Objective Memories

James W. Cooper

Sometimes you can get so focused on one task, you miss everything else that is going on around you. You might be listening to some terrific music, or reading a great novel and forget to feed the dog until he starts slobbering in your lap. You might be trying to get a nice little piece of OO code done and completely overlook the opportunity to use a design pattern to do it better. You might also be trying to use a nice design pattern and in the process completely forget that OO principles still ought to apply.

The Memento Pattern

This happened to me when I was writing a little program that used the Memento pattern pretty effectively. Let me show you what I mean. The purpose of the Memento pattern is to use an external object to save and restore the state of another object. Once place where this can come in handy is in saving the state of drawings you might want to undo later.

Now, objects normally shouldn't expose much of their internal state using public methods, but you would still like to be able to save the entire state of an object because you might need to restore it later. In some cases, you can obtain enough information from the public interfaces (such as the drawing position of graphical objects) to save and restore that data. In other cases, you might need to save the color, shading, angle and connection relationship to other graphical objects, and this information is not readily available.

Now if your object has public methods you can use to get at the values representing its internal state, it is not difficult to save them in some external object. However, making these data public makes the entire system vulnerable to change by external program code, when we usually expect data inside an object to be private and encapsulated from the outside world.

The Memento pattern attempts to solve this problem by having privileged access to the state of the object you want to save. Other objects have only a more restricted access to the object, thus preserving their encapsulation. This pattern defines three roles for objects:

1. The **Originator** is the object whose state we want to save.
2. The **Memento** is another object that saves the state of the Originator.
3. The **Caretaker** manages the timing of the saving of the state, saves the Memento and, if needed, uses the Memento to restore the state of the Originator.

A Privileged Life

Now the term "privileged access" sounds somewhat elitist and a little mystical, especially in an egalitarian language like Java. Languages like C++ have a *friend* modifier that lets one class gain access to private variables of another class, and C# has a kind of friend system too, although it is much more limited in its value.

In Java, the closest we can come is to use the package protected variable visibility. Normally, we declare all of the variables inside a class as private and only get at them using "get" and "set" accessor methods. Alternativley, we could declare them public and anyone can have at them anyway they like. The third alternative is not to declare these variables as either public or private.

In this last case, it turns out that any class in that package can access those variables. This can be kind of dangerous, an that is why we obsessively mark all of our class variables as "private." However, this could be one way to save and restore an object' state without a lot of excess exposure.

Let's now suppose that we have a primitive graphics program that we can use to draw rectangles and move them around on the screen. WE have a toolbar where we can select a button to add a rectangle, and a button to undo the last action. If we click inside a givene rectangle, it draws handles and we can drag it around. You can see the program in action in Figure 1.

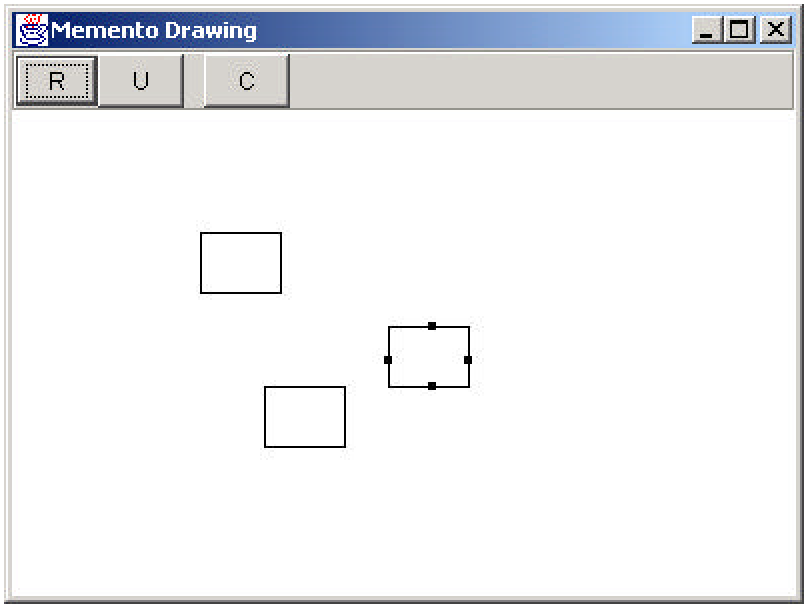

Figure 1 – The rectangle drawing program.

The Memento

Evefry time we drag a rectangle around, we want to save its previous position so we cab undo the latest motion. That is where we use a Memento to save the save of the visual rectangle object. Here is that visual class.

```
public class visRectangle {
    int x, y, w, h;    //package protected
    private Rectangle rect;
    private boolean selected;

    public visRectangle(int xpt, int ypt) {
        x = xpt;   y = ypt;
        w = 40;    h = 30;
        saveAsRect();
    }
    //-------------------------------------------
    public void setSelected(boolean b) {
        selected = b;
    }
    //-------------------------------------------
    private void saveAsRect() {
        rect = new Rectangle(x-w/2, y-h/2, w, h);
    }
    //-------------------------------------------
    public void draw(Graphics g) {
        g.drawRect(x, y, w, h);
        if (selected) {
            g.fillRect(x+w/2, y-2, 4, 4);
            g.fillRect(x-2, y+h/2, 4, 4);
            g.fillRect(x+w/2, y+h-2, 4, 4);
            g.fillRect(x+w-2, y+h/2, 4, 4);
        }
    }
    //-------------------------------------------
    public boolean contains(int x, int y) {
        return rect.contains(x, y);
    }
    //-------------------------------------------
    public void move(int xpt, int ypt) {
        x = xpt; y = ypt;
        saveAsRect();
    }
}
```

Note that the variables x, y, w, and h are not declared as private and are therefore package protected. Now, you can save the state of that rectangle, and a reference to the rectangle itself using a Memento class:

```
public class Memento {
    visRectangle rect;
    /**saved fields- remember internal fields
    of the specified visual rectangle
    */

    private int x, y, w, h;
    public Memento(visRectangle r) {
        rect = r;
        x = rect.x;  y = rect.y;
        w = rect.w;  h = rect.h;
    }
    //-------------------------------------------
    public void restore() {
        //restore the internal state of
        //the specified rectangle
        rect.x = x;  rect.y = y;
        rect.h = h;  rect.w = w;
    }
}
```

The constructor saves a reference to the visRectangle object and then saves the current coordinates. The restore method returns these saved values to the saved visRectangle object.

Taking Care of the Objects

There are two more classes we use in this program. The Mediator simply interprets button clicks and tells the Caretaker to keep track of what has transpired. We've seen lots of Mediators before so we won't work trough that again here. The Caretaker class keeps track of the undo list and creates Memento objects as needed.

If we add a visRectangle to the drawing list, we simply call the following Caretake method to add it to the undo list as well:

```
    public void addElement(Object obj) {
        undoList.addElement (obj);
    }
```

Likewise, if the rectangle gets moved, we call this same method with a Memento object. Here is the Mediator method that adds these Mementos:

```
    public void rememberPosition() {
        if (rectSelected) {
            Memento m =
                  new Memento(selectedRectangle);
            caretaker.addElement(m);
            repaint();
        }
    }
```

It is pretty obvious that we want to add Mementos when we want to save a rectangle motion, but what do we add when we just add a new rectangle to the drawing list? Well in this program, we just add an Integer representing the number of that drawing in the drawing list:

```
Integer count =
      new Integer(drawings.size());
//Save previous drawing list size
caretaker.addElement(count);
```

How to Do Undo?

So now we have an undo list vector that contains a mixture of Integer objects and Memento objects. How do we do undo? We could call an undo method in the Caretaker class and have it tease this apart:

```
      public void undo() {
        if (undoList.size() > 0) {
            //get last element in undo list
            Object obj = undoList.lastElement();
            undoList.removeElement(obj);   //and remove it
            //remove Integer or Memento
            if (obj instanceof Integer)
                remove((Integer)obj);
            else
                remove((Memento)obj);

        }
    }
```

where our two polymorphic remove methods either restore the Memento contents or just shorten the drawing list:

```
//remove the drawing from the list
    private void remove(Integer obj) {
        Object drawObj = drawings.lastElement();
        drawings.removeElement(drawObj);
    }
    //------------------------------------
    //restore the Memento contents
    private void remove(Memento obj) {
        Memento m = (Memento)obj;
        m.restore();     //and restore the old position
    }
```

And this will all work pretty well. Undo undoes the right thing each time. But then, I thought, “Oh horror!” What kind of object oriented programming is that? You should *never* have to check the type of an object and then call a particular method. The method should be polymorphic and each one should work only on the correct type. How do we do this, when we don’t know the type of object in a Vecdtor, since they are all returned as type Object?

Well, we need to do a bit of rewriting. How about making everything a Memento, and making it a more general class?

More and Better Mementos

What we do is redefine the Memento to be any class that has a restore method. In other words, Memento becomes just an interface:

```
public interface Memento {
    public void restore() ;
}
```

Then, we redefine the class that remembers rectangle positions to be called RectangleMemento:

```
public class RectangleMemento implements Memento {
    visRectangle rect;
    private int x, y, w, h;
    public RectangleMemento(visRectangle r) {
        rect = r;
        x = rect.x;  y = rect.y;
        w = rect.w;  h = rect.h;
    }
    //-------------------------------------------
    public void restore() {
        //restore the internal state of
        //the specified rectangle
        rect.x = x;  rect.y = y;
        rect.h = h;  rect.w = w;
    }
}
```

But what do we do about that Integer object that just says remove this one? There are probably a number of similar approaches you could think of. One that occurred to me that required only very small code changes, is to create an IntegerMemento class which is derived from Integer:

```
public class IntegerMemento implements Memento {
    private Vector drawings;
    public IntegerMemento(Vector drws) {
        drawings = drws;
    }
    //-------------------
    public void restore (){
        drawings.removeElementAt (drawings.size () -1);
    }
}
```

All this class really does is save the drawing list. It's *restore* method doesn't even need to know the drawing number, because we will always be remove the last one in the list. That is exactly what the above restore method does.

Now, how does our Caretaker's undo method work? We simplify it greatly because all of the objects in the undo list are Memento objects. The undo is now

```
      public void undo() {
        if (undoList.size() > 0) {
            //get last element in undo list
            Memento mem = (Memento)undoList.lastElement();
            undoList.removeElement(mem);   //and remove it
            mem.restore ();

        }
```

We just call restore on whatever object is on the list. This is much more acceptable from an OO basis, and is also much more extensible. If we add circles, fills and colors, we can make all of them Memento objects as well and call their restore methods.

So you see, I did really improve and simplify this after looking in horror at my lapse. Now I have to go feed the dog.