Building GUIs with SwiXml

by Joshua Marinacci
02/21/2006

Layout has always been a major weak point of the Swing toolkit. At the API level, Swing supports virtually any layout you can imagine, but in practice, the default layout managers leave much to be desired. Over the years, enterprising developers have created a variety of utility classes to make the job easier, but recent work on the problem suggests that the best way to lay out a Swing GUI is to remove layout from the Java code altogether, either through a visual builder like Matisse in NetBeans 5.0 or by using a language other than Java to specify the GUI. One common solution is to use XML that completely separates GUI construction from program logic. This article will introduce an open source library, SwiXml, and show you how to construct your Swing user interfaces faster and more easily than ever before.

The Layout Problem

Building on its AWT foundations, Swing has very powerful support for creating complicated UI designs. Using a standard layout manager like `GridBagLayout` or `SpringLayout`, a developer can build an attractive interface that automatically accounts for different platform component sizes, fonts, and languages. You can even create your own custom layout manager for tricky cases like maintaining a fixed ratio. With the right layout manager, you can do almost anything.

In practice however, as the humorous animation "Totally Gridbag" shows, Swing layout can be a pain to use. `GridBagLayout` code is typically verbose and annoying to read after the fact. `SpringLayout` was intended for visual GUI builders, so coding it by hand is extremely difficult. Over time, layout code often becomes unwieldy, resulting in ugly UIs that break on different platforms. This is the opposite of progress. There has to be a better way.

In recent years, many developers have decided that it is a good practice to separate the GUI layout from event handling and program logic. This keeps the program logic cleaner and allows the layout to be maintained separately, often by a different developer. Some have gone a step further to specify layout in a language other than Java. These days, the natural choice for any new language is of course XML, and a quick internet search for "java xml gui layout" will return many commercial and open source layout products. One of the best is SwiXml, a tiny but powerful library that transforms simple XML descriptions into attractive user interfaces.

What Is SwiXml?

SwiXml is a small Java library created by Wolf Paulus in 2003 to produce Swing GUIs from a small XML language. SwiXml doesn't introduce any new layout managers or component classes. Instead, it directly operates on the Swing classes using reflection. This means that the XML syntax is easy to learn for anyone used to the Swing API. It also has the side benefit of keeping the library very small (under 60k, plus the JDOM .jar), which makes application deployment a pleasant experience.

A Simple Example

Before I get deeper into the syntax and mechanics of SwiXml, I'd like to show you a short example to get a feel for how it works.

This is a very simple XML file that will create a frame with one text field and a button:

HelloWorld.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<frame size="300,100">
    <label constraints="BorderLayout.CENTER" text="Hello World"/>
```

```
    <button id="quit" constraints="BorderLayout.SOUTH" text="Quit"/>
</frame>
```

There is one `frame` element with two sub-elements for the two components. The elements have many more optional attributes, but these are the basics necessary to show something on screen.

Next, you need a static `main` method for your application.

HelloWorld.java

```
import java.awt.event.*;
import javax.swing.*;
import org.swixml.*;

public class HelloWorld {
    public JButton quit;
    public HelloWorld() throws Exception {

        new SwingEngine(this).render("HelloWorld.xml")
                .setVisible(true);

        quit.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                System.exit(0);
            }
        });
    }

    public static void main(String [] args) throws Exception {
        new HelloWorld();
    }
}
```

This class looks just like a normal Swing application. The only unusual thing is the following line:

```
new SwingEngine(this).render("HelloWorld.xml").setVisible(true);
```

This is what calls the SwiXml API to parse the *HelloWorld.xml* file and makes the resulting frame visible on screen.

To compile and run the application code, you need to have the SwiXml and JDOM .jars in your classpath:

```
javac -classpath swixml.jar:jdom.jar HelloWorld.java
java -classpath swixml.jar:jdom.jar:. HelloWorld
```

Running the program above will give you an application that looks like Figure 1:

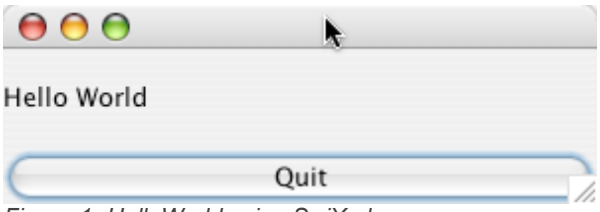

Figure 1. HelloWorld using SwiXml

There is a lot of magic involved here, so I will go through it step by step. The `SwingEngine` class is a parser that reads and processes the XML file. It takes each element in the XML and converts it into a Swing component, placing them directly in the hierarchy. This is all pretty straightforward. The `quit` button is the tricky part. The button is declared as a field in the `HelloWorld` class, but the object is never directly created or assigned, and yet the `addActionListener` method doesn't throw a `NullPointerException`. How can this happen?

If you look at the XML file closely, you will see the `button` element has an attribute called `id` with a value of `quit`. At run time, the `SwingEngine` will create the `button` object during the parsing phase and then assign it to the `quit` field using reflection. This means the `quit` field will no longer be null by the time the action listener is added. In order to perform this sleight of hand, the `SwingEngine` needs a reference to the class with the fields, in this case `HelloWorld`. That is why I passed `this` into the `SwingEngine` constructor. It is this reflection trick that makes SwiXml so powerful. With a reference to your main class, `SwingEngine` can build any user interface and attach it directly to your fields, saving you lots of boilerplate code.

The reflection-based system also has the benefit of keeping your GUI layout separate from your event-handling logic, making maintenance a lot easier in the long run. Note that you may need to make your GUI component fields `public` so that SwiXml can access them. Recent versions of SwiXml can access any field, `public`, `protected`, or `private`; using reflection, however, accessing `private` fields may cause a security violation in sandbox environments like applets or unsigned Java Web Start applications.

SwiXml can handle pretty much every Swing component, including menus, tables, and even inner frames. It uses a simple naming system (usually just dropping the leading "J") so if you already know Swing, it will be easy to learn. All of the standard layouts are supported, as well, though using the `GridBagLayout` from XML can be cumbersome (just as it is in code form). Let's take a look at another example that uses more complicated layout and some nested components.

EmailTest.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<frame size="640,480" title="Email Application"
        defaultCloseOperation="JFrame.EXIT_ON_CLOSE">
    <menubar>
        <menu text="File">
            <menuitem text="New"/>
            <menuitem text="Send"/>
            <menuitem text="Close"/>
            <separator/>
            <menuitem text="Quit" id="quit"/>
        </menu>
        <menu text="Edit">
            <menuitem text="Cut" accelerator="meta X"/>
            <menuitem text="Copy" accelerator="meta C"/>
            <menuitem text="Paste" accelerator="meta V"/>
        </menu>
    </menubar>

    <panel layout="BorderLayout">
        <hbox constraints="BorderLayout.NORTH">
            <vbox>
                <label text="To :"/>
                <label text="Subject :"/>
            </vbox>
            <vbox>
                <textfield Text="joshua.marinacci@sun.com"/>
                <textfield text="SwiXml is really cool!"/>
            </vbox>
        </hbox>

        <scrollpane constraints="BorderLayout.CENTER">
            <textarea>
            </textarea>
        </scrollpane>
    </panel>
</frame>
```

This XML file is launched from code that looks like this:

EmailTest.java

```
import org.swixml.*;

import javax.swing.*;
import java.awt.event.*;

public class EmailTest {

    public JMenuItem quit;

    public EmailTest() throws Exception {
        new SwingEngine(this).render("EmailTest.xml")
                    .setVisible(true);

        quit.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                System.exit(0);
            }
        });

    }

    public static void main(String[] args) throws Exception {
        new EmailTest();
    }

}
```

The finished layout looks like Figure 2:

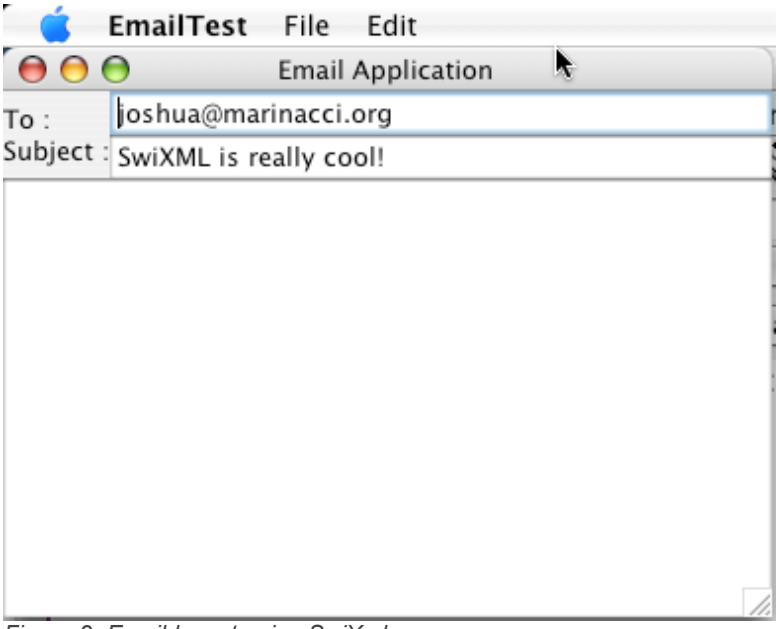

Figure 2. Email layout using SwiXml

SwiXml and Actions

In addition to components, SwiXml has direct support for `Action`s. You can bind a button, menu item, or any other `Action`-aware component directly to an `Action` field in your main class. It also lets you set `Action` properties like mnemonics, accelerators, and tooltips directly from your XML file. These features let you remove more code from your classes, resulting in a cleaner application design and more maintainable programs.

Here is a simple example showing an `Action` shared between two components.

ActionTest.java

```
import java.awt.event.*;
import javax.swing.*;
import org.swixml.*;

public class ActionTest {

    // Create an Action as a member variable
    public Action quit = new AbstractAction() {
        public void actionPerformed(ActionEvent evt) {
            System.exit(0);
        }
    };

    public ActionTest() throws Exception {

        // set the Action's name
        quit.putValue(Action.NAME, "Quit");

        // build the GUI
        new SwingEngine(this).render("ActionTest.xml")
                .setVisible(true);

    }

    public static void main(String[] args) throws Exception {
        new ActionTest();
    }

}
```

The code above creates an `Action` member variable called `quit`. Notice that the `ActionTest` constructor sets the `NAME` property of the action to `"Quit"` **before** building the GUI from XML. This ensures that any components which use the action will start with the same `NAME` property. If the property was set **after** building the GUI, then any customizations done by the XML would be overridden by the `putValue()` call, which may not be the desired behavior.

ActionTest.xml

```
<frame size="200,200" title="Action Test">
    <menubar>
        <menu text="File">
            <menuitem action="quit" accelerator="meta X"/>
        </menu>
    </menubar>

    <button action="quit" text="A Quit Button" ToolTipText="This is a quit button."/>

</frame>
```

The XML file above creates a frame with one button and one menu item. The two components share the same action by using `action` attributes that point to the same named action: `quit`. The `menuitem` has a second attribute, `accelerator`, which overrides the current accelerator property of the action, if any. The `button` element has extra attributes to set the text and tooltip text for the button. These settings will override the defaults in the action, allowing you to customize values for each component. If the properties are later set on the action using `putValue()`, then the new value would override any values set here in the XML. By combining actions and the XML settings in different ways, you can create the appropriate behavior for your application.

Using XInclude

As your interface grows, you may want to split the UI definition into multiple files. SwiXml supports this in two ways. First, you can use multiple files and load them independently. For example, you could create one file for your main application screen, one for the help system, and one for the preferences dialog. Any SwiXml file can be resolved into an instance of `Component`, so you can structure your files to build frames, panels, or even a single text field, each loaded with a call to `SwingEngine.render()`.

SwiXml also supports the XInclude syntax, letting you split your XML into several files that are bound into one master file and loaded as a single unit. This can be used to separate sub-panels and menus into other files without effecting the Java code at all. Here is a simple example that has a main XML file (`XITMain.xml`) containing the main GUI and a secondary file (`XITSub.xml`) containing a sub-panel.

XITMain.xml

```
<frame size="320,150" title="XInclude Test">
    <menubar>
        <menu text="File">
            <menuitem text="Open"/>
            <menuitem text="Save"/>
            <menuitem text="Close"/>
            <menuitem text="Quit"/>
        </menu>
    </menubar>

    <panel constraints="BorderLayout.CENTER">
        <button text="Button in the main file."/>
    </panel>

    <panel constraints="BorderLayout.SOUTH"
        include="XITSub.xml#subpanel"/>
</frame>
```

XITSub.xml

```
<doc>
    <panel id="subpanel" layout="BorderLayout">
        <button text="Button in an included file."/>
    </panel>
</doc>
```

XIncludeTest.java

```
import java.awt.event.*;
import javax.swing.*;
import org.swixml.*;

public class XIncludeTest {
    public XIncludeTest() throws Exception {
        new SwingEngine(this).render("XITMain.xml")
                .setVisible(true);
    }

    public static void main(String[] args) throws Exception {
        new XIncludeTest();
    }
}
```

In the code above, the `SwingEngine` will combine the `XITMain.xml` and the `XITSub.xml` files at run time. The Java code doesn't even know that there are two files involved. It sees a unified component tree and can interact with it as normal. This transparent ability to merge multiple files gives the developer a powerful organizing tool that doesn't clutter up the Swing code at all.

Custom Components

If there isn't a standard Swing class that does what you want, you are always free to build a custom component. SwiXml can support your custom component through a feature called *custom tags*. These let you define new tags that map to your components. For example, suppose I have a custom `JButton` subclass that I use over and over in a drawing program:

ColorButton.java

```
import java.awt.*;
import javax.swing.*;

public class ColorButton extends JButton {
    public void paintComponent(Graphics g) {
        g.setColor(getForeground());
        g.fillRect(0,0,getWidth(),getHeight());
        g.setColor(Color.black);
        g.drawRect(0,0,getWidth()-1,getHeight()-1);
    }
}
```

I can define a custom XML element called `<colorbutton>` by registering it in the `SwingEngine`'s tag library like this:

CustomComponentTest.java

```
import org.swixml.*;
import javax.swing.*;
import java.awt.event.*;

public class CustomComponentTest {
    public CustomComponentTest() throws Exception {
        SwingEngine eng = new SwingEngine(this);
        eng.getTaglib().registerTag("colorbutton",ColorButton.class);
        eng.render("CustomComponentTest.xml").setVisible(true);
    }

    public static void main(String[] args) throws Exception {
        new CustomComponentTest();
    }
}
```

Once the class is registered I can use it in my XML just like any other Swing component:

CustomComponentTest.xml

```
<frame size="320,200">
    <colorbutton constraints="BorderLayout.CENTER" foreground="blue"/>
    <label constraints="BorderLayout.SOUTH" text="Color Test"/>
</frame>
```

Once compiled, the finished program would look like Figure 3:

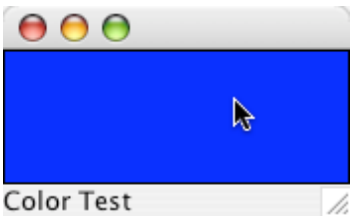

Figure 3. A Custom Component using SwiXml

Conclusion

SwiXml has many features beyond what I have discussed here; these include direct support for resource bundles, making it very easy to localize your application. There are also built-in extensions for Mac OS X, like automatically using the single menu bar and binding actions to the Application menu.

SwiXml is a powerful GUI layout tool, and most importantly, layout is the only thing it does. It doesn't have features to specify event code, animations, or anything else. This means it is easy to learn and small enough to use pretty much anywhere you can use Swing, even in downloaded Java Web Start applications or applets.

Best of all, SwiXml is free and released under an Apache license. This author once created his own similar layout binding system and has now switched entirely to SwiXml because it does the job very well with little extra effort. The net result of working with SwiXml is delivering better-looking applications in less time, and that is always a good thing.

Resources

- Sample code for this article
- SwiXml home page

Joshua Marinacci first tried Java in 1995 at the request of his favorite TA and has never looked back.