Command/Adapter Pattern Assignment

Design Patterns

Jacob Hartman

6 October 2016

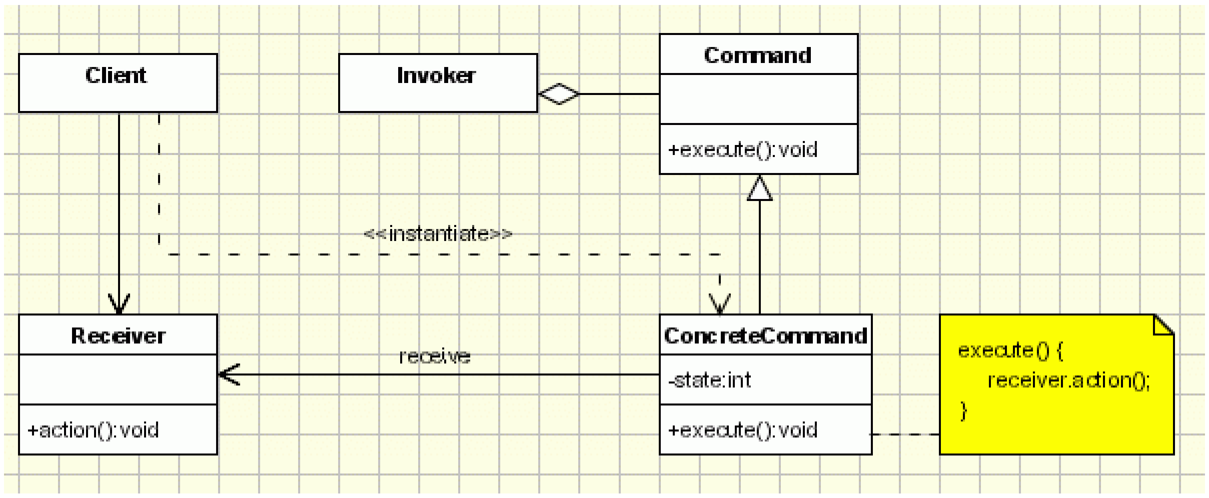
Introduction

This assignment required the creation of a program that implements the command pattern. At least two commands were required to be implemented. The assignment also required implementation of the adapter pattern.

The application I wrote for this assignment is a virtual sketchpad. The user can draw rectangles and circles on the main panel. The program includes undo and redo commands.

UML Diagram - Command Pattern

image taken from http://www.oodesign.com/command-pattern.html

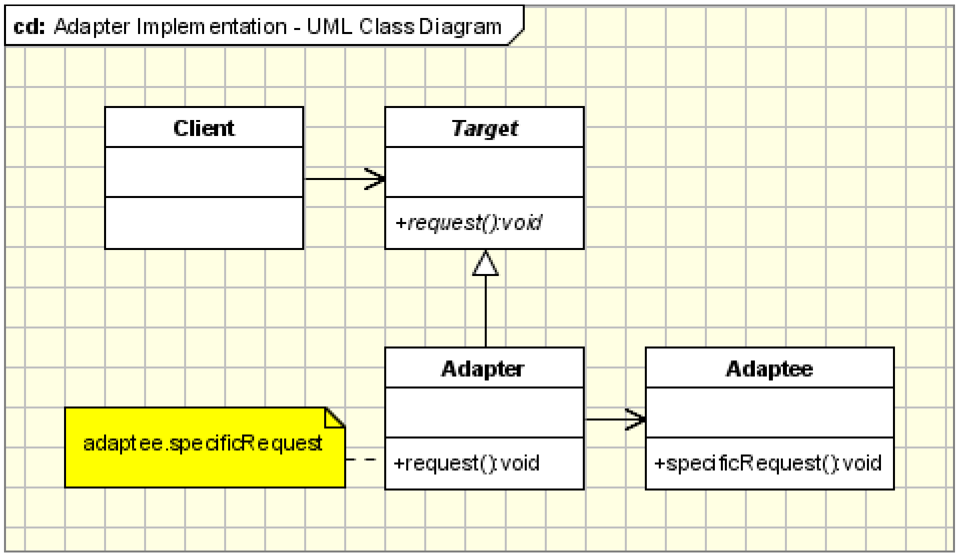


Above is the UML Diagram for the command pattern. It shows the need for an Invoker class, an abstract Command class, a ConcreteCommand class, and a Receiver class in addition to the client. The ConcreteCommand class must implement the Command abstract class, also containing an execute method. The execute method must communicate with the receiver's action method. The client and ConcreteCommand classes must communicate with the Receiver. Lastly, the client must instantiate Command objects.

My receiver class is the Adapter class. I created an invoker class called Invoker. I also wrote a Command abstract class and two ConcreteCommand classes called RectCommand and CircleCommand.

UML Diagram - Adapter Pattern

image taken from http://www.oodesign.com/adapter-pattern.html



Above is the UML Diagram for the adapter pattern. It shows the need for an Adaptee class, and Adapter class, and a Target class in addition to the client. The Adaptee class must contain a specificRequest method called by the Adapter's request method.

I created the Adapter class, and the System.Drawing.Graphics class is my adaptee. The client contains the target.

Narrative

First, I created the Shape class to simplify the passing of arguments and data storage.

```
public class Shape
{
    string shapeName;
    Pen drawingColor;
    int topX;
    int topY;
    int xLen;
    int yLen;
    public Shape(string shape, Pen color, int x1, int y1, int length, int width)
    {
        shapeName = shape;
```

```
            drawingColor = color;
            topX = x1;
            topY = y1;
            xLen = length;
            yLen = width;
        }

        public string getShape()
        {
            return shapeName;
        }

        public Pen getColor()
        {
            return drawingColor;
        }

        public int getTopX()
        {
            return topX;
        }

        public int getTopY()
        {
            return topY;
        }

        public int getLength()
        {
            return xLen;
        }

        public int getHeight()
        {
            return yLen;
        }
    }
```

Each shape contains the name of the shape, a color with which to draw the shape, an X value and Y value for the shape's origin, and a length and height value.

```
    public abstract class Command // this is the Command class

    {
        public abstract void draw(); // this is the execute method
    }
```

The command class contains a draw method, which is the execute method from the UML Diagram. This method is implemented in the ConcreteCommands.

```
    public class RectCommand : Command // this is a ConcreteCommand class

    {
        Shape rectangle;
        Adapter adapter;
        public RectCommand(Pen color, int x1, int y1, int width, int height, Adapter a)
        {
            rectangle = new Shape("rect", color, x1, y1, width, height);
            adapter = a;
        }
        public override void draw()
        {
            adapter.addToList(rectangle);
            adapter.drawAll();
        }
    }
```

The RectCommand class is a ConcreteCommand. When it is instantiated, it uses the constructor's arguments to create a rectangle. The command is able to draw this rectangle when the draw method is called.

```
    public class CircleCommand : Command // this is a ConcreteCommand class
    {
        Shape circle;
        Adapter adapter;
        public CircleCommand(Pen color, int x1, int y1, int width, int height, Adapter a)
        {
            circle = new Shape("circle", color, x1, y1, width, height);
            adapter = a;
        }
        public override void draw()
        {
            adapter.addToList(circle);
            adapter.drawAll();
        }
    }
```

The CircleCommand class serves almost the same purpose as the RectCommand class. The chief difference is that the CircleCommand class stores an ellipse instead of a rectangle.

```
    public class Invoker // this is the invoker class
    {
        Adapter adapter;
        Stack<Command> drawnList;
        Stack<Command> undrawnList;

        public Invoker(Adapter a)
        {
            adapter = a;
            drawnList = new Stack<Command>();
            undrawnList = new Stack<Command>();
        }

        public void Add(Command c)
        {
            drawnList.Push(c);
            drawnList.Peek().draw();
        }

        public void Undo()
        {
            if (drawnList.Count > 0)
            {
                undrawnList.Push(drawnList.Pop());
                adapter.remove();
                adapter.clear();
                adapter.drawAll();
            }
        }

        public void Redo()
        {
            if (undrawnList.Count > 0)
            {
                drawnList.Push(undrawnList.Pop());
                adapter.clear();
                drawnList.Peek().draw();
            }
        }
    }
```

The invoker class tracks all commands that have been instantiated in two stacks. The drawnList stack stores commands that have been executed. The undrawnList stack contains all commands that have been undone. The add method adds a command to the drawnList stack and executes it. The undo method moves the command to the undrawnList stack and undoes it. The redo command moves a command from the undrawnList stack back to the drawnList stack and executes it again.

```
    public class Adapter // this is the adapter and receiver class
    {
        Graphics g;
        List<Shape> shapeList;
        public Adapter(System.Windows.Forms.Panel panel)
        {
            g = panel.CreateGraphics();
            shapeList = new List<Shape>();
        }

        public void drawRect(Pen color, int x1, int y1, int width, int height)
        {
            g.Clear(Color.White);
            g.DrawRectangle(Pens.Red, x1, y1, width, height);
        }

        public void drawCircle(Pen color, int x1, int y1, int width, int height)
        {
            g.Clear(Color.White);
            g.DrawEllipse(Pens.Red, x1, y1, width, height);
        }

        public void addToList(string shape, Pen color, int x1, int y1, int width, int height)
        {
            shapeList.Add(new Shape(shape, color, x1, y1, width, height));
        }

        public void addToList(Shape shape)
        {
            shapeList.Add(shape);
        }

        public void clear()
        {
            g.Clear(Color.White);
        }
        public void remove()
        {
            int c = shapeList.Count - 1;
            shapeList.RemoveAt(c);
        }

        public void drawAll()
        {
            for (int i = 0; i < shapeList.Count(); i++)
            {
                Shape s = shapeList[i];
                if (s.getShape() == "rect")
                {
                    g.DrawRectangle(s.getColor(), s.getTopX(), s.getTopY(), s.getLength(),
s.getHeight());
                }
                else if (s.getShape() == "circle")
                {
                    g.DrawEllipse(s.getColor(), s.getTopX(), s.getTopY(), s.getLength(),
s.getHeight());
                }
            }
        }
    }
```

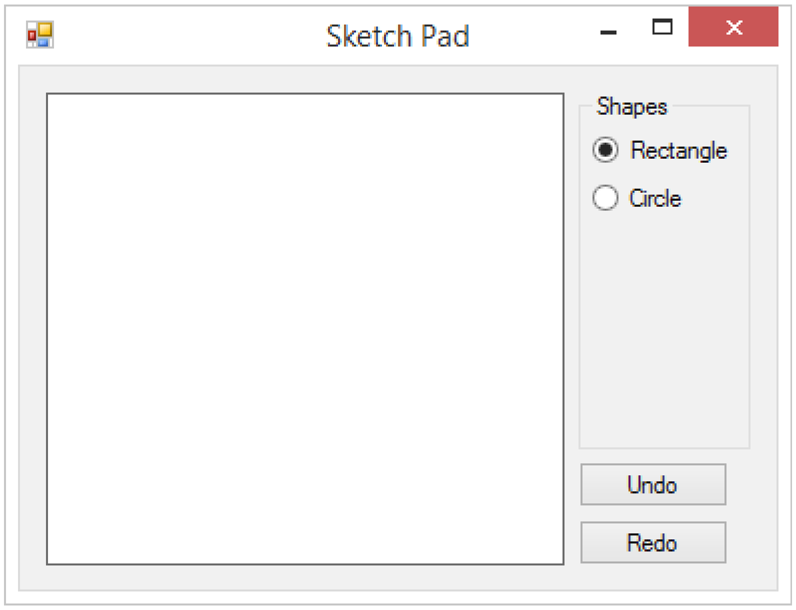
The adapter class is used for both the adapter pattern and the command pattern. It contains methods that make calls to the System.Drawing.Graphics class in ways that work better for the current application. The drawRect, drawCircle, and drawAll methods are part of this implementation. DrawRect draws a rectangle, drawCircle draws an ellipse, and drawAll draws all of the shapes that have been instantiated. These methods are called by the ConcreteCommand classes, allowing this class to also serve as the Receiver class. In order for the commands to communicate properly, several more methods are provided. The clear method clears the Graphics object, which is linked to the panel on the main form. The addToList method is an overloaded method that allows the program to add a new shape to the list either by passing in all the information for the shape or passing in the shape itself. The remove method removes a shape from the list.

```
    public partial class Form1 : Form
    {
        int x1;
        int y1;
        string currentShape;
        Adapter adapter;
        Invoker invoker;
        public Form1()
        {
            InitializeComponent();
            adapter = new Adapter(pnl_canvas);
            currentShape = "rect";
            invoker = new Invoker(adapter);
        }
```

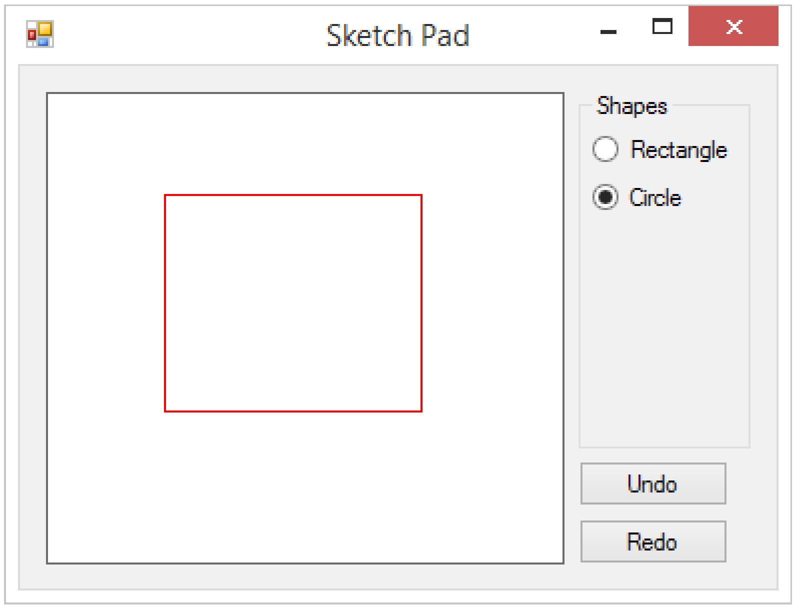
```
private void pnl_canvas_MouseDown(object sender, MouseEventArgs e)
{
    x1 = e.X;
    y1 = e.Y;
}

private void pnl_canvas_MouseUp(object sender, MouseEventArgs e)
{
    if (currentShape == "rect")
    {
        invoker.Add(new RectCommand(Pens.Red, x1, y1, e.X - x1, e.Y - y1, adapter));
    }
    else if (currentShape == "circle")
    {
        invoker.Add(new CircleCommand(Pens.Red, x1, y1, e.X - x1, e.Y - y1, adapter));
    }
}

private void pnl_canvas_MouseMove(object sender, MouseEventArgs e)
{
    if (e.Button != MouseButtons.Left)
    {
        return;
    }
    else
    {
        if (currentShape == "rect")
        {
            adapter.drawRect(System.Drawing.Pens.Red, x1, y1, e.X - x1, e.Y - y1);
        }
        else if (currentShape == "circle")
        {
            adapter.drawCircle(System.Drawing.Pens.Red, x1, y1, e.X - x1, e.Y - y1);
        }
        adapter.drawAll();
    }
}

private void rb_rect_CheckedChanged(object sender, EventArgs e)
{
    if (rb_rect.Checked)
    {
        currentShape = "rect";
    }
}

private void rb_circle_CheckedChanged(object sender, EventArgs e)
{
    if (rb_circle.Checked)
    {
        currentShape = "circle";
    }
}

private void btn_undo_Click(object sender, EventArgs e)
{
    invoker.Undo();
}

private void btn_redo_Click(object sender, EventArgs e)
```

```
        {
            invoker.Redo();
        }
    }
```

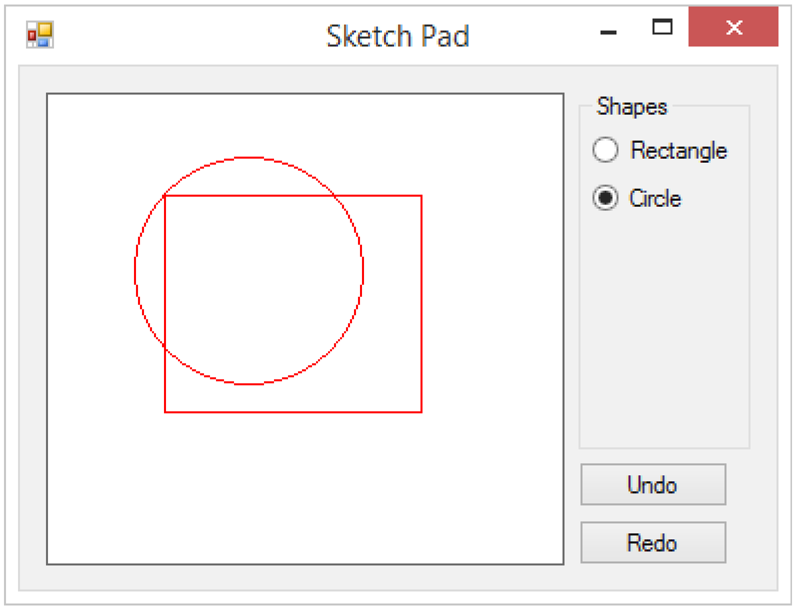
The main form acts as the client for both patterns. It instantiates both an invoker and an adapter. The adapter's graphics object is linked to the panel on the form, and the invoker is linked to the adapter. The default shape is a rectangle, and the shape can be changed via the radio buttons on the side of the program. When the user clicks on the panel, the program takes note of the position at which the click occurred. While the mouse is still held down, the program continuously draws the correct shape between the cursor and the point where the initial click occurred. When the mouse is let up, the final shape is instantiated as a command and drawn. Clicking the undo and redo buttons calls the corresponding events from the invoker, allowing shapes to be undrawn and redrawn. Below are some screenshots of the program in use.



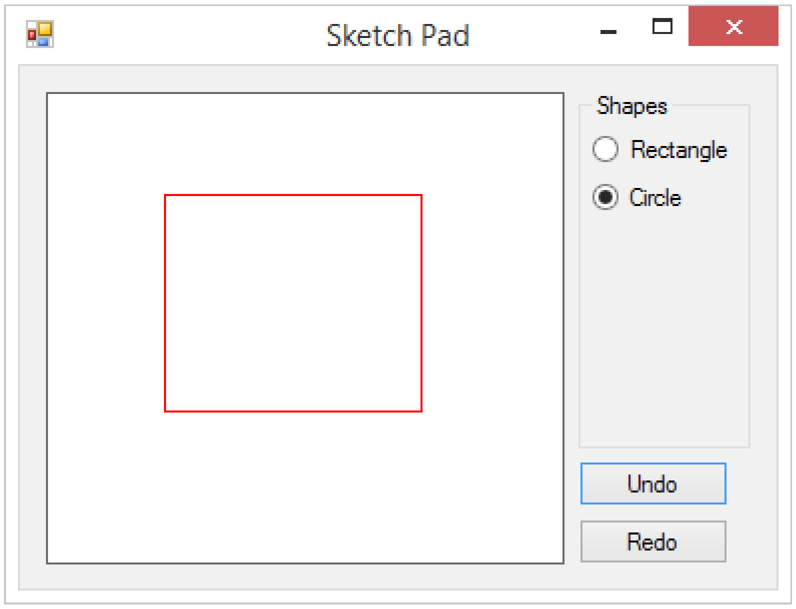
Initial view of the application.



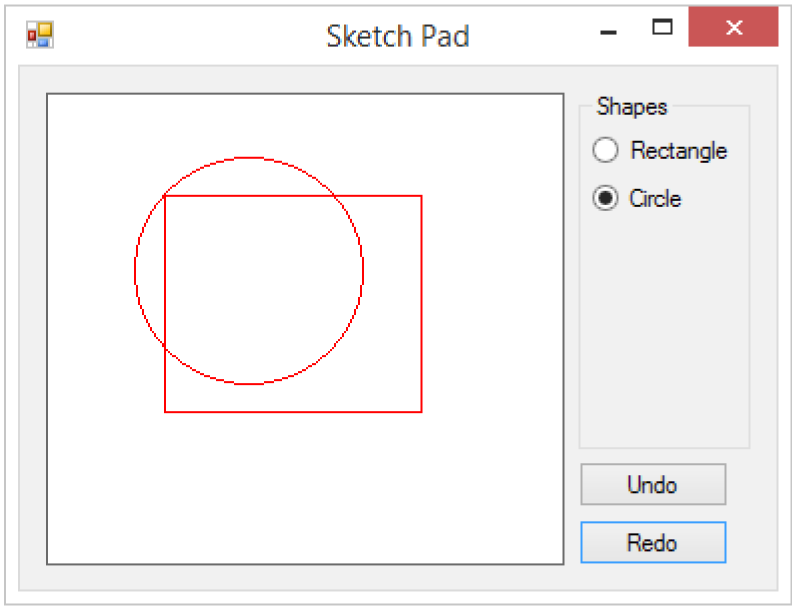
The user can draw rectangles by clicking and dragging on the panel.



Using the radio buttons, the user can elect instead to draw ellipses by clicking and dragging.



The undo button removes the last shape drawn.



The redo button recreates the last shape removed with the undo command.

<u>Observations</u>

Both the command pattern and the adapter pattern are useful. The command pattern allows easy implementation of the undo and redo commands, which almost any application can make use of. The adapter pattern could be useful for repurposing old code. I feel that my submission fits the assignment well. I closely followed the implementation of the command pattern in a way that is beneficial to the user. My adapter pattern implementation was in a context where I would likely have needed it even if I weren't trying to use it on purpose. As such, my application implements both patterns nicely. This assignment taught me about techniques for doing operations I've been trying to figure out for some time. The knowledge it provided will be useful in the future.