You Can't Go Home Again

James W. Cooper

Recently I was invited to give a talk for the Computer Science department at Oberlin College, a school I attended long before there was a computer science major. In fact, I attended long before they even had ones and zeros. I think we used Monroe calculators and either slide rules or abacuses. So I pulled out the text of my September, 2000 column and begin adapting it into a talk. To my surprise, I didn't agree with some of my conclusions exactly any more. And further, the whippersnapper students at my alma mater pointed out some further improvements.

So lets assume we are writing a program with several radio buttons to select different groups from a longer list, as we see in Figure 1. When you click on the Female radio button, only the women's names are shown, and when you click on the Male button, the program shows the men's names.

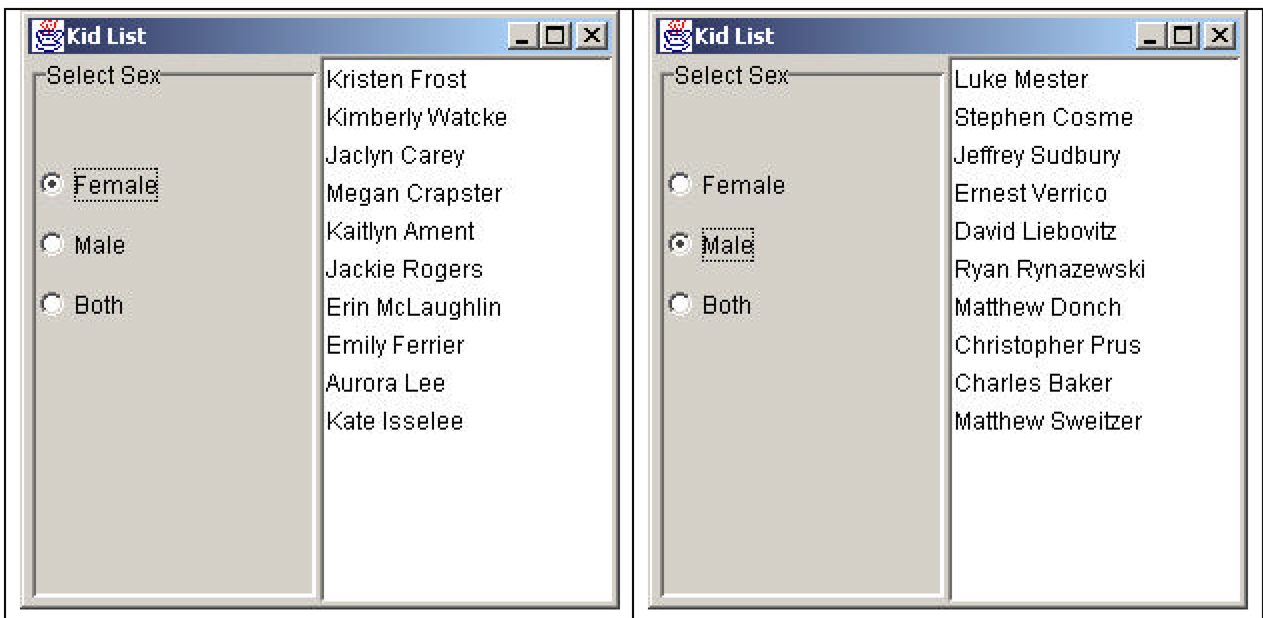

Figure 1 – A simple Java program to select groups from a longer list.

This is quite an easy program to write. However, it is simple enough that we can use it to illustrate a number of ways to improve a program using Design Patterns. This is what I started out to do in the previous column, and what I am going to take another stab at today.

The really simple program we won't even discuss does everything in the main class with some arrays or vectors to hold the kid's names and sexes. The next simplest approach creates objects for each kid and has methods like *isFemale* to help us determine whether the kids we want to display are male or female. I took this list of names from the result of a swim meet on the web, so we'll call the class for each kid Swimmer. When we read the data in from a file, it also provides information on the swimmer's age, club and time, but we won't use that in this display example. The basic Swimmer class is merely

```
public class Swimmer implements Serializable {
```

```
    String name;
    int age;
    String club;
    float time;
    boolean female;
//---------------
//represents one swimmer read fron a line in a datafile
    public Swimmer(String dataline) {
        StringTokenizer st = new StringTokenizer(dataline, ",");
        name = st.nextToken();
        age = new Integer(st.nextToken().trim()).intValue();
        club = st.nextToken().trim();
        time = new Float(st.nextToken().trim()).floatValue();
        String sx = st.nextToken().trim().toUpperCase();
        female = sx.equals("F");
    }
    public boolean isFemale() {
        return female;
    }
    public int getAge() {
        return age;
    }
    public float getTime() {
        return time;
    }
    public String getName() {
        return name;
    }
    public String getClub() {
        return club;
    }
}
```

The next thing we do is to decide which swimmers to pick from a list. We could do this in the main UI class as well, but it is better to do it in a collection class called Swimmers, that has a getList method with a Boolean for male or female:

```
//get a vector of swimmers who are (female)
      public Vector getList(boolean female) {
            Vector v = new Vector();
            for (int i=0; i< kids.size(); i++ ) {
                  Swimmer swm = (Swimmer)kids.elementAt(i);
                  if(swm.isFemale() == female) {
                        v.addElement(swm);
                  }
            }
            return v;
      }
```

We also provide a polymorphic getList method with no argument, which returns the entire list.

Now we get to the first major point. To display these kids' names, we need to add an event listener to each of the three radio buttons and then display the right kids. We might consider doing this as follows:

```
      public void actionPerformed(ActionEvent evt) {
```

```
        //listen for button clicks
        //and do the right thing
        Object obj = evt.getSource();
        if (obj == female) {
              loadFemales();
        }
        if (obj == male) {
              loadMales();
        }
        if (obj == both) {
              loadBoth();
        }
  }
  //-----------
  private void loadFemales() {
        //display female swimmers
        Vector v = swimmers.getList(true);
        loadList(v);
  }
  //-----------
  private void loadMales() {
        //display male swimmers
        Vector v = swimmers.getList(false);
        loadList(v);
  }
```

A Commanding Lead

However well this approach works for the small, simple case, it is not very scalable. If there are 5 or 10 different buttons, it is not very readable to extend the actionPerformed method to test for each of them and call some appropriate routine. Instead, we would bet better off moving all these processing decisions out of the class containing the user interface.

One way to do this is using the Command pattern. When we use this pattern, we create a Command interface:

```
//the Command interface
public interface Command {
      public void Execute();
}
```

This interface simply indicates that classes that implement it must have a method called Execute. The point of this is if we extend our 3 radio buttons to special classes with a Command interface, we can move the execution of the commands out of the main Jframe class and into each button's class. For example, we could create a FemaleButton class that is derived from JradioButton and has this method:

```
      //Radio button to select female swimmers
      public class FemaleButton extends JRadioButton
                  implements Command {
      protected Swimmers swimmers;
      protected JawtList kidList;
      //-----
      public FemaleButton(String title, Swimmers sw, JawtList klist) {
            super(title);
```

```
        kidList = klist;
        swimmers = sw;
    }
    //-----
    public void Execute() {
        Vector v = swimmers.getList(true);
        loadList(v);
    }
    //----------
    protected void loadList(Vector v) {
        kidList.clear();
        for (int i = 0; i < v.size(); i++) {
            Swimmer swm = (Swimmer) v.elementAt(i);
            kidList.add(swm.getName());
        }
    }
}
```

Now you see that in order to load a list with female swimmers, you need only call this button's Execute method. We can do the same for the male button, reusing some of the same code:

```
    public class MaleButton extends FemaleButton {
    public MaleButton(String title, Swimmers sw, JawtList jlist) {
        super(title, sw, jlist);
    }
    //-----
    public void Execute() {
        Vector v = swimmers.getList(false);
        loadList(v);
    }
}
```

Now, let's see what we have wrought. All of our buttons are Command buttons and our actionPerfomed method now reduces to the following simple method.

```
    public void actionPerformed(ActionEvent evt) {
        Command comd = (Command) evt.getSource();
        comd.Execute();
    }
```

As you can see this is much simpler and completely scalable.

Mediating the Labor

But there is still more we ought to do. We have now written three radio button classes that know how to load a list box. In our haste to get the details out of the main class, we have required that each button know about the list box. If we decided to change to a different kind of display, we'd have to change 3 different classes.

It is better if we create a mediator class that mediates between the buttons and the list, so they don't know about each other. The Mediator Design Pattern does this. We create a mediator class that loads the list when a button is clicked. Then all the buttons only have to know about the Mediator, and what list gets loaded is known only to the Mediator:

```
public class Mediator {
    private JawtList kidList;
    //the list box is passed in the constructor
```

```
        public Mediator(JawtList klist) {
              kidList = klist;
        }
        //---------------
        //load the list box
        public void loadList(Vector v) {
              kidList.clear();
              for(int i=0; i< v.size(); i++ ) {
                    Swimmer sw = (Swimmer) v.elementAt(i);
                    kidList.add(sw.getName());
              }
        }
}
```

Then our radio button classes become:

```
public class FemaleButton extends JRadioButton
              implements Command {
        protected Swimmers swimmers;
        protected JawtList kidList;
        protected Mediator med;
        //------
        public FemaleButton(String title, Swimmers sw, Mediator md) {
              super(title);
              swimmers = sw;
              med = md;
        }
        //------
        //use the Mediator to load the list
        public void Execute() {
              Vector v = swimmers.getList(true);
              med.loadList(v);
        }
}
```

Now, you see that we use the Mediator to keep the knowledge of the buttons and the list more separate. But we can do better than this, and this is where I depart from my earlier faith.

We can simply tell the Mediator that a button has been clicked, and let it decide what to do about it. In this version of the Mediator, we have 3 button_clicked methods that the 3 buttons call. The Mediator itself decides what to do about these clicks:

```
public class Mediator {
        private JawtList kidList;
        private Swimmers swimmers;
        public Mediator(JawtList klist, Swimmers sw) {
              kidList = klist;
              swimmers = sw;
        }
        //---------------
        public void fButtonClick() {
              Vector v = swimmers.getList(true);
              loadList(v);
        }
        //---------------
        public void mButtonClick() {
              Vector v = swimmers.getList(false);
              loadList(v);
```

```
	}
	//--------------
	public void bButtonClick() {
		Vector v = swimmers.getList();
		loadList(v);
	}
	//--------------
	private void loadList(Vector v) {
		kidList.clear();
		Iterator iter = v.iterator();
		while(iter.hasNext()){
			Swimmer sw = (Swimmer) iter.next();
			kidList.add(sw.getName());
		}
	}
}
```

Better Buttons from a Template

Now here is where I got some help from the students. One pointed out that a base class really shouldn't have a name that specifies one of several possible functions, like male, female and both. Instead, the base class should be a general class like SexButton and we should derive all the others from it. So, we'll create an abstract base class SexButton. At the same time, we'll move that ActionListener code into the base class, so we don't have to add an action listener to each button separately:

```
//abstract radio button class
public abstract class SexButton extends JRadioButton
		implements Command {
	protected Mediator med;		//keep the mediator here

	public SexButton(String title, Mediator md,
			ActionListener al) {
		super(title);
		med = md;
		addActionListener(al);
	}
	//abstract Execute method
	public abstract void Execute();
}
```

Now, note that this class must be extended to actually be useful, since we haven't made the Execute method concrete. This base abstract button class is just a template for the concrete classes we derive from it, and is in fact a simple example of the Template design pattern.

Got Any More Patterns?

So far we have adapted this really simple program of less than 100 lines of Java to use the Command, Mediator and Template patterns. Are there any others lurking below the surface? Why, funny you should ask. There are 4 more.

Many columns ago, we discussed simplifying the use of the JList class by building an interface that emulates the simpler java.awt.list class. Our JawtList class is in fact an Adapter pattern that makes using the Jlist quite easy:

```java
//convert List awt methods to Swing methods

public class JawtList extends JScrollPane
implements ListSelectionListener, awtList {
    private JList listWindow;
    private JListData listContents;

    public JawtList(int rows) {
        listContents = new JListData();
        listWindow = new JList(listContents);
        listWindow.setPrototypeCellValue("Abcdefg Hijkmnop");
        getViewport().add(listWindow);
    }
//-----
    public void add(String s) {
        listContents.addElement(s);
    }
//-----
    public void remove(String s) {
        listContents.removeElement(s);
    }
//-----
    public void clear() {
      listContents.clear();
    }
//-----
    public String[] getSelectedItems() {
        Object[] obj = listWindow.getSelectedValues();
        String[] s = new String[obj.length];
        for (int i =0; i < obj.length; i++)
            s[i] = obj[i].toString();
        return s;
    }
}
```

This class make use of the JListData to get data to and from the list

```java
public class JListData extends AbstractListModel {
      private Vector data;

      public JListData() {
            data = new Vector();
      }
      //------
      public int getSize() {
            return data.size();
      }
      //------
      public Object getElementAt(int index) {
            return data.elementAt(index);
      }
      //------
      public void addElement(String s) {
            data.addElement(s);
            fireIntervalAdded(this, data.size() - 1, data.size());
      }
      //-----
```

```
	public void removeElement(String s) {
		data.removeElement(s);
		fireIntervalRemoved(this, 0, data.size());
	}
	//------
	public void clear() {
		int size = data.size();
		data = new Vector();
		fireIntervalRemoved(this, 0, size);
	}
}
```

And in fact, the JawtList observes changes in the JListData class. As such, it is an example of the Observer pattern.

Further, the InputFile code that reads in the data from our data file wraps the somewhat complicated java.io.* functions in a simpler interface, and can be regarded as a primitive example of the Façade pattern. Here is how we use our simple InputFile class:

```
public class Swimmers {
	private Vector kids;
	//-----
	public Swimmers(String file) {
		kids = new Vector();
		//read in the data a line at a time
		InputFile inf = new InputFile(file);
		String line = inf.readLine();
		while(line != null) {
			Swimmer swm = new Swimmer(line);
			kids.addElement(swm);
			line = inf.readLine();
		}
		inf.close();
	}
```

And finally, when we generate our cute little borders around the panels in our window we write the following code:

```
kidList = new JawtList(20);
kidList.setBorder(
	BorderFactory.createBevelBorder(
	BevelBorder.LOWERED));
```

This is the simplest example of a Factory pattern.

Summary

So, we've written a very simple program over several times, ending up using the Command, Mediator, Template, Adapter, Observer, Façade and Factory patterns before breakfast! Bet you couldn't do that without going to college! Yes, you can go home again, but don't reread your old columns.