Design Patterns: Observer and MVC

[Skrien §8.0] Skrien uses a case study of a figure-drawing application to introduce several important design patterns.

Today we will consider two of them,

• Observer, and
• Model/View/Controller

Both of these are used to allow the GUI of an application to communicate with the “business logic.”

The figure-drawing application

The figure-drawing application was introduced in Section 3.8 of the text, and we first saw it in Lecture 8.

Here is the window in which the application runs:

Note that there is a toolbar across the top with three tools.

The rest of the window is a canvas on which figures can be drawn.

At any given time, one of the tools will be selected.

When the user clicks on the canvas, a new figure of fixed size will appear where the mouse was clicked.
Let’s take (just) a quick look at the code. We will modify it as we go along.

```java
package drawer0;

import javax.swing.*;
import java.awt);

public class DrawingFrame extends JFrame {
    public DrawingFrame() {
        super("Drawing Application");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Container contentPane = getContentPane();

        JComponent drawingCanvas =
            createDrawingCanvas();
        contentPane.add(drawingCanvas, BorderLayout.CENTER);

        JToolBar toolbar = createToolBar();
        contentPane.add(toolbar, BorderLayout.NORTH);
    }

    private JComponent createDrawingCanvas() {
        JComponent drawingCanvas = new JPanel();
        drawingCanvas.setPreferredSize(new Dimension(400, 300));
        drawingCanvas.setBackground(Color.white);
        drawingCanvas.setBorder(BorderFactory.createEtchedBorder());
        return drawingCanvas;
    }

    private JToolBar createToolBar() {
        JToolBar toolbar = new JToolBar();
        JButton ellipseButton =
            new JButton("Ellipse");
        toolbar.add(ellipseButton);
        JButton squareButton = new
            JButton("Square");
        toolbar.add(squareButton);
        JButton RectButton = new JButton("Rect");
        toolbar.add(RectButton);
    }
```
public static void main(String[] args) {
    DrawingFrame drawFrame = new DrawingFrame();
    drawFrame.pack();
    drawFrame.setVisible(true);
}

How should this application handle user input? Well, what kind of user input is there?

- 

Clicking on a tool causes

How should the canvas be informed of clicks on a tool?

Well, should the canvas be informed directly? It is the canvas’s job to display the figures.

It would be better to have another class, say, CanvasEditor, handle interaction with the user.

So it is CanvasEditor that needs to handle the interactions.

Let’s assume that the three buttons are instances of a class called ToolButton.

A ToolButton is like a JButton except that it keeps track of the time when it was last clicked.

Here is a class diagram showing how ToolButtons relate to the CanvasEditor:
Now, when it’s time to draw a figure, the CanvasEditor just needs to poll the buttons to determine which was clicked most recently.

Here’s a UML collaboration diagram that shows how this is done. Names that are underlined in the diagram refer to objects, not classes.

![Collaboration Diagram]

This approach is rather awkward. Why?

Instead of the CanvasEditor keeping track of the ToolButtons, perhaps the ToolButtons could keep track of the CanvasEditor.

How would this help?

Here’s a class diagram for this approach:

![Class Diagram]

However, the classes ToolButton and CanvasEditor classes are still tightly coupled together.
Observer

We can reduce the coupling by using the Observer pattern, which will allow ToolButtons and CanvasEditors to be unaware of what type of objects they are communicating with.

The easiest way of understanding the Observer pattern is to consider its alternate name, Publish-Subscribe.

- Publishers produce information that is of interest to other objects in the system.
- An object that is interested in a piece of this information subscribes to this publisher.

Then when an event of interest happens in the system, a publisher sends it to its subscribers.

Subscribers can also be thought of as “observers,” and hence the name of the pattern.

In our example, who’s the publisher and who’s the subscriber?

In Java, many observer classes are called listeners.

Since our ToolButton no longer needs to maintain the time at which it was last clicked, it can be a JButton instead. Event handling for a JButton is performed by an ActionListener.

Here’s the class diagram for that:
Here’s the `ActionListener` interface:

```java
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```

Here is the code in `CanvasEditor` that implements the `ActionListener` interface:

```java
public class CanvasEditor implements ActionListener {
    private JButton selectedButton; // instance var.

    public CanvasEditor(JButton initialButton) {
        this.selectedButton = initialButton;
    }

    public void actionPerformed(ActionEvent e) {
        selectedButton = (JButton) e.getSource();
    }
}
```

Here is the code that registers the observers.
CanvasEditor CanvasEditor = new
CanvasEditor(ellipseButton);
ellipseButton.addActionListener(CanvasEditor);
squareButton.addActionListener(CanvasEditor);
rectButton.addActionListener(CanvasEditor);

Which class is it in?
Which method is it in?

How does this approach reduce coupling between the buttons and the CanvasEditor?

Here's a synopsis of the Observer pattern.

**Observer**

*Intent:* Define a one-to-many dependence between objects so that when one object changes state, all its dependents are notified automatically.

*Problem:* A varying list of objects needs to be notified that an event has occurred.

*Solution:* Observers delegate the responsibility for monitoring for an event to a central object, the Subject.

*Implementation:* • Have objects (Observers) that want to know when an event happens attach themselves to another object (Subject) that is watching for the event, or that triggers the event itself.
• When the event occurs, the Subject tells the list of Observers that it has occurred.

How would you use the Observer pattern in Program 2?

What other kind of user input—besides clicking on drawing tools—does our application need to handle?
These events can be handled similarly, but by a different kind of listener. Which?

Here’s our first attempt at the code:

```java
public void mouseClicked(MouseEvent e) {
    //handle clicks in the canvas
    int x = e.getX();
    int y = e.getY();
    JPanel canvas = (JPanel) e.getSource();
    if (currentButton.getText().equals("Ellipse"))
        canvas.getGraphics().drawOval(x - 30, y - 20, 60, 40);
    else if
        (currentButton.getText().equals("Rect"))
        canvas.getGraphics().drawRect(x - 30, y - 20, 60, 40);
    else if
tcanvas.getGraphics().drawRect(x - 25, y - 25, 50, 50);
}
```

What’s not nice about this code?

A good way to get around this problem is to have an abstract class `Figure` with subclasses for the various kinds of figures.

The code, except for accessor methods, is shown below.

```java
package drawer2.figure;
import java.awt.*;
```
public abstract class Figure
{
    private int centerX, centerY; //center coords
    private int width;
    private int height;

    public Figure(int centerX, int centerY,
                  int w, int h)
    {
        this.centerX = centerX;
        this.centerY = centerY;
        this.width = w;
        this.height = h;
    }

    // accessor methods ...

    public abstract void draw(Graphics g);
}

public class Ellipse extends Figure {
    public Ellipse
    (int centerX, int centerY, int w, int h)
    {
        super(centerX, centerY, w, h);
    }

    public void draw(Graphics g) {
        int width = getWidth();
        int height = getHeight();
        g.drawOval(getCenterX() - width/2,
                   getCenterY() - height/2,
                   width, height);
    }
}

public class Rect extends Figure {
    public Rect
    (int centerX, int centerY, int w, int h)
    {
        super(centerX, centerY, w, h);
    }
}
public void draw(Graphics g) {
    int width = getWidth();
    int height = getHeight();
    g.drawRect(getCenterX() - width/2,
               getCenterY() - height/2,
               width, height);
}

public class Square extends Rect {
    public Square(int centerX, int centerY, int w) {
        super(centerX, centerY, w, w);
    }
}

Unfortunately, this just moves the conditional from the
mouseClicked method to the actionPerformed method:

    public void actionPerformed(ActionEvent e) {
        JButton currentButton =
            (JButton) e.getSource();
        if (currentButton.getText().equals
            ("Ellipse"))
            currentFigure =
                new Ellipse(0, 0, 60, 40);
        else if (currentButton.getText().equals
                  ("Rect"))
            currentFigure =
                new Rect(0, 0, 60, 40);
        else //if( currentButton.getText().equals
                // ("Square") )
            currentFigure = new Square(0, 0, 50);
    }

This problem can be solved by giving each of the buttons their own
ActionListener (instead of making the CanvasEditor the
ActionListener for all buttons):

    ellipseButton.addActionListener(canvasEditor);
ellipseButton.addActionListener
(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        canvasEditor.setCurrentFigure
            (new Ellipse(0, 0, 60, 40));
    }
});

Model-View-Controller

The Observer pattern tells us how to get changes in the “business
data” of a program reflected in the various views presented to the
user.

But this does not completely support the way a graphical application
works, because some of the changes that affect views are made by
the user.

Should the user “talk to” the model or the view?

For example, in the drawing application, we have the following
components:

- The Drawing, consisting of the collection of figures
currently on the canvas.
- The DrawingCanvas, which renders the figures and
  makes them visible to the user.
- The CanvasEditor, which deals with user input.

The MVC pattern has a name for each of these.

These names come originally from Smalltalk:

- Model was the superclass of all “things that modeled the
  world”. Your application would be a Model subclass, as
  would be many other classes in your system.
• **View** was the superclass of all things graphical. The class View had many, many subclasses that would display text (TextView), selection buttons (BooleanView), and just about any other graphical object you can think of.

• Controllers are objects that control how mouse movements and keyboard actions are interpreted. Controllers work in concert with Views to show the effects of user actions.

  For instance, ParagraphController worked with TextView to interpret keystrokes as free text, and mouse movements as selection and highlighting.

MVCs came in a triad, with communication between the components occurring as the following diagram illustrates.

MVC allows us to …

  • provide several views to display the same data in different forms,
  
  • add several controllers, each for handling a different kind of interaction.

Model, view, and controller usually interact via the Observer pattern:
In our application, we will use a variation on MVC.

1. Since the view needs to display the data, the view will maintain a reference to the model. Our view will consist of a new `DrawingCanvas` subclass of `JPanel` that has an instance variable corresponding to the drawing and that has a `paint` method for drawing the figures stored in that model.

2. Since the controller needs to interpret user actions on the canvas, the controller is a `MouseListener` for the canvas, and so the canvas maintains a link to the controller through the canvas' list of listeners.

3. The controller needs to tell the model to change its data based on user actions. The controller will access the model by a two-step process by
   a. getting the canvas from the `MouseEvent` object and then
   b. by telling the canvas to modify the drawing.