DON'T OBJECT TO OBJECTS

James W. Cooper

Last month I was writing a simple Visual Basic program with 3 radio buttons and a list box. The point of the program was to display different things in the list box depending on which button was selected, and to print out that list when a Print button was selected. This is about as easy to do in VB as it is in Java, but I started thinking about how much better you could do this in Java, and realized that a really good example of that program exemplified a whole bunch of OO programming concepts.

Let's start by defining a simple problem. We have a list of kids, both boys and girls, and we'd like the option of displaying the girls, the boys, or both. A simple GUI for this problem is shown in Figure 1.

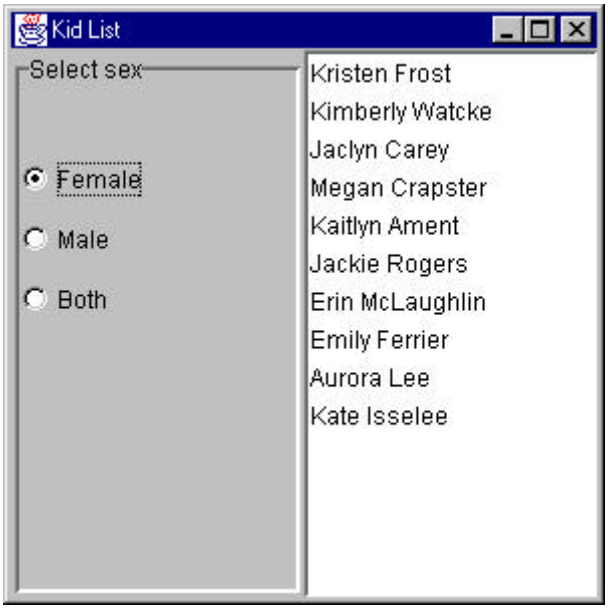

Figure 1 – The Swing window showing the results of clickingon the Female radio button.

A Simple Implementation

Let's write this program in the simplest way possible, the way it might occur to you when you start out. We'll use the Swing classes, because they look a lot nicer on the screen, but we'll take some shortcuts. We have previously described a simple wrapper that makes the JList easier to use, in a class I called JawtList, because it has similar properties to the original awt List object. We'll use that for our list box. Then, we'll put the three JRadioButtons in a BoxLayout on the left side as shown above, and surround it with a TitledBorder. This is done in Listing 1, and represents a fairly straightforward way of constructing a Swing window.

```
public class ShowList extends JxFrame implements ActionListener {
    private JRadioButton female, male, both;
    private JawtList kidList;
    private Swimmer sw;
    private Vector swimmers;

    public ShowList() {
```

```
        super("Kid List");
        JPanel jp = new JPanel();   // create interior panel
        jp.setLayout(new GridLayout(1,2));
        getContentPane().add(jp);
        JPanel lp = new JPanel();
        lp.setLayout(new BoxLayout(lp, BoxLayout.Y_AXIS));
        jp.add(lp);

        //create titled border around radio button panel
        Border bd = BorderFactory.createBevelBorder (BevelBorder.LOWERED );
        TitledBorder tl = BorderFactory.createTitledBorder (bd, "Select sex");
        lp.setBorder(tl);

        //put bevel border around list
        kidList = new JawtList(20);
        kidList.setBorder (BorderFactory.createBevelBorder (BevelBorder.LOWERED));
        jp.add(kidList);

        //add in the radio buttons
        female = new JRadioButton("Female");
        male = new JRadioButton("Male");
        both = new JRadioButton("Both");

        //keep them all together
        ButtonGroup grp = new ButtonGroup();
        grp.add(female);
        grp.add(male);
        grp.add(both);

        //make sure they all receive clicks
        female.addActionListener (this);
        male.addActionListener (this);
        both.addActionListener (this);

        //space the buttons out
        lp.add(Box.createVerticalStrut (30));
        lp.add(female);
        lp.add(Box.createVerticalStrut (5));
        lp.add(male);
        lp.add(Box.createVerticalStrut (5));
        lp.add(both);
```

Listing 1 – Setting up the Swing

Note that the base class is derived from JxFrame, a class I described previously which includes automatic setting of the look and feel, and setting the window to exit when the Close box is clicked. This class implements the ActionListener interface, and each of the radio buttons has been told that this frame is the listener for their actions. I also had some code form a previous example around which would read in a list of kids from a file along with their sexes along with other information we don't use here. We bury this in a Swimmer class which parses each line of that file.

Now, the simplest way to write this program is to carry out the loading of the list in the actionPerfomed method:

```
public void actionPerformed(ActionEvent evt) {
        Object obj = evt.getSource ();
        if(obj == female)
            loadFemales();
        if(obj == male)
```

```
            loadMales();
        if(obj == both)
            loadBoth();

    }
```

Then, we have three code loading methods for females, males and both. They look like this:

```
private void loadFemales() {
        kidList.clear();
        Enumeration enum = swimmers.elements();
        while(enum.hasMoreElements ()) {
            Swimmer sw = (Swimmer)enum.nextElement ();
            if (sw.isFemale ()) {
                kidList.add (sw.getName ());
            }
        }
    }
```

It has often been noted that for every problem, there is a solution that is neat, simple, and wrong. This is such a case. Even though the code works fine in this example, it is about as far from being object oriented as you can imagine. Whenever you see a set of if statements deciding which thing to do next, as we have in the actionPerformed method above, you should immediately suspect that you have not written the best possible OO program.

Taking Command

Let's see what we can do to make this a little less tacky. Suppose we derive 3 button subclasses from JRadioButton and have each implement the Command interface. Recall that the Command interface just says that each class will have an Execute method:

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

So, we'll make a FemaleButton, a MaleButton and a BothButton, each of which have an Execute method that loads the list with the right data. Here's the female version:

```
public class FemaleButton extends JRadioButton implements Command {
    Vector swimmers;
    JawtList kidList;

    public FemaleButton(String title, Vector sw, JawtList klist) {
        super(title);
        swimmers = sw;
        kidList = klist;
    }
    public void Execute() {
        Enumeration enum = swimmers.elements();
        kidList.clear();
        while(enum.hasMoreElements ()) {
            Swimmer sw = (Swimmer)enum.nextElement ();
            if (sw.isFemale ()) {
                kidList.add (sw.getName ());
            }
        }
    }
}
```

Note that we pass in the instance of the kidList list box and of the Vector of kids and add the right ones to the list box when Execute is called. This approach greatly simplifies the actionPerfomed method to just:

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

Now there is no testing of buttons, since each one knows the right thing to do in its Execute method.

Making Better Choice

But we can still do better than this. We have more or less the same code in the 3 button classes, each of which loads the list with something based on a decision that the button makes. We really ought to separate the interface from the data better than that. Buttons themselves should not be making decisions. They ought only to implement the visual logic needed to display the results of decisions made elsewhere.

So, we should consider replacing that vector of kids names with classes that make the decisions. Let's start with a Kids class which holds the data and loads the list:

```
public class Kids {
    protected Vector swimmers;

    //set up the vector
    public Kids(){
        swimmers = new Vector();
    }
    //add a kid to the list
    public void add(String line) {
        Swimmer sw = new Swimmer(line);
        swimmers.add(sw);
    }
    //return the vector
    public Vector getKidList() {
        return swimmers;
    }
    //get an enumeration of the kids
    public Enumeration getKids() {
        return swimmers.elements();
    }
}
```

This class creates a Vector of Swimmers and returns an enumeration of them as needed. The enumeration is returns is of the whole list of kids. However, we can derive classes from Kids that return enumerations of males or females by simply extending the getKids method. So we create a FemaleKids class just like the one above, except that the getKids methods returns only girls:

```
public class FemaleKids extends Kids {
    //set up vector
    public FemaleKids(Kids kds) {
        swimmers = kds.getKidList();
    }
    //return female sonly
    public Enumeration getKids() {
```

```
        Vector kds = new Vector();
        Enumeration enum = swimmers.elements();
        while(enum.hasMoreElements ()) {
            Swimmer sw = (Swimmer)enum.nextElement ();
            if(sw.isFemale ())
                kds.add (sw);
        }
        return kds.elements();
    }
}
```

This is the whole class: the remaining methods are in the base class. Similarly, we create a MaleKids class which differs only in the line

```
      if(! sw.isFemale ())
```

The buttons themselves then instantiate an instance of the correct class, with each using only that class. This gets away from having to have a getAll, a getFemales and a getMales method, when they are really all the same. Here's the MaleButton class as we recast it to use the MaleKids class.

```
public class MaleButton extends JRadioButton implements Command {
    MaleKids kds;
    JawtList kidList;

    //constructor
    public MaleButton(String title, Kids sw, JawtList klist) {
        super(title);
        kds = new MaleKids(sw);
        kidList = klist;
    }
    //The getKids method is the same in all three classes
    public void Execute() {
        Enumeration enum = kds.getKids();
        kidList.clear();
        while(enum.hasMoreElements ()) {
            Swimmer sw = (Swimmer)enum.nextElement ();
            kidList.add (sw.getName ());
        }
    }
}
```

Now, not only have we gotten rid of that awkward set of if statements in the actionPerformed routine, we've replaced three methods in one awkward class with a single method in 3 simpler classes. This is a much simpler and even more object-oriented approach. Having three button classes like this, each of which instantiates a different instance of the Kids class is an example of the Factory Method pattern, and having the 2 classes derived from the base Kids class is an example of the Template pattern.

Mediating the Final Difference

But, we're still not done. Our 3 button classes all have to know about the kidList list box and add the names to it in the Execute method. This means that each button object has to know the details of the kidList object, and this is also poor design. It makes the program hard to change and maintain. Suppose we wanted to replace that list with a table or a tree list. We'd have to change 3 classes. Clearly that is a terrible idea, especially if the number of buttons grows.

But if the Execute method has to be in the button class, and the list has to be elsewhere, how do we resolve this? We resolve it be creating another class called a Mediator. This Mediator is the only class that knows about the details of the kidList. And all other classes only have to know about the Mediator. Here's the entire class:

```
//this mediator is used to get the enumeration
//and load the list
public class Mediator {
    JawtList kidList;

    //save the list in the constructor
  public Mediator(JawtList klist) {
      kidList = klist;
  }
  //load the list from the enumeration
  public void loadList(Enumeration enum) {
      kidList.clear();
       while(enum.hasMoreElements ()) {
           Swimmer sw = (Swimmer)enum.nextElement ();
           kidList.add (sw.getName ());
       }
  }
}
```

The way we use this, is that we create an instance of the kidList class and then create a Mediator:

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

         loadSwimmers();                //read in file
         med = new Mediator(kidList);   //create Mediator
```

Then, we pass an instance of the Mediator to each button when we create it

```
        female = new FemaleButton("Female", kds, med);
         male = new MaleButton("Male", kds, med);
         both = new BothButton("Both", kds, med);
```

and each Execute method just tells the Mediator what to do.

```
public class FemaleButton extends JRadioButton implements Command {
    Kids kds;
    Mediator med;

    public FemaleButton(String title, Kids sw, Mediator md) {
        super(title);
        kds = sw;
        med = md;
    }
    public void Execute() {
        Enumeration enum = kds.getFemales ();
        med.loadList (enum);
    }
}
```

And, this brings us to the end of our design exercise. We've taken a pretty poor first program example and made it more object oriented and more extensible, and made it simpler as well.

The UML Diagram

Figure 2 shows the UML diagram for this program.

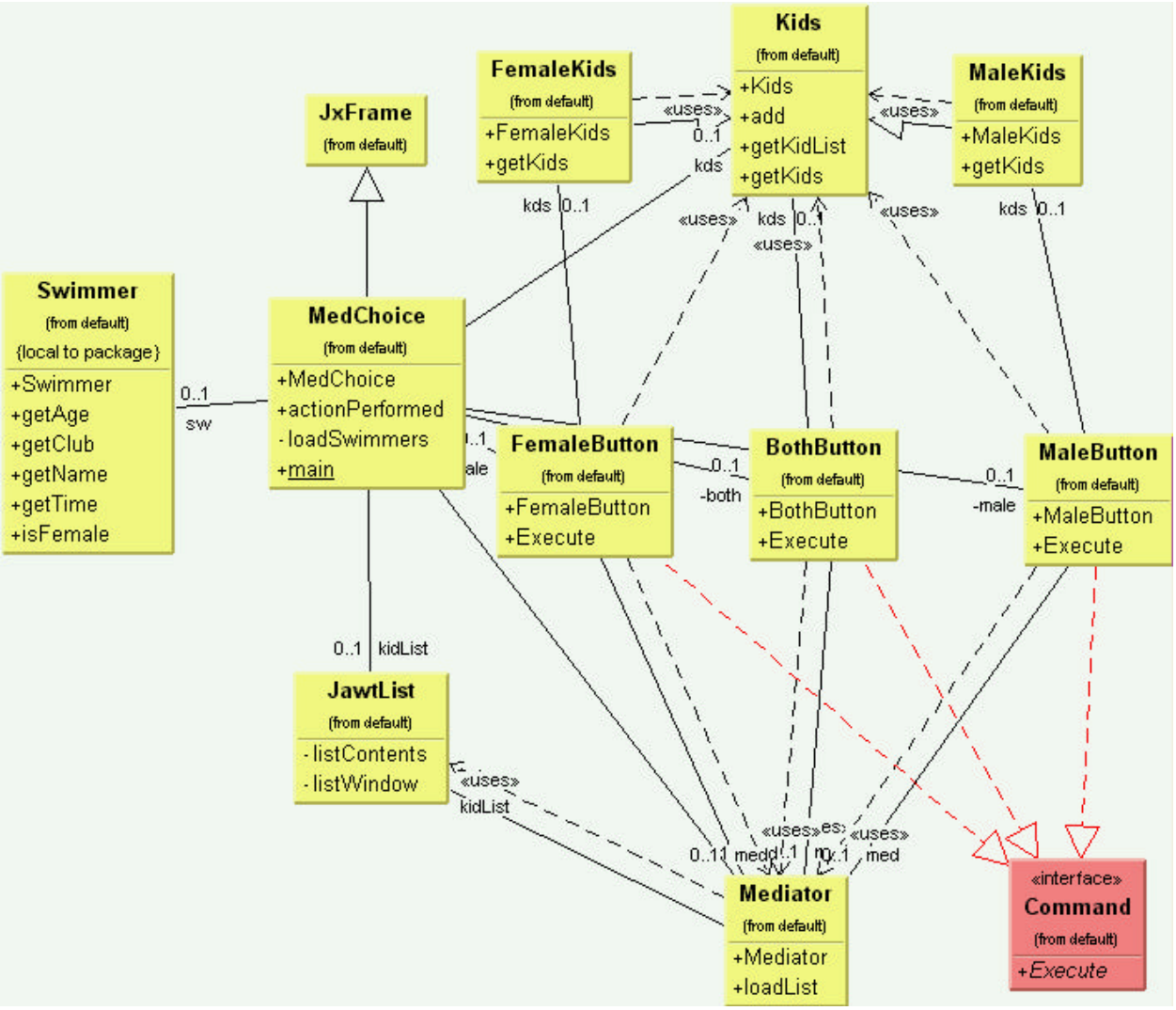

Figure 2 – The UML diagram for the final mediated version of the 3 button choice and display program.

While this diagram seems a bit complex at first, it is easy to see that there are 3 classes called Kids, FemaleKids and MaleKids and corresponding button classes FemaleButton, BothButton and MaleButton, all of which implement the Command interface. It is this parallel set of classes that gives us the Factory Method pattern and makes the final program so simple to read and change.

Concluding Patterns

In this brief article, we've taken a simple idea and turned it into a program that uses a whole bunch of design patterns. We've seen examples of the Command, the Template, the Factory Method and the Mediator. In addition, our JawtList is an example of the Adapter pattern and all Swing lists are themselves examples of the Observer pattern. So you see that Design patterns do

not complicate your life, but in fact make the programs simpler and easier to maintain. I put all of the 4 versions in the example code, so you can see just how they differ and how they become simpler.