Design Patterns: Observer and Prototype

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

Today we will cover two of them, Observer and Prototype.

Like MVC, which is built from it, Observer is 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 3.9 of the text, and we first saw it in Week 10.

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.
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);
        return toolbar;
    }

    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
• Clicking in the drawing canvas

Clicking on a tool causes the corresponding shape to be drawn when the user clicks on the canvas.

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.
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:

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? The ToolButton is the publisher, and the CanvasEditor is the subscriber (or observer).

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:

```
<< interface >>

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

Here’s the ActionListener interface:
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.

```java
CanvasEditor canvasEditor = new CanvasEditor(ellipseButton);
ellipseButton.addActionListener(canvasEditor);
squareButton.addActionListener(canvasEditor);
rectButton.addActionListener(canvasEditor);
```

**Which class** is it in? `DrawingFrame`.

**Which method** is it in? `createToolbar`.

How does this approach reduce coupling between the buttons and the `CanvasEditor`? The buttons don’t care who’s subscribing to them. `CanvasEditor` no longer needs to know that it has buttons for observers.

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.

Exercise: Can you come up with an example of Observer that does not involve user interfaces?

When there is a reply to a post on piazza an email gets sent to the people who followed the post. It’s not dependent on events on the User Interface.

Whenever a theft is reported, a Wolfalert mail is sent to all the subscribers.

When I want to get updates from a particular feed, I add it to my feed reader. Any time that the RSS feed has an update, it will appear in my reader automatically. This is the Observer pattern in action, a publisher/subscriber relationship with one source having many subscribers.

What other kind of user input—besides clicking on drawing tools— does our application need to handle? Clicking in the drawing canvas.

These events can be handled similarly, but by a different kind of listener. Which? MouseListener.

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"))
```

```java
```
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( currentButton.getText().equals
    //("Square") )
  canvas.getGraphics().drawRect
  (x - 25, y - 25, 50, 50);
}

What’s not nice about this code? The case statement based on classes.

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;
  }
```

Figure

Rect

Ellipse

Square
// 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:

```java
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):

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

*Exercise: Submit the code for `SquareButton.addActionListener`.*

**Prototype**

[Skrien §8.5] Let’s reconsider the `CanvasEditor`. 
When a button is clicked, the button’s custom `ActionListener` is invoked, and it sets the current figure to the proper type:

```java
rectButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        canvasEditor.setCurrentFigure
            (new Rect(0, 0, 60, 40));
    }
});
```

The `setCurrentFigure` message is received by the `canvasEditor` object. Let’s take a look at the code for it.

```java
public class CanvasEditor implements MouseListener {
    private Figure currentFigure;
    public CanvasEditor(Figure initialFigure) {
        this.currentFigure = initialFigure;
    }

    public void setCurrentFigure(Figure newFigure) {
        currentFigure = newFigure;
    }

    public void mouseClicked(MouseEvent e) {
        Figure newFigure = ... a new figure based on currentFigure...
        ((DrawingCanvas) e.getSource()).
            addFigure(newFigure);
    }

    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
}
```

Notice that we haven’t said how to draw a new figure based on the current figure.
How shall we draw this figure? Can we use the currentFigure object? No; it’s a Singleton; we only have one figure for each type. And we need another copy each time we click somewhere else on the canvas.

Can we use a constructor to create it? Well, which constructor would we use? We would need a case statement …

Should we test which kind of figure we need?

```java
if( currentFigure instanceof Ellipse )
    newFigure = new Ellipse(...);
else if( currentFigure instanceof Rect )
    newFigure = new Rect(...);
else
    newFigure = new Square(...);
```

Of course not!

Instead, we can just clone the current object. This makes use of Java’s Cloneable interface.

By implementing Cloneable, a class allows field-by-field copies to be made of its instances.

This allows the Figure referred to by currentFigure to act as a prototype for the figure to be created. Here is the code.

```java
public Object clone() {
    try {
        return super.clone();
    } catch (CloneNotSupportedException e) {
        // This should never happen
        e.printStackTrace();
        return null;
    }
}
```

**Note:** Object here is a return type, not the class where the impl. is Why does the method call super.clone()? It is invoking the clone method of class Object, which is where clone() is defined.
What class do we place this definition in?  **Figure.**

How do we use this method to create “... a new figure based on currentFigure...”?  

```java
  public void mouseClicked(MouseEvent e) {
    Figure newFigure = (Figure) currentFigure.clone();
    newFigure.setCenter(e.getX(), e.getY());
    ((DrawingCanvas) e.getSource()).addFigure(newFigure);
  }
```

This is an instance of the Prototype pattern.

**Prototype**

*Intent:*
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

*Problem:*
Customized objects must be created, but we do not know their class or any details of how to create them.

*SOLution:*
Each class that might have to create an object has a prototypical object. A creation-initiating object is invoked, and it creates objects by asking one of the prototypical objects to make a copy of itself.

*Implementation:*
Each class that is going to create an object must have a prototypical object. The parent class implements the `Cloneable` interface. Then the creation-initiating object clones itself, producing an object of the correct class.

*Exercise:*
Look up on the Web the differences between the Abstract Factory pattern and the Prototype pattern.

"The Prototype pattern builds objects by copying and customizing a prototype of the object."

"The intent of an Abstract Factory is simply to provide an interface for creating a set of related, possibly interdependent objects."