The SOAP Factory

James W. Cooper

Sometimes you happily adopt something new because it looks shiny, new and seductive, but only later discover that under the covers are some real problems. For example, most of us have gone through that downward spiral on a new car. It's shiny, looks nice, smells great and drives well. But, wait, where's the cup holder? And can I fit a bale of peat moss in that quirky trunk?

New technologies are no different. They look like a panacea at first, and seem to work like a dream out of the box, but after while the bugs and quirks start to crawl out from under the disks and manuals, and you probably have as much chance of getting your money back there as you would from a car dealer.

So, for some time, I've been trying to get the point across that Design Patterns are a great way to write better Java code. They represent a set of good ideas and best practices from our collective knowledge of OO programming. I use them all the time: in every non-trivial program I write, and not only in Java but in Visual Basic and C++ as well.

The new, shiny toy on the shelf more recently has been the whole idea of Web Services, whatever they are. And while the whole venue of such services continues to evolve, the underlying SOAP technology is quite solid and sophisticated and worthy of adoption by anyone who wants to exchange information across any kind of network.

Recall that SOAP is an object-access protocol that is most commonly used to convert objects to a data stream and send it across the wire. Converting objects to a data stream is called *serialization* and SOAP converts object representations into XML for transmission, and then reserializes them into objects at the other end. SOAP itself doesn't require any specific transmission method, although HTTP is very commonly used. You can use SMTP as well as other less well-known methods.

One place where I've used SOAP very effectively is when I have a simple Java applet or application on my client, but want to use some server resources to look something up. If the result of that lookup is best represented as an object rather than a string, you can well imagine that it might be nice if the system sent you objects instead of a bunch of text you have to take apart and reassemble as objects. Of course, this is what SOAP does, and even is a simplistic description of how it does it. The process of serializing and deserializing is just the conversion of objects in XML and back to objects again.

One problem we might face, though, is how we can incorporate some of the more sophisticated object technology, like Design Patterns into our SOAP server system. The reason this might be a problem is that SOAP requires you to register and serialize specific classes rather than handling objects that may contain other objects, or make use of other objects.

Let's consider what really goes on in SOAP serialization. The underlying assumption is that the serializer program can figure out what is inside the object and pull it out to represent it in XML and then put it back again. The most common serializer is the

BeanSerializer program, which simply uses the getXxx and setXxx accessor methods to obtain the contents of the values inside the class instance. In other words, serialization takes place by using introspection to obtain all the contents of an object by calling its *get* methods, and deserialization consists of creating a new class instance and calling its *set* methods to put those values back. So, in one sense, SOAP is like the science-fictional (stefnal) matter transmitter that breaks down our body into atoms and transmits signals representing each atom and then builds new ones to those specs and reconstructs the body at the receiving end. Is that the same "you" at this end as it was on that end? No, but it has the identical contents.

So, since objects are destroyed and recreated, it doesn't much matter what kind they are as long as they have the same interface for getting the contents and putting them back. Let's take an example of SOAP using a simple Design Pattern to see how this works.

The Name Factory

We'll return now to those thrilling days of yesteryear, when we first talked about the simple factory pattern. Suppose we have a simple interface that allows the user to enter a name in the order first-name<space>last-name, or in the order last-name<comma>first-name. We decide somewhere inside the system which has been entered and take the string apart accordingly. We show a simple user interface for this program in Figure 1

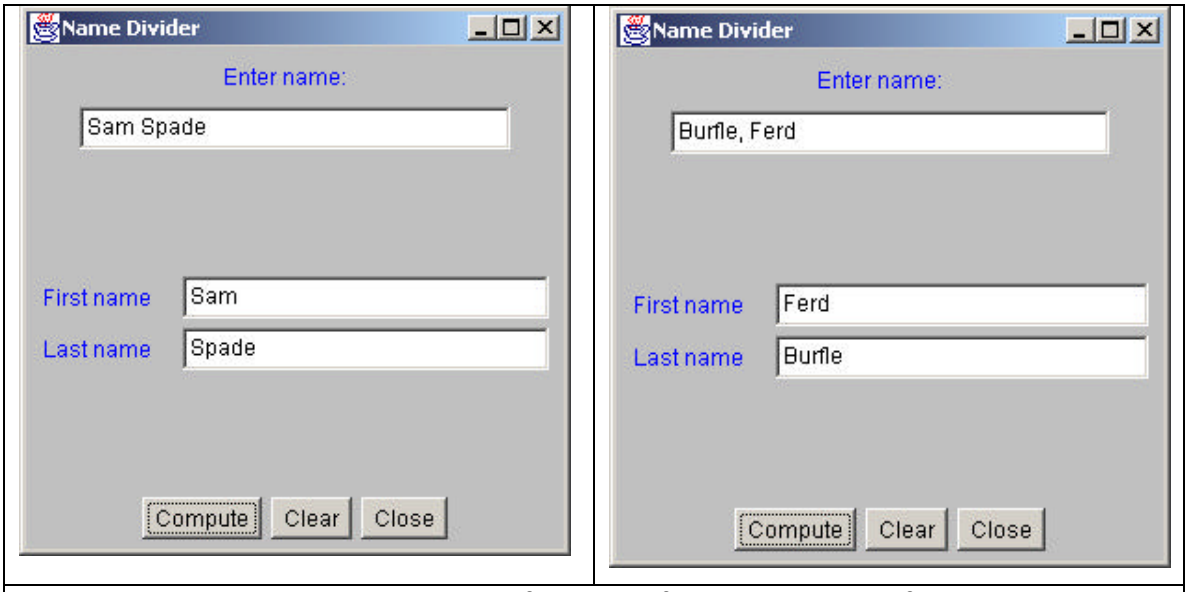

Figure 1- A name divider program: (a) first name first; (b) last name first.

Inside, what we want is a Namer object that somehow computes the division of first and last names and provides a couple of *get* methods to obtain these values. It might look like this:

```
public class Namer {
    //base class extended by two child classes
    protected String last;     //split name
    protected String first;    //stored here

    public String getFirst() {
```

```
        return first;           //return first nmae
    }
    public String getLast() {
        return last;            //return last name
    }
}
```

Of course, the critical difference we have to detect is whether the entered name contains a comma or not. If it does, we assume last name first. So we can imagine creating two child classes that handle the two kinds of name division. Here is the first name first class:

```
public class FirstFirst extends Namer {
    //extracts first name from last name
    //when separated by a space
    public FirstFirst(String s) {
        int i = s.lastIndexOf(" "); //find sep space
        if (i>0) {
            first = s.substring(0, i).trim();
            last =s.substring(i+1).trim();
        } else {
            first = ""; // if no space
            last = s;   // put all in last name
        }
    }
}
```

and here is the last name first class:

```
public class LastFirst extends Namer {
// extracts last name from first name
// when separated by a comma
    public LastFirst(String s) {
        int i = s.indexOf(","); //find comma
        if (i > 0) {
            last = s.substring(0, i).trim();
            first = s.substring(i + 1).trim();
        } else {
            last = s;       //if no comma,
            first = "";     //put all in last name
        }
    }
}
```

Note that in both cases, we carry out the name division in the constructor, and then can use the getFirst and getLast methods from the base class to obtain the results.

Then we decide which class to use by simply checking for the presence of a comma in a driving program:

```
public class NamerFactory {
    //Factory decides which class to return based on
    //presence of a comma
    public Namer getNamer(String entry) {
        //comma determines name order
        int i = entry.indexOf(",");
        if (i > 0)
            return new LastFirst(entry);
        else
            return new FirstFirst(entry);
    }
}
```

This driver program is, in fact, a Factory. It returns one of several classes of the same inheritance tree, or the same interface, and you never need to know which one, because they all have the same interface. The client program just executes the getFirst and getLast methods of the Namer class without ever knowing whether this is a FirstFirst or a LastFirst child class.

Using Our SOAP Factory

Now, what would we have to do to put this factory on a server and serve it up using SOAP. We'd have to have classes that could be serialized. In other words, classes that have *get* and *set* methods for all of the values they contain. Here is our breakthrough moment. We almost have this already. If we just modify the base Namer class to contain *set* methods along with the existing *get* methods, we have serializable classes. We see the revised Namer class below:

```
//Serializable base Namer class
public class Namer {
    protected String last;     //split name
    protected String first;    //stored here

    public String getFirst() {
        return first;          //return first nmae
    }
    public void setFirst(String frst) {
       first = frst;           //set first name
    }
    public String getLast() {
        return last;           //return last name
    }
    public void setLast(String lst) {
        last = lst;            //set last name
    }
}
```

Starting up the Factory

Now what do we have to do to turn what once was a simple stand-alone program into a client server system using SOAP? First we have to create a jar file containing the NamerFactory, Namer, FirstFirst and LastFirst classes, and put it in our classpath, so the SOAP server system can find it. Then, we have to register these classes with the Soap administrator system. In my system, I have installed the Tomcat Apache server, and the soap.jar and xerces.jar files from Apache, and put them in my classpath as well.

Then, after starting Tomcat, I can point my web browser to

http://localhost:8080/soap/admin

and register the NameFactory class as shown in Figure 2.

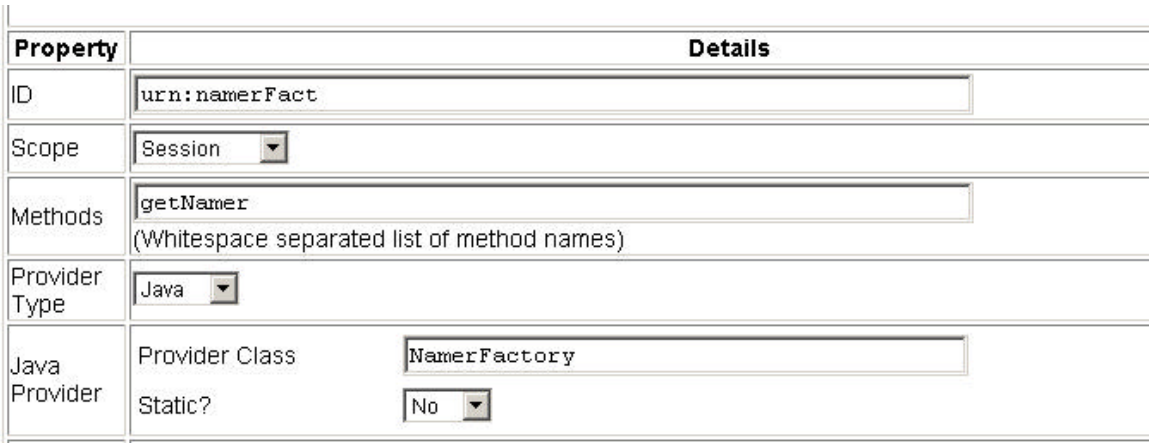

Figure 2 – Defining the NamerFactory to SOAP

In the top line we enter "urn:" and follow it with some name we make up that defines the factory class. Then we decide whether the scope of a SOAP connection is Request, Session or Application and pick that in the dropdown. In the third line we enter the names of the method(s) that class will support, and in the fifth line, we enter the name of the class itself. If this is a class in a package, you enter the whole package path here. So, in other words, the ID in line 1 is a shortcut for the package name string in line 5.

Then, we register the methods we are going to use lower on the same page as we see in Figure 3.

Figure 3 – Registering the mappings of the methods.

There is only one method being mapped, and we give it some name prefixed with "urn:."

The Local Part is the name of the argument to that method from the factory's one method:

```
public Namer getNamer(String entry) {
```

The type the method returns is a class of type Namer. Note that we have just made Namer and its descendants serializable, as long as they can be reconstructed using the *set* methods we added above. The serializer we use in both directions is:

```
org.apache.soap.encoding.soapenc.BeanSerializer
```

Then, we click on the Deploy button in the administrator user interface, and the system writes these data to a configuration file.

That completes the SOAP server part. The Tomcat server will look up these classes based on these "urn:" names and make the right connection.

Calling the Factory for SOAP

Now, we need only write the client. We have to make sure all the SOAP classes are imported and available, so I always include all of these declarations in the client code:

```
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
```

The system we are writing here is a Java application, but as we have noted before, a Java applet will work as well. Our client application constructs the user interface and makes the SOAP mapping connection:

```
public Person() {
        super("Name Divider");
        setGUI();               //build the UI
        setMapping(8080);      //could be a variable
    }
```

The mapping code is shown below:

```
private void setMapping(int portNum) {
        url = null;
        int port = portNum;
        //create the URI to connect to
        encodingStyleURI =  Constants.NS_URI_SOAP_ENC;
        try {
            url = new URL("http://localhost:"+ portNum +
                          "/soap/servlet/rpcrouter");
        } catch (MalformedURLException e) {
            System.out.println("Bad url");
        }
        //map the NameFactory object
        smr = new SOAPMappingRegistry();
        BeanSerializer beanSer = new BeanSerializer();
        //the method we are calling
        //represented as a urn
        //and the name of the argument
        //to the method
        smr.mapTypes(Constants.NS_URI_SOAP_ENC,
                     new QName("urn:get-Namer", "entry"),
                     Namer.class, beanSer, beanSer);
    }
```

Note that the mapTypes method connects the client mapping with that we declared in the administrator interface.

To make the SOAP call when the Computer button is clicked in our UI, we create a target object URI and pass it the value of the entry field as shown below:

```
private Namer getSoapName(String entry) {
        Namer nmr = null;
        // Build the call.
        Call call = new Call();

        call.setSOAPMappingRegistry(smr);
        //this is the object we are calling
        call.setTargetObjectURI("urn:namerFact");
        //and this is the method we will call
```

```
        call.setMethodName("getNamer");
        call.setEncodingStyleURI(encodingStyleURI);

        //here we set the parameters to the call
        Vector params = new Vector();
        params.addElement (
            new Parameter("name", String.class, entry, null));
        call.setParams(params);

        Response resp = null;

        try {
            resp = call.invoke(url, "");
        } catch (SOAPException e) {
            System.err.println(e.getMessage());
            return null;
        }

        // Check the response.
        if (!resp.generatedFault()) {
            Parameter ret = resp.getReturnValue();
            //cast returned object to correct type
            nmr = (Namer)ret.getValue();

        } else {
            Fault fault = resp.getFault();
            System.err.println("Generated fault: ");
            System.out.println ("  Fault Code   = " + fault.getFaultCode());
            System.out.println ("  Fault String = " + fault.getFaultString());
        }
        return nmr;        //return Namer class
    }
```

This returns an instance of the Namer class. We don't know which one. Whichever one it is, we can call its getFirst and getLast methods and load the UI fields with the results:

```
private void computeName() {
        Namer namer = getSoapName(entryField.getText());
        txFirstName.setText(namer.getFirst ());
        txLastName.setText(namer.getLast ());
    }
```

And that's the whole program. It's interesting to note that the client doesn't know about the NameFactory class or the two derived Namer classes. It only knows about the base Namer class, and about how to connect to SOAP to get Namer objects back.

Summary

Now we see how powerful SOAP can be. As long as the objects have the specified interface and are serializable, we can write some pretty elaborate programs using Design Patterns like the simple Factory pattern we used here.