Calling the Mediator

James W. Cooper

Not long ago, I spent some time trying out the Java Foundation (Swing) classes and learning how they worked. I made roll-over buttons, a toolbar, a tree list, a tabbed panel, and used the JList control for the first time. Of course, as soon as I finished getting the code all working, the program looked like a candidate for a project we were working on. Then we found some code from another project we could use for part of this project, and I integrated the two during a coast-to-coast flight.

Considering how little room there is in airline seats these days, I had to operate with my elbows in mid-air and the laptop tipped so I could see the screen and type at the same time. You can well imagine what an elegant program this was at the end of my trip. It looked OK on the screen and it worked, but what a tangle of classes and objects. It was pretty much unreadable and unmaintainable. What to do? Well, of course, I'd like to make the program more structured and object-oriented than it was, and I'd like to make sure that each of the objects was of modest size.

To simplify a bit for the pedagogical purposes of this article, the program consisted of a toolbar, a text entry panel and a result list display. The data that filled the result list came from a server, connected using Java RMI. Now the problem I faced was not unusual in Java programming: how do you keep all the user interface elements updated without having everything in a single module.

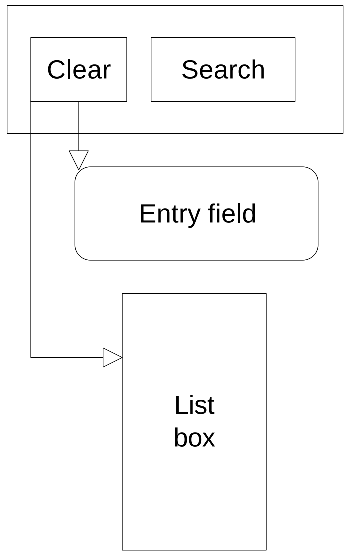


Figure 1: How the Clear button works in our simple search interface.

For example, suppose I click on a button marked "Clear." It should clear the data entry field and the results list. That means the button needs to know how to access the text entry field and the results list box. Now, suppose that I click on the other button marked "Search." That button needs to know how to read the query from the text entry field, clear the results list, access the data server, get the results and load the results list. This is illustrated in Figure 2.

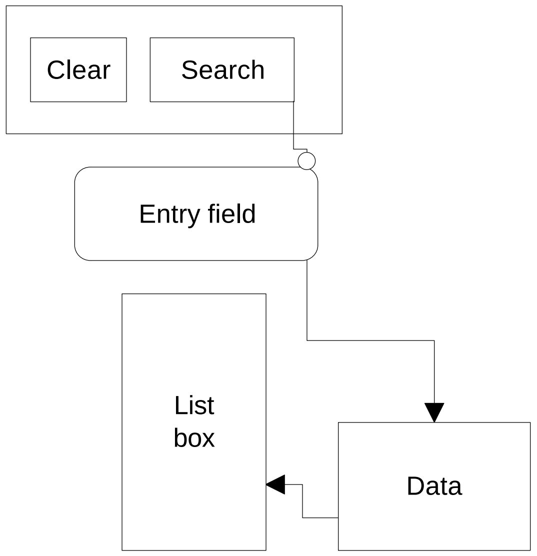


Figure 2: How the Search button works in this simple search user interface.

Whew! What a tangle!

We first need to decide how to break this program into useful objects. Then we'll figure out how the objects can best communicate. It seems reasonable to define the toolbar, entry field, list box and data server as objects.

Then we'll extend every button to be a Command object. Remember from our last column that a Command object is any class that has an Execute method. Typically we make Buttons and menuItems into Command objects so our actionListener can just call their Execute methods.

```
abstract interface Command {
   abstract public void Execute();
}
```

This seems like a good idea, but it has the potential implication that each button class should have access to all the UI elements, such as could occur if the buttons were inner classes of the main program.

```
class clearButton extends Jbutton
      implements Command   {
  public void Execute() {
      entryField.setText(“”); //clear entry field
      list.clear();           //clear list box
  }
}
```

In this code snippet, the text field **entryField** and the list box **list** must be objects that can be accessed by the clearButton class. This can happen if the clearButton class is an inner class and the list and entryField are in the outer class. For one or two buttons, this is probably OK, but if we have 8-10 buttons, this approach can become pretty clumsy and confusing. It is just substituting one tangle for another.

Of course, we could still make the classes separate if we had a simple way to pass references to various objects between these classes. However, if every object knows about the methods of every other object, we still have written a pretty tangled program. A better solution is to introduce a Mediator class.

Using a Mediator

We can untangle these inter-class references by introducing another class to mediate the interactions between these GUI objects and the data server. This object follows a common design pattern called a Mediator. In essence, you send all the actions to the Mediator and let it decide which objects need to be notified of that action.

The Mediator is set up to know about all the interacting classes and knows whose methods it needs to call in response to any action. Each of these classes registers itself with the mediator as part of initialization:

```
dmodel = new DataModel();            //create the data model
med = new Mediator(dmodel);          //tell mediator where data is
pToolBar pTool = new pToolBar(med); //create toolbar

JTextField ef = new JTextField();   //text field

pList list = new pList(med);        //create list

med.registerTextField(ef);          //tell mediator where
med.registerList(list);             //text field, list
med.registerToolbar(pTool);         //and toolbar are
```

Now, instead of having the main program catch the actionPerformed events, the toolbar itself became an ActionListener for the two (or more) buttons which are now command objects. Then within the toolbar class, we simply call each individual tool button's Execute method.

```
public class pToolBar extends JToolBar
        implements ActionListener
{
   public pToolBar(Mediator med)
   {
   //pass Mediator and ActionListener
   //to each button
   add(new ClearButton(this, med));    //Clear
   addSeparator();
   add(new SearchButton(this, med));   //Search
   }
   //-----------------------------------------
   public void actionPerformed(ActionEvent e)
   {
      Command comd = (Command)e.getSource();
      comd.Execute();
   }
}
```

A typical button is shown below. It calls specific methods of the Mediator class.

```
public class ClearButton extends JToolButton implements Command
```

```
{
   Mediator med;
   public ClearButton(ActionListener act, Mediator md)
   {
      setIcon(new ImageIcon("images/new.gif", "new"));
      setToolTipText("Clear for new search");
      addActionListener(act);
      setRolloverIcon( new ImageIcon("images/rnew.gif", "new"));
      med = md;

   }
   public void Execute()
   {
   med.clear();        //tell datamodel to clear all data
   }
}
```

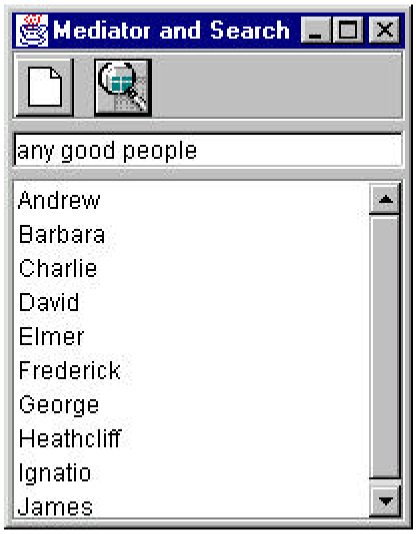
Note that the Execute method calls the **clear** method in the Mediator. In the same way, each button calls only Mediator methods, and only the Mediator needs to know which other classes it needs to communicate with.

In this simple example, the Mediator itself has two methods: **clear** and **search,** which correspond to the buttons. The mediator sends out the commands to the various objects:

```
public class Mediator
{
   private pToolBar ptool;
   private JTextField entry;
   private pList list;
   private DataModel dmodel;

   public Mediator(DataModel dm)    {
   dmodel = dm;
   }
   //-----------------------------
   public void clear()    {
    entry.setText("");  //clear the text field
    list.clear();       //and the list box
   }
   //-----------------------------
   public void search()    {
      String searchText = entry.getText();   //get the search text
      String s[] = dmodel.search(searchText);//do the search
      list.clear();                          //clear the list
      list.addToList(s);                     //display search results
   }
   //-----------------------------
   public void registerToolbar(pToolBar pt) {
      ptool = pt;
   }
   //-----------------------------
   public void registerTextField(JTextField tf)    {
      entry = tf;
   }
   //-----------------------------
   public void registerList(pList lst)   {
        list = lst;
    }
}
```

We show the final program in the display below.



The source code for this simple program, made up of 12 small classes, is available for download from this magazine's web site. The main program is called protoSearch.java.

Is there One More Class?

There actually is one more class our Mediator needs to be cognizant of. Suppose one of our command buttons initiates a process which will take a significant amount of time. We might want to change the cursor to an hourglass until that process completes. But whom do we tell to change a cursor? We haven't included the Frame UI as an object our Mediator makes use of. But we clearly need it in order to change the cursor. So as part of the Mediator's constructor or as a set method, we need to pass that frame or applet window to the Mediator. Then it can process all the commands and change the cursor whenever it wishes.

Other Uses of the Mediator

In addition to simply keeping classes from tangling when you issue commands, you can also use a mediator to keep track of whether buttons should be enabled or disabled. One of the most common uses for this is the case where certain menu or button actions can only occur when a list item is selected. Here you just have the ListSelectionListener's method **valueChanged()** call a Mediator method when a list item is selected, and then have the mediator enable the correct buttons.

Summary

In conclusion, the Mediator can be one of your most powerful tools for simplification of "object tangling." Almost any program with two or more user interface commands can benefit from using a Mediator to keep the objects separate. I’d like to emphasize that a Mediator has no special list of approved or necessary methods: you can write a Mediator to contain whatever methods you find useful. The only important aspect of the Mediator is that it is one of the few objects that knows about the methods of the other objects. It also makes changing this behavior much simpler, since all of the interactions are encapsulated in a single class, and when you change the interface to one of your objects, only the Mediator is affected. It makes programming an awful lot easier all around.

James W. Cooper is at work on his 13th book, Java Design Patterns, for Addison-Wesley.