Creating a declarative XML UI language

Build a UI and the accompanying framework in the Java language

Arron Ferguson

September 01, 2009

Writing GUIs in program code can often lead to messy design choices, which in turn results in a blurring between business logic and UI code. Discover how to create a declarative XML UI tag set with an accompanying Java™ framework that parses, validates, constructs, and finally binds the declared GUI components to business logic at runtime.

Introduction

GUI development can be a daunting task. GUI frameworks are not always well documented, and the amount of required code can grow quickly, slowing down development workflow. On top of that, drag-and-drop tools and IDEs that support these GUI frameworks can often steer the GUI software developer towards creating unmanageable and unreadable code. This can further blur the line between business logic and the code describing the GUI which can make maintenance of the software more difficult.

Frequently used acronyms

- API: Application programming interface
- DOM: Document Object Model
- GUI: Graphical user interface
- HTML: Hypertext Markup Language
- IDE: Integrated Development Environment
- JAR: Java Archive
- UI: User interface
- URI: Uniform Resource Identifier
- XML: Extensible Markup Language
- XSD: XML Schema Infoset Model
- W3C: World Wide Web Consortium

This is where a declarative UI language becomes handy. A UI language describes the "what," not the "how." For example, HTML describes the content that displays, not the rendering functions used to render content. By not specifying the "how" in declarative languages, control flow is also left out. Although this loss sounds like a limitation, it becomes a strength, as side effects—such as modifying global state (variables for example) or calling other functions or methods—are
eliminated. Choosing a declarative language also offers the benefit of separating the UI code from the application code. This decoupling can offer future benefits such as a clear distinction between project and team roles which might even reduce costs for integrating business logic with multiple views or view technology.

Today, a fair number of examples of declarative XML UIs are in use. Linux® and UNIX® operating systems that use the GNOME desktop environment have Glade. Microsoft® Windows® users have Extensible Application Markup Language (XAML), which supports a rich set of features including code insertion within the XML. The Adobe® Flex® Framework's MXML format describes GUIs for the Adobe Shockwave (SWF) player and also includes code insertion. See Related topics for links to more information.

A set of requirements for a basic declarative UI framework in Java technology might be:

- **Validation**: Using XML Schema
- **A DOM**: A custom DOM to handle specifics such as keeping in sync the GUI component state and XML node state
- **Persistence**: Marshalling and unmarshalling of the GUI
- **Image data**: Stored as Base64 data
- **Swing components**: Representation of the more commonly used Swing components for GUI development

With these requirements in mind, it's time to create the declarative XML.

### Declarative XML

A first attempt at the XML format, in **Listing 1**, demonstrates a simple window, a panel, and a button. The attributes found in **Listing 1** represent fundamentally required properties such as coordinates, dimensions, and unique identifiers that refer to individual in-memory components.

**Listing 1. Declarative XML concept**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xui:XUI>
  <xui:Window id="window_0" name="Hello World" width="300" height="300" x="426" y="282" visible="true">
    <xui:GridLayout height="1" width="1"></xui:GridLayout>
  </xui:Window>
  <xui:Panel id="panel_0" x="0" y="0" name="Hello Panel" width="1" height="1">
    <xui:GridLayout height="1" width="1"></xui:GridLayout>
    <xui:Button x="0" y="0" width="1" height="1" id="button_0" label="Press Me" enabled="true" selected="true" orientation="horizontal"/>
  </xui:Panel>
  <xui:Resource type="java" class="ButtonModel" uri="model.jar"/>
</xui:XUI>
```

### The XML schema

This declarative XML UI will map the XML elements to the **Java Swing framework** which offers the greatest in portability since Swing is guaranteed to be available on all current Java run time...
environments. Many Swing components will have representative XML elements within the XML format.

The framework uses an XML Schema. XML Schema allows enforcement of specified ordering, cardinality, and data types within a schema instance. This is important; the framework will expect a certain set of XML elements of specified type and in a particular order. **Listing 2** demonstrates the initial elements and attributes of the hierarchy within an XML schema instance.

**Listing 2. Declarative XML UI schema: initial elements**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified"

targetNamespace="http://xml.bcit.ca/PurnamaProject/2003/xui"
xmns:xs="http://www.w3.org/2001/XMLSchema"
xmns:xui="http://xml.bcit.ca/PurnamaProject/2003/xui">
 <xs:element name="XUI">
  <xs:complexType>
   <xs:sequence>
    <xs:element minOccurs="0" maxOccurs="128" ref="xui:Window"/>
    <xs:element minOccurs="0" maxOccurs="1" ref="xui:Resource"/>
   </xs:sequence>
   <xs:attribute name="id" type="xs:anyURI" use="required"/>
  </xs:complexType>
 </xs:element>

 <xs:element name="Resource">
  <xs:complexType>
   <xs:sequence>
    <xs:attribute name="uri" type="xs:anyURI" use="required"/>
    <xs:attribute name="class" type="xs:token" use="required"/>
    <xs:attribute name="type" use="required">
     <xs:simpleType>
      <xs:restriction base="xs:token">
       <xs:enumeration value="java"/>
       <xs:enumeration value="groovy"/>
      </xs:restriction>
     </xs:simpleType>
    </xs:attribute>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="Window">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="xui:GridLayout"/>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
     <xs:element ref="xui:BasicDialog"/>
     <xs:element ref="xui:OpenFileDialog"/>
     <xs:element ref="xui:SaveFileDialog"/>
     <xs:element ref="xui:CustomDialog"/>
     <xs:element ref="xui:Panel"/>
     <xs:element ref="xui:SplitPanel"/>
     <xs:element ref="xui:TabbedPanel"/>
    </xs:choice>
   <xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
    <xs:attribute name="x" type="xs:short" use="required"/>
    <xs:attribute name="y" type="xs:short" use="required"/>
    <xs:attribute name="width" type="xs:unsignedShort" use="required"/>
    <xs:attribute name="height" type="xs:unsignedShort" use="required"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```
Look at the schema in detail. First, the XML declaration must come before anything—even before spaces and comments as indicated the XML Recommendation. Next, the schema element contains other elements:

- **elementFormDefault="qualified"** states that all elements must have a namespace—either a prefix or a default namespace.
- **targetNamespace="http://xml.bcit.ca/PurnamaProject/2003/xui"** specifies the target namespace URI.
- The schema instance uses the W3C XML Schema Recommendation and all elements within it (**xmlns:xs="http://www.w3.org/2001/XMLSchema"**).

Using namespaces within an XSD is important: It eliminates namespace collisions. A namespace collision occurs when two or more elements from two or more XML formats have the same name. This collision causes confusion to any application interested in its respective tag set. By using namespaces and accompanying namespace prefixes, you avoid this problem altogether.

Next, the root-level data type element **xui** states that:

- It permits one sequence of 0 to 128 **Window** elements and, at the end, one **Resource** element. Both of these are referenced elements found later in the schema instance.
- It has an **id** attribute that is required and has to be of type **anyURI**.

The **xui** element (possibly) contains many **Window** elements, although it might have no **Window** elements based on the a value of 0 in the **minOccurs** attribute. As for the **Resource** element:

- It has an empty content model because of its empty **xs:sequence** element.
- It has three attributes, all required.
- The last attribute, the **type** attribute, creates a derived simple type from XSD's defined type (token), where the restriction facet is **enumeration**, allowing for the enumerated literal text values of java and groovy.

The purpose of the **Resource** element is to provide the Java framework the URI of a resource (a JAR, in this case) that contains compiled Java classes that can be loaded at run time and bound to. This resource relies on a particular class (the value of the **class** attribute) that will be called, essentially offering an exposed class that will answer to all events generated from the GUI.
The **Window** element:

- Contains a sequence of one **GridLayout**, an unlimited choice among a **BasicDialog**, **OpenFileDialog**, **SaveFileDialog**, **CustomDialog**, **Panel**, **SplitPane**, and **TabbedPane** elements, and, at the end, either zero or one **MenuBar**.
- Has seven attributes—all required—that use various defined data types (notice the `xs` prefix) within the XML Schema Recommendation.

The **Window** can contain many different top-level and intermediate-level containers. The **Window** element references a **GridLayout** element. The **GridLayout** element specifies the dimensions of a grid of cells that components can occupy. The **GridLayout** offers layout features similar to `java.awt.GridBagLayout` in the Java environment except without all of the complexity.

Looking no further, it is apparent that the XML Schema is substantially expressive. **Listing 3** shows a few more elements.

**Listing 3. Declarative XML UI schema: more elements**

```xml
...<xs:element name="CustomDialog">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="xui:GridLayout"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:Panel"/>
    </xs:sequence>
    <xs:attribute name="modal" type="xs:boolean" use="required"/>
    <xs:attribute name="idref" type="xs:IDREF" use="optional"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="id" type="xs:ID" use="required"/>
    <xs:attribute name="x" type="xs:short" use="required"/>
    <xs:attribute name="y" type="xs:short" use="required"/>
    <xs:attribute name="width" type="xs:unsignedShort" use="required"/>
    <xs:attribute name="height" type="xs:unsignedShort" use="required"/>
    <xs:attribute name="visible" type="xs:boolean" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="Panel">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="1" minOccurs="1" ref="xui:GridLayout"/>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:Button"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:Calendar"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:CheckBox"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:ComboBox"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:HypertextPane"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:Image"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:Label"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:List"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:PasswordField"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:ProgressBar"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:RadioButton"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:SliderBar"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:Table"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:TextArea"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:TextField"/>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="xui:Tree"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
...
Notice that no volatile state information is stored—only state information that might help in the reconstruction of GUI components. An example is the state information of the CustomDialog element:

- The number of Panel elements allowed in the dialog
- Whether the dialog is modal (A modal dialog traps focus until the user closes the dialog.)
- The coordinates (x and y in pixels) within the desktop
- The dimensions (width and height in pixels)
- The window’s visibility

A Panel is an intermediate container and permits a rather large number of contained atomic components. Referring back to Listing 3, the Panel has exactly one GridLayout and a choice of either no atomic components placed inside of the Panel or as many as needed. The Panel itself has x and y coordinates. However, rather than referring to pixels within the desktop (as the CustomDialog does), the Panel uses the x and y coordinates to refer to positioning within the parent container's GridLayout. Like a Russian doll, this nested composition closely mimics Swing’s layout rules. With all of this in place, it’s time to address the software implementation.

**Supporting Java framework**

Let’s start with an overview of the proposed Java framework. The code in Listing 4 shows the steps the application programmer must follow to create an application.
Listing 4. Java API Calls Concept

try {
    // Gain access to a XUI builder through factory
    // In this framework the term XUI is going to represent the custom DOM
    XUIBuilder builder = XUIBuilderFactory.getInstance().getXUIBuilder();  // (1)

    // Validate and parse (unmarshal) the XML document
    builder.parse("browser.xml"); // (2)

    // Build a custom DOM
    XUI xui = builder.getXUIDocument();  // (3)

    // Create 1:1 GUI component mapping to custom DOM
    xui.visualize(); // (4) (5)

    // Create bindings to data model (i.e. JAR file from Resource element)
    xui.bind(); // (6)

    // Get root node from the XUI document
    XUINode root = xui.getRoot();

    // Save a copy of the DOM to file (marshal)
    xui.marshalXUI("browser-marshalled.xml");
}
} catch (XUIParseException xpe) {
    xpe.printStackTrace();
} catch (XUIBindingException xbe) {
    xbe.printStackTrace();
} catch (IOException ioe) {
    ioe.printStackTrace();
}

The steps in Listing 4 define a clear separation of functionality and allow for further refinement of the framework's components. An attempt at visualizing this flow is shown in Figure 1. Each circled number in Figure 1 coincides with each commented number in Listing 4 although the code demonstrates two additional steps (retrieve a reference to the xui root node and marshal the DOM to file). The steps are:

What is a Document Object Model?

A document object model (DOM for short) is a mapping of XML elements to in-memory objects (provided by an API) thus allowing programmers to read XML data into in-memory objects (called unmarshalling), manipulate data, and then write data back out to XML (called marshalling). The most well-known and commonly used DOM is the W3C DOM.

Figure 1 illustrates the following steps

1. A Builder is retrieved from aBuilderFactory.
2. The Builder first ensures the XML document has been validated and parsed before it allows retrieval of a XUI document. If parsing or validation fail, a XUIParseException will occur and the framework will abort document loading.
3. The Builder creates the DOM where the objects reflect the XML elements that were read in.
4. A Realizer object, called internally by the xui object, is instantiated and made ready to perform the next step.
5. Realizing is where the framework creates a hierarchy of GUI components based on the hierarchy of XML nodes created previously (the true core of the framework’s engine).
6. Using the power of reflection in the Java environment, the model logic (the portion of the application that drives the UI) is bound to the GUI components that have just been realized.

**Figure 1. Framework flow and detailed view of steps XUI API takes to build a GUI**

This six-step call flow is easy to use but contains a flurry of messages and object instantiations under the hood, something worth exploring next. The heart of the framework is in steps 5 and 6.

**GUI component & XML node composition**

In **Figure 1**, step 5 creates a component model. This allows a pairing of the XML node (now an in-memory object) with a GUI component. This pairing requires a very strict synchronization of the following events:

- For every `xuinode` (the in-memory object representing any XML element) read in by the framework, a `xuicomponent` must be created to encompass the `xuinode`.
- For every `xuicomponent` created in memory, a GUI peer, such as a `javax.swing.JFrame`, must be created.
- Every time a `xuiComponent` instance—or one of its sub-types such as a `xUIButton`—is modified (for example, when dimensions are changed), the `xuiComponent` will ensure that both the `xuinode` and the GUI peer will be updated simultaneously and equivalently.

By satisfying the requirements above, the framework allows programmers to read XML documents in (unmarshalling), modify the DOM, and save changes back to an XML document (marshalling). Programmers can even create new DOMs programmatically and marshal them.
DOM node marshalling
In order for a XUINode to marshal itself as XML, a customized implementation of the toString method is provided (in Listing 5). The root node can contain many child nodes. Each child node can contain its own set of child nodes and so on. By calling the toString method of the root-level node, the framework can easily marshal the entire XML document. The namespaces are added and each element is made aware of its level within the hierarchy (through the level variable). That way, when the toString method is called, it offers indentation for easier manual reading of these documents.

Listing 5. XUINode toString method implementation

```java
@Override
public String toString() {
    StringBuffer sb = new StringBuffer();
    String namespacePrefix = "";
    // insert indenting ... 2 spaces for now.
    if(isRoot) {
        sb.append(XMLPI + "\n");
        sb.append(API_COMMENT + "\n");
    } else {
        sb.append("\n");
        for(int s = 0; s < level; s++) {
            sb.append(" ");
        }
    }
    sb.append("<");
    // get namespaces for this node
    Enumeration keys = nameSpaces.keys();
    String names = "";
    while(keys.hasMoreElements()) {
        String uri = (String)keys.nextElement();
        String prefix = (String)nameSpaces.get(uri);
        /* if its the xsi namespace (XML Schema Instance),
         * ignore it, this isn't part of that namespace but it is
         * needed for the XML Schema validator to work. */
        if(!(prefix.equals("xsi"))) {
            sb.append(prefix + ":");
            namespacePrefix = prefix;
        }
        names += (" " + "xmlns:" + prefix + "=" + uri + "\""");
    }
    if(beginOfNamespace) {
        sb.append(name + names);
    } else {
        sb.append(name);
    }
    // do attributes if there are any
    if(attributes.getLength() > 0) {
        int length = attributes.getLength();
        for(int i = 0; i < length; i++) {
            String attributeValue = attributes.getValue(i);
            String attributeQName = attributes.getQName(i);
            sb.append(" " + attributeQName + "=" + attributeValue + "\"");
        }
    }
    sb.append(">");
    sb.append(cdata);
    int size = childNodes.size();
    for(int i = 0; i < size; i++) {
        XUINode e = (XUINode)childNodes.get(i);
        sb.append(e.toString());
    }
    return sb.toString();
}
```
Adding to a container component

Another section worth exploring is the container type XUIWindow, which is an indirect sub-type of XUIComponent. The XUIWindow implementation represents a javax.swing.JFrame component and therefore must allow child components to be added to the layout. Listing 6 demonstrates the implementation. The first step is to ensure that only certain types of components can be added to the XUIWindow. If so, the XUIComponent’s DOM node representation, a XUINode, is retrieved in order to access properties of that component. Note that this requires all XUIComponents' constructors to initialize these values.

A further check is made to ensure that the component is an intermediate container (for example, an XUIPanel) and that the intermediate container fits within the XUIWindow's grid of rows and columns. Finally, the component can be added to the XUIWindow by ensuring the component is enabled, set in the correct position within the layout grid, and that the XUIWindow's XUINode (the win variable) is given a reference to the new child component's XUINode—the addChildNode() call.

Listing 6. XUIWindow addComponent Method implementation

```java
public void addComponent(XUIComponent component) throws XUITypeFormatException {
    if(component instanceof XUIBasicDialog
            || component instanceof XUIOpenFileDialog
            || component instanceof XUICustomDialog
            || component instanceof XUIMenuBar
            || component instanceof XUIPanel
            || component instanceof XUISplitPanel
            || component instanceof XUITabbedPanel
            || component instanceof XUISaveFileDialog) {
        // get the node
        XUINode node = component.getNodeRepresentation();

        if(!(component instanceof XUIMenuBar)) {
            int x = Integer.parseInt(node.getAttributeValue("x"));
            int y = Integer.parseInt(node.getAttributeValue("y"));
            int width = Integer.parseInt(node.getAttributeValue("width"));
            int height = Integer.parseInt(node.getAttributeValue("height"));

            // can't add dialogs so need to check for type here.
            if(component instanceof XUIBasicDialog
                    || component instanceof XUIOpenFileDialog
                    || component instanceof XUICustomDialog
                    || component instanceof XUISaveFileDialog) ; // nothing
            else {
                // check to make sure it fits within the grid.
                Dimension localGrid = this.getGrid();
                if(width > localGrid.getWidth() || height >
```

```java
```
localGrid.getHeight()) {
    throw new XUITypeFormatException(node.getName()
        + " (id: " + node.getAttributeID()
        + ") must be within this window's grid width and"
        + "height (w: " + localGrid.getWidth()
        + " + h: " + localGrid.getHeight() + ")");
}  
Rectangle rect = new Rectangle(y, x, width, height);
    component.getPeer().setEnabled(true);
    frame.getContentPane().add(component.getPeer(), rect);
    // for mapping components to the regions they occupy
    childComponentMappings.put(component, rect);
}  
    component.setComponentLocation(x, y);
} else {
    // do specifics for a menubar
    frame.setJMenuBar((JMenuBar)component.getPeer());
}

frame.invalidate();
frame.validate();

// add the component's node
    int level = win.getLevel();
    node.setLevel(++level);
    if(win.getParent() == null)
        win.addChildNode(node);
} else {
    StringBuffer sb = new StringBuffer();
    sb.append("Type not supported in XUIWindow. ");
    sb.append("The following types are supported:
    ");
    for(int i = 0; i < supportedComponents.size(); i++) {
        String s = (String)supportedComponents.get(i);
        sb.append(" - " + s + "\n");
    }  
    throw new XUITypeFormatException(sb.toString());
}

Binding

One last area of code worth inspecting is the handling of run time binding. When the bind method of the XUI object is called, an instance of the BindingFactory is invoked.

The BindingFactory's doBinding method (in Listing 7) has to do several things to bind the model code to the constructed GUI:

- Grab the URL, whether local, on the Internet, or relative.
- Peer into the JAR through the JarURLConnection class and load classes using a custom and separate class loader.
- Look for a class that matches the name of the Resource element's class attribute from the loaded XML document. That class is the entry point into the model.
- Instantiate an instance of the entry point class using the Java reflection framework and call its init method. The init method is similar in concept to the main method of a typical Java class in that they both are entry points.
If the JAR file contains images, load them into memory, as well.

**Listing 7. The BindingFactory's doBinding Method**

```java
public void doBinding(XUI resource, XUI xui) throws XUIBindingException, IOException {
    MalformedURLException, IOException {
        if (resource.getAttributeValue("type").equals("java")) {
            String className = resource.getAttributeValue("class");
            String aURLString = resource.getAttributeValue("uri");
            URL url = null;
            // get the url ... if it's not a valid URL, then try and grab
            // it as a relative URL (i.e. java.io.File). If that fails
            // re-throw the exception, it's toast
            try {
                url = new URL("jar:" + aURLString + ":/!");
            } catch (MalformedURLException mue) {
                String s = "jar:file://" + new File(aURLString)
                    .getAbsolutePath().replace("\", "/") + ":/!";
                url = new URL(s);
            }
            // it really was malformed after all
            if(url == null) {
                throw new MalformedURLException("Couldn't bind to: " + aURLString);
            }
            // get a jar connection
            JarURLConnection jarConnection = (JarURLConnection)url.openConnection();
            // get the jar file
            JarFile jarFile = jarConnection.getJarFile();
            // jar files have entries. Cycle through the entries until finding
            // the class sought after.
            Enumeration entries = jarFile.entries();
            // the class that will be the entry point into the model
            JarEntry modelClassEntry = null;
            Class modelClass = null;
            XUIClassLoader xuiLoader = new XUIClassLoader(this.getClass().getClassLoader());
            while(entries.hasMoreElements()) {
                JarEntry remoteClass = (JarEntry)entries.nextElement();
                // load the classes
                if(remoteClass.getName().endsWith(".class")) {
                    // have to get the second last word between period marks. This
                    // is because the convention allows for:
                    // org.purnamaproject.xui.XUI
                    // that is, the periods can represent packages.
                    StringTokenizer st =
                        new StringTokenizer(remoteClass.getName(), ".");
                    String previousToken = st.nextToken();
                    String currentToken = ";
                    String nameOfClassToLoad = previousToken;
                    while(st.hasMoreTokens()) {
                        currentToken = st.nextToken();
                        if(currentToken.equals("class"))
                            nameOfClassToLoad = previousToken;
                        else {
                            nameOfClassToLoad += currentToken;
                        }
                    }
                    // get an output stream (byte based) attach it to the
                    // inputstream from the jar file based on the jar entry.
                    ByteArrayOutputStream baos = new ByteArrayOutputStream();
                    InputStream is = jarFile.getInputStream(remoteClass);
                    final byte[] bytes = new byte[1024];
                    int read = 0;
                    while ((read = is.read(bytes)) >= 0) {
                        baos.write(bytes, 0, read);
                    }
                }
            }
        }
    }
}
```
Class c = xuiLoader.getXUIClass(nameOfClassToLoad, baos);
// check for the class that has the init method.
if(remoteClass.getName().equals(className + ".class")) {
    modelClassEntry = remoteClass;
    modelClass = c;
} else {
    String imageNameLowerCase = remoteClass.getName().toLowerCase();
    if(imageNameLowerCase.endsWith(".jpeg")
        || imageNameLowerCase.endsWith(".jpg")
        || imageNameLowerCase.endsWith(".gif")
        || imageNameLowerCase.endsWith(".png")) {
        // add resources (images)
        XUIResources.getInstance().addResource(remoteClass, jarFile);
    }
}

// now instantiate the model.
try {
    // create a new instance of this class
    Object o = modelClass.newInstance();
    // get the method called 'init'. This is part of the API
    // requirement
    Method m = modelClass.getMethod("init", new Class[] {XUI.class});
    // at last, call the method up.
    m.invoke(o, new Object[] {xui});
} catch(InstantiationException ie) {
    ie.printStackTrace();
} catch(IllegalAccessException iae) {
    iae.printStackTrace();
} catch(NoSuchMethodException nsm) {
    nsm.printStackTrace();
} catch(InvocationTargetException ite) {
    System.out.println(ite.getTargetException());
    ite.printStackTrace();
} else {
    throw new XUIBindingException(
        "This platform/API requires Java libraries.");
}

Having looked at the mechanics of this framework, it's time to take the framework for a test drive and show off one of the example applications.

Framework in action

The project framework (see Download) contains several examples. The Web browser example is fairly exhaustive in what it performs.

Web browser XML UI document

This example offers a reasonable real-world example of what you might expect to place within a declarative XML UI document. Looking at Listing 8, the main window has x and y coordinates specified as well as an id value. All elements must have unique ID values for the business logic to be able to refer to these components.
The `Window` element contains several child elements including:

- A `Panel` which offers the main layout
- An `OpenFileDialog` for opening new Web pages
- A `SaveFileDialog` for saving the currently viewed Web page
- A `CustomDialog` that displays a yes or no quit dialog
- A `CustomDialog` that displays Web bookmarks
- A `MenuBar` that is displayed at the top of the `Window` and that offers menu item functionality
- A `Resource` for referencing the Java model code that will drive this UI

All coordinates of contained components (such as a `Button`) refer to positions within the grid. All dimensions of contained components refer to how many cells wide and tall each component is within the grid. The definitions of components are highly declarative in that they define properties but not the logic on how those properties are to be used or created. A few other points of interest within this document are:

- `MenuItems` can have quick keys, such as Ctrl-X to exit the application.
- The `Window` has dialogs, but, by default, these dialogs are not made visible until the user invokes them.
- All containers (for example, a `Panel`) must have layouts and must specify the number of rows and columns in that layout.

### Listing 8. Web browser XML UI

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated by The Purnama Project XUI API version 0.5 -->
<xui:xui xmlns:xui="http://xml.bcit.ca/PurnamaProject/2003/xui"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xml.bcit.ca/PurnamaProject/2003/xui ../../xui.xsd"
    id="http://xml.bcit.ca/PurnamaProject/examples/XUIWebBrowser">
    <xui:Window id="window_0" name="XUI Web Browser" x="200" y="20" width="800"
        height="600" visible="true">
        <xui:GridLayout width="1" height="1"></xui:GridLayout>
        <xui:Panel x="0" y="0" width="1" height="1" id="panel_0" name="main panel"
            idref="window_0">
            <xui:GridLayout width="8" height="8"></xui:GridLayout>
            <xui:HypertextPane x="1" y="0" width="8" height="7" id="hyper_0"
                url="http://www.w3c.org"></xui:HypertextPane>
            <xui:Button  x="0" y="0" width="1" height="1" id="button_0" label="Back"
                enabled="true" orientation="horizontal"></xui:Button>
            <xui:Button  x="0" y="3" width="1" height="1" id="button_1" label="Home"
                enabled="true" orientation="horizontal"></xui:Button>
            <xui:Button  x="0" y="7" width="1" height="1" id="button_2"
                label="Forward" enabled="true" orientation="horizontal"></xui:Button>
        </xui:Panel>
    </xui:Window>
    <!-- For opening files. Only want to see html files -->
    <xui:OpenFileDialog x="10" y="10" width="400" height="300"
        id="filedialog_0" idref="window_0" visible="false">
        <xui:Filter>html</xui:Filter>
        <xui:Filter>htm</xui:Filter>
    </xui:OpenFileDialog>
    <!-- For saving files. Only want to save html files -->
    <xui:SaveFileDialog x="10" y="10" width="400" height="300"
        id="savedialog_0" idref="window_0" visible="false">
        <xui:Filter>html</xui:Filter>
        <xui:Filter>htm</xui:Filter>
    </xui:SaveFileDialog>
</xui:xui>
```
/!– Ask the user if they really want to quit -->
<xui:CustomDialog x="200" y="200" width="320" height="160"
    id="customdialog_1" idref="window_0" name="Exit Purnama Browser"
    modal="true" visible="false">
    <xui:GridLayout width="1" height="1"></xui:GridLayout>
    <xui:Panel x="0" y="0" width="1" height="1" id="panel_2"
        name="Quit Panel" idref="customdialog_0">
        <xui:GridLayout width="5" height="4"></xui:GridLayout>
        <xui:Label x="1" y="1" width="3" height="1"
            text="Do you really want to exit?"></xui:Label>
        <xui:Button x="2" y="1" width="1" height="1" id="button_3"
            label="Yes" enabled="true" orientation="horizontal"></xui:Button>
        <xui:Button x="2" y="3" width="1" height="1" id="button_4"
            label="No" enabled="true" orientation="horizontal"></xui:Button>
    </xui:Panel>
</xui:CustomDialog>

<!-- For displaying the bookmarks -->
<xui:CustomDialog x="100" y="100" width="300" height="300"
    id="customdialog_0" idref="window_0" name="Bookmarks" modal="false"
    visible="false">
    <xui:GridLayout width="1" height="1"></xui:GridLayout>
    <xui:Panel x="0" y="0" width="1" height="1" id="panel_1"
        name="bookmarks panel" idref="customdialog_0">
        <xui:GridLayout width="1" height="1"></xui:GridLayout>
        <xui:List x="0" y="0" width="1" height="1" id="list_0"
            enabled="true" itemSelected="0" scrolling="vertical">
            <xui:ListItem>http://www.w3c.org</xui:ListItem>
            <xui:ListItem>http://www.agentcities.org</xui:ListItem>
            <xui:ListItem>http://www.apache.org</xui:ListItem>
            <xui:ListItem>http://www.gnu.org</xui:ListItem>
        </xui:List>
    </xui:Panel>
</xui:CustomDialog>

<!-- The menu bar with pop-up menu items too -->
<xui:MenuBar id="menuBar_0" idref="window_0">
    <xui:Menu id="menu_0" idref="menuBar_0" enabled="true"
        isPopupMenu="false" isSubMenu="false" label="File">
        <xui:MenuItem id="mi_1" idref="menu_0" enabled="true" label="Open URL"
            checked="false">
            <xui:Shortcut keyCode="F" keyModifier1="ALT"></xui:Shortcut>
        </xui:MenuItem>
        <xui:MenuItem id="mi_0" idref="menu_0" enabled="true" label="Save"
            checked="false">
            <xui:Shortcut keyCode="F" keyModifier1="ALT"></xui:Shortcut>
        </xui:MenuItem>
        <xui:MenuItem id="mi_2" idref="menu_0" enabled="true" label="Exit"
            checked="false">
            <xui:Shortcut keyCode="X" keyModifier1="CTRL"></xui:Shortcut>
        </xui:MenuItem>
    </xui:Menu>
    <xui:Menu id="menu_1" idref="menuBar_0" enabled="true"
        isPopupMenu="false" isSubMenu="false" label="Bookmarks">
        <xui:MenuItem id="mi_3" idref="menu_1" enabled="true" label="Add Bookmark"
            checked="false">
            <xui:Shortcut keyCode="D" keyModifier1="CTRL"></xui:Shortcut>
        </xui:MenuItem>
        <xui:MenuItem id="mi_4" idref="menu_0" enabled="true" label="Manage Bookmarks"
            checked="false">
            <xui:Shortcut keyCode="M" keyModifier1="CTRL"></xui:Shortcut>
        </xui:MenuItem>
    </xui:Menu>
    <xui:Menu id="menu_2" idref="hyper_0" enabled="true" isPopupMenu="true"
        isSubMenu="false" label="">
Of course, none of this is of any value without user interaction, which is next.

Web browser Java code model logic

In Listing 8, the Resource element contained the name of the class that acts as an entry point into the application model. The name given was BrowserModel and so, on the Java end, the name of the compiled class must match. This includes the namespace which, in this case is the default namespace.

Any class can therefore act as the entry point into the model portion of the application as long as its name is the same as the class attribute value of the Resource element. For user interaction to be correctly wired at runtime, the implementing class must follow several other rules:

- Have a method with the following signature: public void init(XUI document).
- Implement the appropriate event handling interface to listen to events (such as ActionModel for XUIButton implementations).
- Use the id values of the XML elements to reference the GUI components. (This can be done using several different methods found within the xui class.)
- Add itself as a listener to the appropriate component. All event-generating components within this framework, such as the XUIButton class implementation, implement XUIEventSource, and therefore, generate UI events.

In Listing 9, the BrowserModel class performs its initialization within the init method. This includes gaining references to components through id values, creating menu items that contain Web URL bookmarks, and adding itself as a listener to components through the addEventListener method. The BrowserModel can add itself as a listener because it is an XUIModel (ActionModel is a sub-type of XUIModel). It is also worth mentioning that the XUIComponentFactory class offers many methods to create XUI components.

Listing 9. Partial code listing: initialization

```java
import org.purnamaproject.xui.binding.ActionModel;
...

public class BrowserModel implements ActionModel, TextModel, WindowModel, ListActionModel {
    ...
```
private XUI xui;
...
public void init(XUI document) {
    xui = document;
    ...
    bookmarksList = (XUIList)xui.getXUIComponent("list_0");
    homeButton = (XUIButton)xui.getXUIComponent("button_1");
    ...
    List bookmarks = bookmarksList.getItems();
    for(int i = 0; i < bookmarks.size(); i++) {
        String url = (String)bookmarks.get(i);
        XUIMenuItem aMenuItem = XUIComponentFactory.makeMenuItem(url);
        bookmarksMenu.addMenuItem(aMenuItem);
        linkModel.addSource(aMenuItem);
        aMenuItem.addEventListener(linkModel);
    }
    ...
    homeButton.addEventListener(this);
    ...
}

Digging deeper, Listing 10 presents the event handling code for various components. For example:

- **openMenuItem** will cause a **fileDialog** to appear (a modal dialog to open a locally stored Web page).
- **homeButton** and **popuphomeMenuItem** (right click to access within the window) both call the **doHome** method which will direct the browser to the **uri** attribute value of the **HypertextPane** element (from Listing 8).
- **fileDialog** will load a new file and then increment the **index** variable which is used by the application's **queue** in order to track previously visited Web pages.

**Listing 10. Partial Code Listing: Event Handling**

```java
public void action(XUIComponent component) {
    if(component == openMenuItem) {
        fileDialog.setVisible(true);
    } else if(component == homeButton || component == popuphomeMenuItem) {
        doHome();
    } else if(component == prevButton || component == popupprevMenuItem) {
        doPrevious();
    } else if(component == nextButton || component == popupnextMenuItem) {
        doNext();
    } else if(component == fileDialog) {
        if(fileDialog.getSelectedFile() !=null)
            hyperTextPane.setURL(fileDialog.getSelectedFileAsURL());
        index++;
        if(index != queue.size()) {
            nextButton.setEnabled(false);
            popupnextMenu.setEnable(false);
            for(int i = index; i < queue.size(); i++) {
                queue.remove(i);
            }
        }
        queue.add(hyperTextPane.getURL());
    }
```
prevButton.setEnabled(true);
popupprevMenuItemsetEnabled(true);

} else if(component == saveDialog) {
    try {
        FileOutputStream fos = new FileOutputStream(saveDialog.getSelectedFile());
        hyperTextPane.getDocument().writeTo(fos);
    } catch (FileNotFoundException fnfe) {
        fnfe.printStackTrace();
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}

} else if(component == popupsaveasMenuItem || component == saveMenuItem) {
    saveDialog.setVisible(true);
}

} else if(component == popupbookmarkMenuItem || component == bookmarkMenuItem) {
    doBookmark(hyperTextPane.getURL());
}

} else if(component == notDontExit) {
    exitDialog.setVisible(false);
    browserWindow.setVisible(true);
}

} else if(component == yesExit) {
    System.exit(0);
}

} else if(component == exitMenuItem) {
    exitDialog.setVisible(true);
}

} else if(component == manageBookmarksMenuItem) {
    bookmarksDialog.setVisible(true);
}

}

The final application (in Figure 2) demonstrates a basic Web browser that allows you to display local pages, Web-based pages, and previously visited Web pages, plus the ability to manage bookmarks.
Figure 2. Screenshot of Web browser

You'll find several other example applications in the Download for this article.

Gotchas and challenges

As exciting as this solution is, the approach is rather idealistic: Security issues within this framework have been ignored. Recall how the API innocently loaded a JAR file from any URI. Recall the Resource element shown in Listing 8. The type is literally anyURI. This means local file, file on the network, file on the Internet. Anywhere. Should an application trust business logic from anywhere? Clearly, you want to consider some sort of security model to limit the loading of untrusted resources. One way to address this problem is to limit URIs is to reference a lookup table. Another (cleaner) solution is to use digital certificates.

Lastly, consider the loading of other XML formats within this declarative XML UI format. The XML Schema supports this due to the required use of namespaces. As an example, you can embed a separate XML format to represent scalable vector graphics within the XML document.

Conclusion

This article covered what a declarative XML UI language is and what one looks like. It introduced an accompanying Java framework as well as an example application—a Web browser. Finally, it brought up potential security issues and concerns.

Creating declarative XML UIs is certainly not new. It is, however, an area of software development that is maturing and becoming more commonplace. One plus is that creating declarative XML UIs helps promote software reuse and modularity.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>XUI XSD and Java API for article¹</td>
<td>xui.zip</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>30KB</td>
</tr>
</tbody>
</table>

**Note**

1. This .zip file contains all the source code for this project, the Apache Ant build file, XML schema, examples, and third-party libraries (JARs) that this article references and uses.
Related topics

- **XML Schema Part 0: Primer**: Get a good start towards learning XML schema in this W3C primer.
- **XML Schema Part1: Structures**: Learn the data structures within XML schema.
- **XML Schema Part 2: Datatypes**: Review this W3C chapter about the defined data types within XML schema.
- **Swing tutorial**: Take Sun's online tutorial covers in great length how to develop GUIs for applications and applets in Swing.
- **Document Object Model (DOM) on W3C**: Learn more about the W3C Document Object Model in this overview of DOM-related material.
- **Base64 data** (Wikipedia): Read a good explanation of Base64 encoding.
- **XAML**: Learn more about XAML.
- **Tip: Work with schemas and namespaces** (Brett McLaughlin, developerWorks, September 2002): Demystify namespace coverage within XML schema, including multiple namespaces.
- **Glade**: Learn more about and download Glade, a RAD tool to enable quick, easy development of UIs for the GTK+ toolkit and the GNOME desktop environment.
- **IBM certification**: Find out how you can become an IBM-Certified Developer.
- **XML technical library**: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- **IBM product evaluation versions**: Get your hands on application development tools and middleware products.

© Copyright IBM Corporation 2009

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)