THE MEMENTO PATTERN

Suppose you would like to save the internal state of an object so you can restore it later. Ideally, it should be possible to save and restore this state without making the object itself take care of this task, and without violating encapsulation. This is the purpose of the Memento pattern.

Motivation

Objects frequently expose only some 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 could obtain enough information from the public interfaces (such as the drawing position of graphical objects) to save and restore that data. In other cases, the color, shading, angle and connection relationship to other graphical objects need to be saved and this information is not readily available. This sort of information saving and restoration is common in systems that need to support Undo commands.

If all of the information describing an object is available in public variables, it is not that difficult to save them in some external store. 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.

Implementation

Saving the state of an object without making all of its variables publicly available is tricky and can be done with varying degrees of success in various languages. *Design Patterns* suggests using the C++ *friend* construction to achieve this access, and the *Smalltalk Companion* notes that it is not directly possible in Smalltalk. In Java, this privileged access is possible using a little known and infrequently used protection mode. Variables within a Java class can be declared as

1. Private
2. Protected
3. Public, or
4. (private protected)

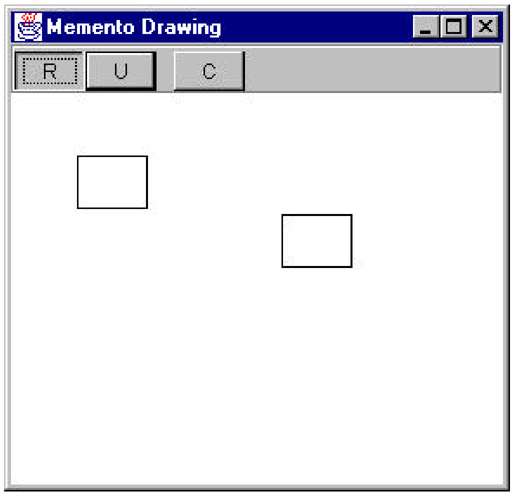
Variables with no declaration are treated as private protected. Other classes can access public variables, and derived classes can access protected variables. However, another class in the same module can access protected or private-protected variables. It is this last feature of Java that we can use to build Memento objects. For example, suppose you have classes A and B declared in the same module:

```
public class A {
int x, y:
public Square() {}
x = 5;                          //initialize x
}
//------------------------
class B {
public B() {
  A a = new A();                //create instance of A
  System.out.println (a.x);     //has access to variables in A
 }
}
```

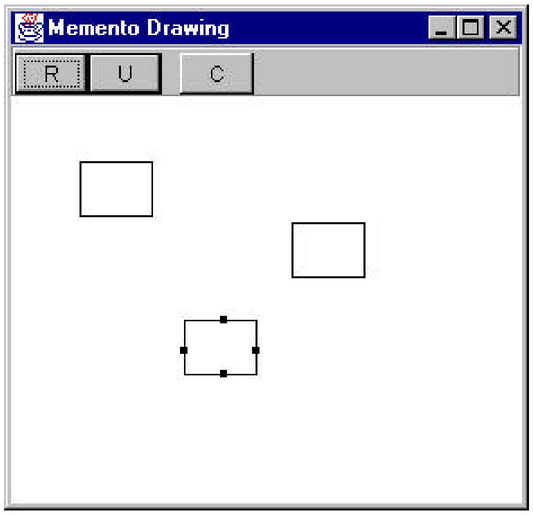
Class A contains a private-protected variable *x*. In class B in the same module, we create an instance of A, which automatically initializes *x* to 5. Class B has direct access to the variable *x* in class A and can print it out without compilation or execution error. It is exactly this feature that we will use to create a Memento.

Sample Code

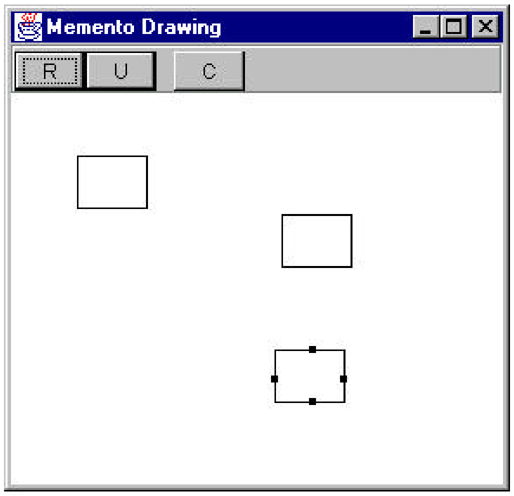
Let’s consider a simple prototype of a graphics drawing program that creates rectangles, and allows you to select them and move them around by dragging them with the mouse. This program has a toolbar containing three buttons: Rectangle, Undo and Clear:



The Rectangle button is a JToggleButton which stays selected until you click the mouse to draw a new rectangle. Once you have drawn the rectangle, you can click in any rectangle to select it;



and once it is selected, you can drag that rectangle to a new position using the mouse:



The Undo button can undo a succession of operations. Specifically, it can undo moving a rectangle and it can undo the creation of each rectangle.

There are 5 actions we need to respond to in this program:

1. Rectangle button click
2. Undo button click
3. Clear button click
4. Mouse click
5. Mouse drag.

The three buttons can be constructed as Command objects and the mouse click and drag can be treated as commands as well. This suggests an opportunity to use the Mediator pattern, and that is, in fact, the way this program is constructed.

Moreover, our Mediator is an ideal place to manage the Undo action list; it can keep a list of the last *n* operations so that they can be undone. Thus, the Mediator also functions as the Caretaker object we described above. In fact, since there could be any number of actions to save and undo in such a program, a Mediator is virtually required so that there is a single place where these commands can be stored for undoing later.

In this program we save and undo only two actions: creating new rectangles and changing the position of rectangles. Let’s start with our visRectangle class which actually draws each instance of the rectangles:

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

  public visRectangle(int xpt, int ypt)   {
     x = xpt;   y = ypt;   //save location
     w = 40;    h = 30;    //use default size
     saveAsRect();
  }
  //--------------------------------------------
  public void setSelected(boolean b)   {
     selected = b;
  }
  //--------------------------------------------
  private void saveAsRect()   {
  //convert to rectangle so we can use the contains method
     rect = new Rectangle(x-w/2, y-h/2, w, h);
  }
 //--------------------------------------------
  public void draw(Graphics g)   {
     g.drawRect(x, y, w, h);
     if (selected)    {  //draw “handles”
        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();
  }
}
```

Drawing the rectangle is pretty straightforward. Now, let’s look at our simple Memento class, which is contained in the same file, visRectangle.java, and thus has access to the position and size variables:

```
class Memento
{
   visRectangle rect;
   //saved fields- remember internal fields
   //of the specified visual rectangle
   int x, y, w, h;
   public Memento(visRectangle r)    {
      rect = r;     //Save copy of instance
      x = rect.x;  y = rect.y;   //save position
      w = rect.w;  h = rect.h;   //and size
   }
   //-------------------------------------------
   public void restore()    {
      //restore the internal state of
      //the specified rectangle
      rect.x = x;  rect.y = y;   //restore position
      rect.h = h;  rect.w = w;   //restore size
   }
}
```

When we create an instance of the Memento class, we pass it the visRectangle instance we want to save. It copies the size and position parameters and saves a copy of the instance of the visRectangle itself. Later, when we want to restore these parameters, the Memento knows which instance it has to restore them to and can do it directly, as we see in the *restore()* method.

The rest of the activity takes place in the Mediator class, where we save the previous state of the list of drawings as an Integer on the undo list:

```
public void createRect(int x, int y)
{
   unpick();   //make sure no rectangle is selected
   if(startRect)  //if rect button is depressed
   {
   Integer count = new Integer(drawings.size());
   undoList.addElement(count);   //Save previous list size
   visRectangle v = new visRectangle(x, y);
   drawings.addElement(v);       //add new element to list
   startRect = false;            //done with this rectangle
   rect.setSelected(false);      //unclick button
   canvas.repaint();
   }
   else
      pickRect(x, y); //if not pressed look for rect to select
   }
```

and save the previous position of a rectangle before moving it in a Memento:

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

Our undo method simply decides whether to reduce the drawing list by one or to invoke the *restore* method of a Memento:

```
public void undo()
{
   if(undoList.size()>0)
   {
      //get last element in undo list
      Object obj = undoList.lastElement();
      undoList.removeElement(obj);   //and remove it
      //if this is an Integer,
      //the last action was a new rectangle
      if (obj instanceof Integer)
      {
         //remove last created rectangle
         Object drawObj = drawings.lastElement();
         drawings.removeElement(drawObj);
      }
      //if this is a Memento, the last action was a move
      if(obj instanceof Memento)
      {
         //get the Memento
         Memento m = (Memento)obj;
         m.restore();     //and restore the old position
      }
     repaint();
   }
 }
```

Consequences of the Memento

The Memento provides a way to preserve the state of an object while preserving encapsulation, in languages where this is possible. Thus, data that only the Originator class should have access to effectively remains private. It also preserves the simplicity of the Originator class by delegating the saving and restoring of information to the Memento class.

On the other hand, the amount of information that a Memento has to save might be quite large, thus taking up fair amounts of storage. This further has an effect on the Caretaker class (here the Mediator) which may have to design strategies to limit the number of objects for which it saves state. In our simple example, we impose no such limits. In cases where objects change in a predictable manner, each Memento may be able to get by with saving only incremental changes of an object's state.

Other Kinds of Mementos

While supporting undo/redo operations in graphical interfaces is one significant use of the Memento pattern, you will also see Mementos used in database transactions. Here they save the state of data in a transaction where it is necessary to restore the data if the transaction fails or is incomplete.