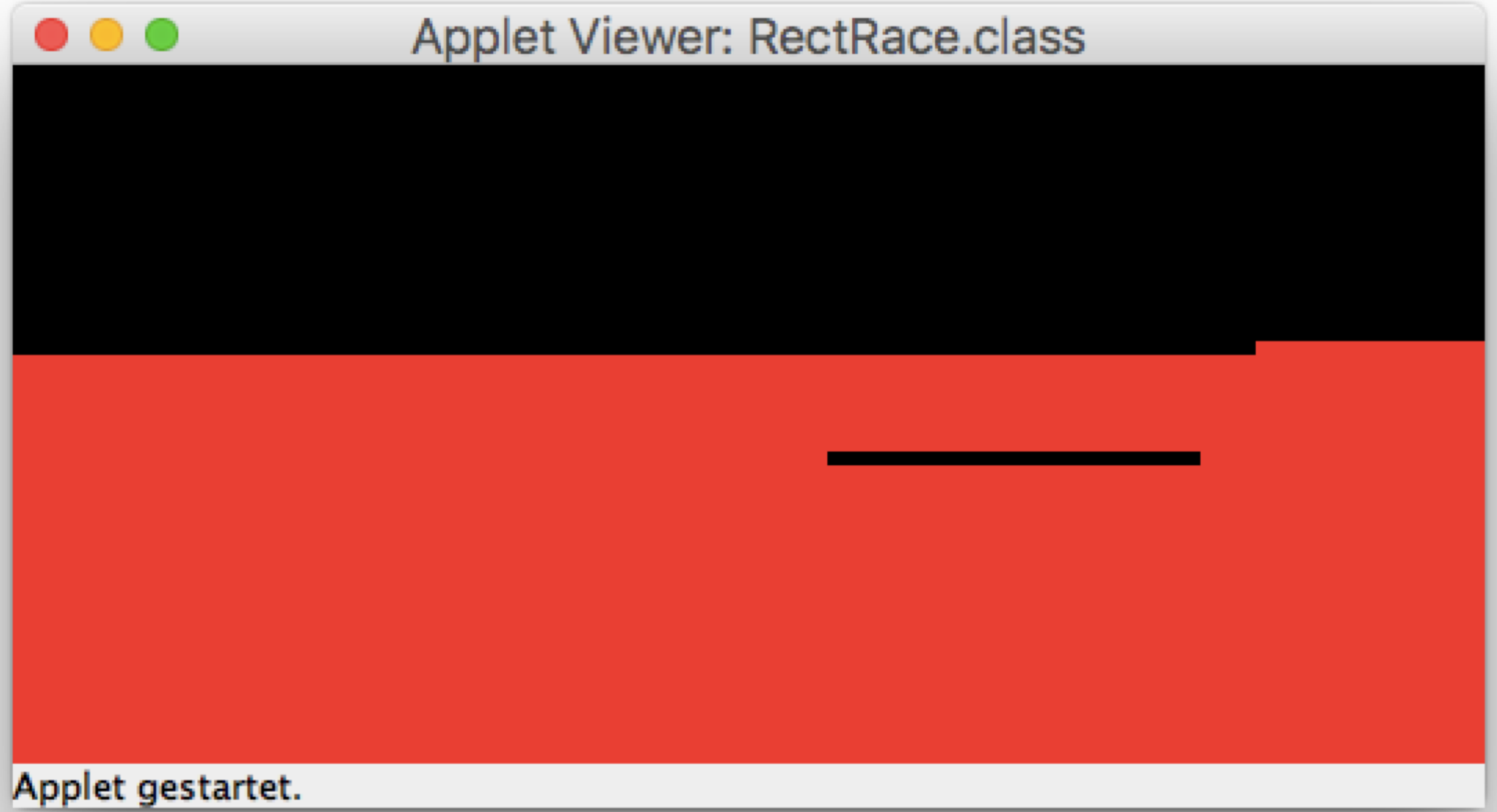
Concurrency



```
public class AnimatedSquare extends GRect
    implements Runnable {

  private static final double DELTA = 2;
  private static final int PAUSE_TIME = 20;
  private static final int CHANGE_TIME = 50;

  private RandomGenerator rgen =
      RandomGenerator.getInstance();
  private double direction;

  public AnimatedSquare(double size) {
    super(size, size);
  }

  public void run() {
    for (int t = 0; true; t++) {
      if (t % CHANGE_TIME == 0) {
        direction = rgen.nextDouble(0, 360);
      }
      movePolar(DELTA, direction);
      pause(PAUSE_TIME);
    }
  }
}
```

```
public class TestAnimatedSquare extends GraphicsProgram {

  private static final double SIZE = 75;

  public void run() {
    double x1 = getWidth() / 3 - SIZE / 2;
    double x2 = 2 * getWidth() / 3 - SIZE / 2;
    double y = (getHeight() - SIZE) / 2;
    AnimatedSquare redSquare = new AnimatedSquare(SIZE);
    redSquare.setFilled(true);
    redSquare.setColor(Color.RED);
    add(redSquare, x1, y);
    AnimatedSquare greenSquare = new AnimatedSquare(SIZE);
    greenSquare.setFilled(true);
    greenSquare.setColor(Color.GREEN);
    add(greenSquare, x2, y);
    Thread redSquareThread = new Thread(redSquare);
    Thread greenSquareThread = new Thread(greenSquare);
    waitForClick();
    redSquareThread.start();
    greenSquareThread.start();
  }
}
```

Race Conditions – Graphically



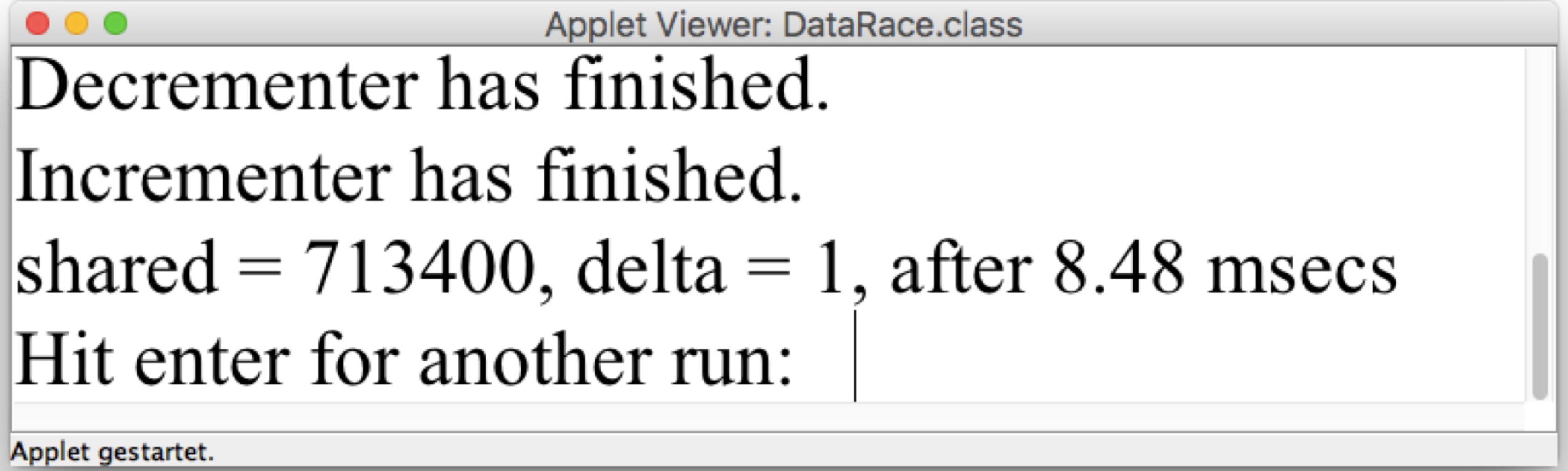
Applet gestartet.

```
public class RectRace extends GraphicsProgram
    implements ComponentListener {

  private static final int PIX = 5; // Size of pixel

  // Enable listening to component resizing
  public void init() {
    addComponentListener(this);
  }

  // Whenever we get resized, start the painting threads
  public void componentResized(ComponentEvent e) {
    RectPaint paintRed = new RectPaint(Color.RED);
    RectPaint paintBlack = new RectPaint(Color.BLACK);

    Thread threadRed = new Thread(paintRed);
    Thread threadBlack = new Thread(paintBlack);

    threadRed.start();
    threadBlack.start();
  }
```

```
// Repaint the canvas in given color
// Make this "synchronized" for consistent canvas
private void repaint(Color color) {
  int width = getWidth();
  int height = getHeight();

  for (int y = 0; y < height; y += PIX) {
    for (int x = 0; x < width; x += PIX) {
      GRect pixel = new GRect(PIX, PIX);
      pixel.setColor(color);
      pixel.setFilled(true);
      add(pixel, x, y);
    }
  }
}
```

```
  // The repainting thread class
  class RectPaint implements Runnable {

    RectPaint(Color color) {
      this.color = color;
    }

    public void run() {
      repaint(color);
    }

    private Color color;
  }
}
```

Race Conditions – With Data



```
Decrementer has finished.
Incrementer has finished.
shared = 713400, delta = 1, after 8.48 msecs
Hit enter for another run:
```

```
class Adder implements Runnable {

    private String name;
    private int threadDelta;
    static private final int ITER_CNT = 1000000;

    public Adder(String name, int threadDelta) {
      this.name = name;
      this.threadDelta = threadDelta;
    }

    public void run() {
      for (int i = 0; i < ITER_CNT; i++) {
        addDelta(threadDelta);
      }
      println(name + " has finished.");
    }
  }
}
```

```
public class DataRace extends ConsoleProgram {
  private int shared, delta;

  public void run() {
  while (true) {
      readLine("Hit enter for another run: ");
      long start = System.nanoTime();
      shared = 0;

      runThreads();

      long time = System.nanoTime() - start;
      println("shared = " + shared +
          ", delta = " + delta + ", after " +
          time / 1000 / 1e3 + " msecs");
    }
  }
```

```
private void runThreads() {
   Adder incr = new Adder("Incrementer", 1);
   Adder decr = new Adder("Decrementer", -1);

   Thread incrThread = new Thread(incr);
   Thread decrThread = new Thread(decr);

   decrThread.start();
   incrThread.start();

   try {
     incrThread.join();
     decrThread.join();
   } catch (InterruptedException e) {
     throw new ErrorException(e);
   }
 }
```

```
private void addDelta(int d) {
  delta = d;
  shared += delta;
}
```

- `addDelta()` is not *thread-safe*
- To avoid interruption, make it `synchronized`
- However, this still does not resolve write-write race on `delta`!
- Concurrency offered by Java (and most other programming languages) is *not deterministic*!
- For deterministic concurrency, consider e.g. *data-flow languages* or *synchronous languages*

For more details:
Edward A. Lee, *The Problem with Threads*, Computer, vol 39, issue 5, May 2006 (see also https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf)

Periodic Tasks

```
private Timer timer;
private static final int BALL_CYCLE = 25;

public BallModel() {
  ...
  timer = new Timer(BALL_CYCLE);
}

public void run() {
  while (true) {
    move();
    timer.pause();
  }
}
```

```
public class Timer {
  private double nanoTime;
  private double period;
  private double delay;
  private static final double
    DAMPENING = 0.1;

  public Timer(double period) {
    this.period = period;
    reset();
  }

  public void reset() {
    nanoTime = System.nanoTime();
    delay = period;
  }
  public void pause() {
    double preNanoTime = nanoTime;
    nanoTime = System.nanoTime();
    double cycleTime =
      (nanoTime - preNanoTime) / 1e6;
    double rawDelayAdjustment =
      period - cycleTime;
    double delayAdjustment =
      DAMPENING * rawDelayAdjustment;
    delay += delayAdjustment;
    if (delay > 0) {
      JTFTools.pause(delay);
    }
  }
}
```

Summary I

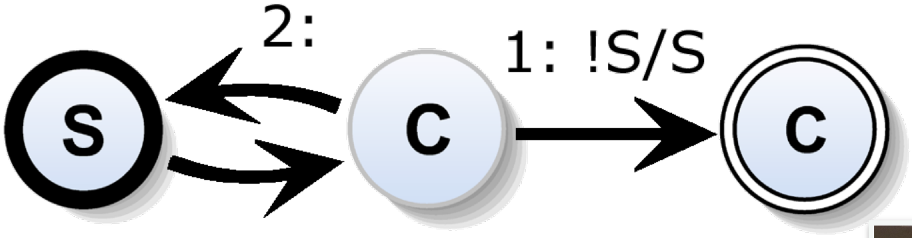
- Good software engineering makes use of *patterns*, as proposed by *Gang of Four*
- One important pattern is *Model-View-Controller* (*MVC*)
- Concurrency is an powerful yet tricky programming concept
- The Java event model already entails concurrency
- Java also supports concurrency with user-level *threads*
- A thread can be constructed with a runnable object, which is constructed from a class that implements the **`Runnable`** interface

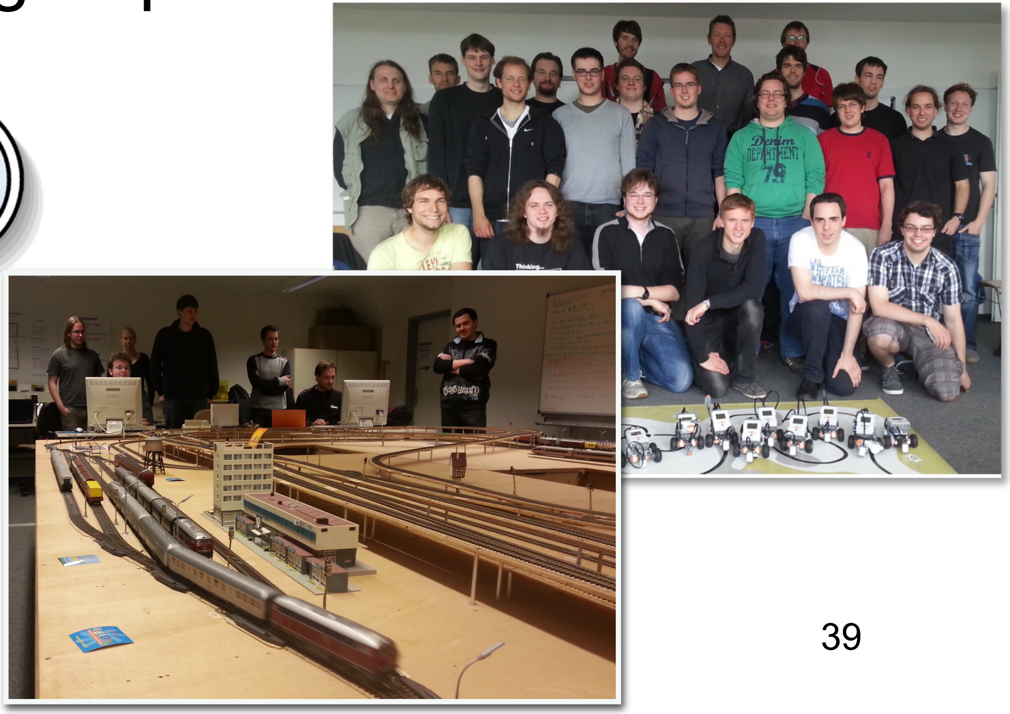
Summary II

- The runnable object needs a **`run`** method
- After creating a thread with a Thread constructor, we must still explicitly **`start`** it
- We can wait for completion of a thread with **`join`**
- The concurrent use of shared may result in *write-write races* or *write-read races*
- Some, but not all, race conditions can be avoided by acquiring a *monitor lock* (or simply *lock*) on an object
- One way to get a lock on an object is to call a **`synchronized`** method of it

Summary III

- For *real* solutions to the concurrency problem, ensuring determinism, consider other languages, such as *synchronous languages*
- See further advanced lectures, including those offered by the RTSYS group ☺





[Wikimedia Commons]

We Want **You** for future InfProgOO !

Apply now to rvh@...

Programming Problem Pouncer Prize

This is to certify that

...

has achieved **... out of 1210** Points

in the problem sets of the Imperative and Object-Oriented Programming
(InfProg OO) class conducted in the Winter Semester 2019/20
at the Department of Computer Science, Kiel University,
the fifth-highest score among 340 class participants.

1st

Kiel, January 29th, 2020

Prof. Reinhard von Hanxleden