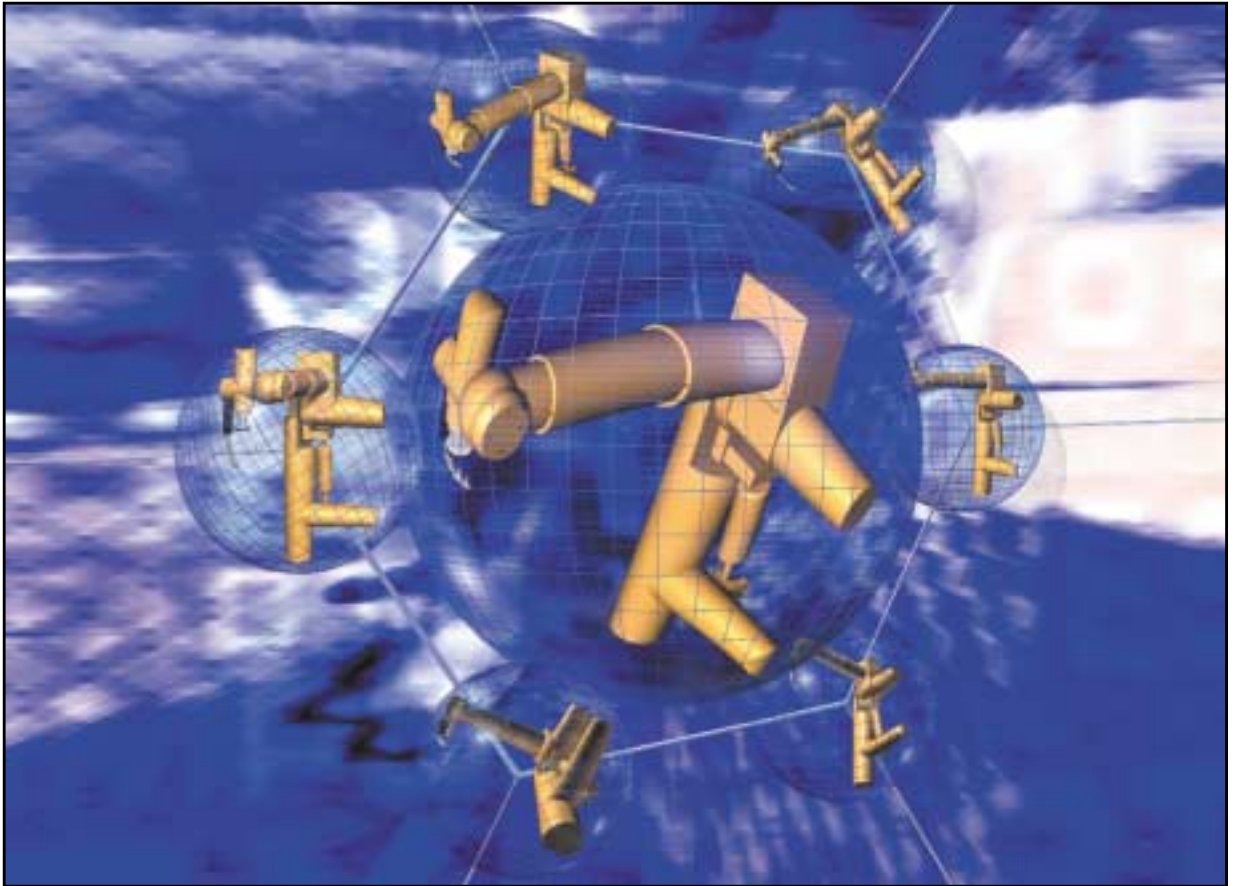


XMLTalk:

A framework for automatic GUI rendering from XML specs



User interfaces (UIs) are the most visible and, consequently, the most frequently changed part of any application. Tracking these changes makes client GUI development one of the most labor-intensive parts of system development and maintenance. This is compounded by the industry trend toward personalization, which drastically increases the complexity of GUI development as users may require different views based on preferences or role.

Most of these headaches of GUI development would be a thing of the past if the GUI could be defined by some kind of a specification instead of large amounts of code. This article introduces XMLTalk, an application framework that lets programmers specify the application's UI in an Extensible Markup Language (XML) file, instead of hundreds or potentially thousands of lines of Java code. The framework uses

key concepts from both the Java and Smalltalk¹ programming languages to aid in the automatic rendering of UIs using a set of predefined models and widgets.

XMLTalk applications follow the model-view-controller (MVC) paradigm, and both the models and the views are specified in the XML specification. In addition—and this is the key to XMLTalk—the connections between the views (widgets) and their models are also specified in the XML. This dramatically reduces the amount of code required to develop UI applications. The models used in XMLTalk are ValueModels, a concept originating in Visualworks' Smalltalk in the early 1980s. The framework contains a number of implementations of this very simple interface, namely ValueHolder, AspectAdaptor, SelectionInList, RangeAdaptor, and others. Complex application behavior can be achieved simply by connecting ValueModel objects

By Frank Sauer

together in various ways. These connections are also specified in the XML specification.

XMLTalk supports the entire range of Swing widgets. Views are laid out using layout managers, such as BorderLayout, GridBagLayout, and others. XMLTalk provides several enhancements to JTable, JTree, and other complex Swing widgets, such as striped tables, tree nodes with individual icons and support for drag and drop, input validation, and field formatting for both input fields and table editors.

Every application developed with XMLTalk becomes a component that can be used in another (or the same) XMLTalk application in the form of a window, dialog, inner frame, tab in a TabPanel, or simply an embedded Panel. For example, an address entry panel can be written once and reused in many applications. Applications provide the framework with an URL to their XML GUI specification, so the XML is not restricted to residing locally on the machine rendering the GUI, but could be obtained from a server where it is generated based on conditions, such as the user's security profile or personal preferences.

The XMLTalk framework contains three key packages: the ValueModel package that implements the various ValueModels; the widgets package, which specializes some of the more complex Swing widgets; and the XML UI builder, which reads the XML specifications and constructs the GUI from it. Applications using the framework extend the abstract ApplicationModel class of the framework and specify where the XML GUI specification can be found by implementing an abstract method defined in ApplicationModel (see Figure 1).

XMLTalk contains a Document Type Descriptor (DTD) that constrains valid content of the UI specifications. (The complete source code for this article is available at the code section of www.javareport.com.) The XML GUI builder reads the specification, optionally validates it, and builds the models, actions, and views contained in it. All elements in the specification are named and the builder constructs data structures where these elements can be found by name when they are later referenced from the XML. When the builder is constructing the widgets, it automatically connects them to the previously built models. Application programmers can also find models, actions, and widgets by using those same names in their code.

To get a better understanding of this process, let's examine a simple example application—the Java code, the XML specification, and the resulting UI.

XMLTalk Application Example

Consider the UI shown in Figure 1. All the sliders, progress indicators, and scrollbars move in a synchronized manner. When you move one, all of the others move with it. In addition, each of these components is constrained by a minimum and maximum value, which can be set by typing them in the input fields labeled "Min:" and "Max:". Typing a value in the input field labeled "Value:" makes all the sliders and scrollbars move to that position. The input field labeled "Extent:" controls the width of the scrollbar extents. (See Listing 1 for the Java code for this application.)

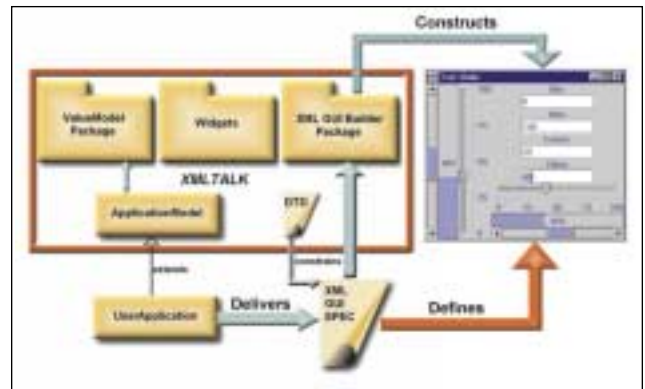


Figure 1. Overview of XMLTalk architecture.

An independently developed standard Swing version that implements the described behavior contains 230 lines of code and 20 instance variables. How can 10 lines of code in Listing 1 achieve the same functionality? The answer is ValueModels. To explain the concept of ValueModels in more detail, here is an excerpt from the XML specification for the application shown in Listing 1.

```
<MODELS>
<VALUEHOLDER name="min"/>
<VALUEHOLDER name="max"/>
<VALUEHOLDER name="extent"/>
<VALUEHOLDER name="value"/>
<RANGEADAPTOR name="range"
    extentholder="extent"
    maxholder="max"
    minholder="min"
    valueholder="value"/>
</MODELS>
```

This XML code snippet defines four ValueHolder objects

LISTING 1

The complete Java source code for the Slider example.

```
public class SliderExample
    extends ApplicationModel {

    public static void main(String argv[]) {
        (new SliderExample()).open();
    }

    public void windowClosed() {
        System.exit(1);
    }

    public URL getUI() {
        return
            getClass().getResource(
                "sliderexample.xml");
    }
}
```

and one RangeAdaptor object. The RangeAdaptor object refers to the ValueHolder objects by name. So, what are these ValueHolder and RangeAdaptor objects? They are implementations of an abstract class called ValueModel, which is an invisible Java bean with a single bound property of type Object named “value.”

This translates to a ValueModel being an object that has a `getValue()` method, a `setValue(Object)` method, and the important capability of firing a `PropertyChangeEvent` every time the `setValue(Object)` method is called. This firing of events when the value of a ValueModel changes is one of the most important factors that makes XMLTalk work. Recipients of the event implement the `PropertyChangeListener` defined in the standard Java JDK (1.1 and up). All ValueModel implementations and widgets defined in XMLTalk implement this interface, which allows them to receive these events.

This event propagation mechanism allows Value-Model objects to be connected together in various useful ways. In the code snippet on page 17, the RangeAdaptor is connected to four ValueHolder objects and is notified each time an object changes its value. On the other side, the ValueHolders are connected to the input fields, and the (single) RangeAdaptor is connected to all other widgets (sliders, progress bars, and scrollbars). These widgets refer to the ValueModels as their model. Because the widgets are connected to the models, they will also receive the `PropertyChangeEvents` as they occur.

Putting all this together enables the following scenario: A user types the value 40 in the input field labeled “Value:”. The input field will set the value in its model, the ValueHolder named “value.” In turn, the ValueHolder will fire the `PropertyChangeEvent`, indicating its value has changed. The RangeAdaptor receives the event and fires an event of its own to indicate its current value has changed. This event is received by all the sliders, progressbars, and scrollbars that make them move to position 40.

Figure 2 shows how all the models and widgets are connected. The blue arrows indicate the associations between the various models and widgets, while the red arrows show the flow of `PropertyChangeEvents` after the user types the value 40 in the value input field.

Listing 2 is the remainder of the XML UI specification, showing how the widgets are laid out, as well as how they refer to their models.

Java GUI programmers familiar with Swing will recognize most of the attributes in this XML specification as the usual properties one can set on the various Swing widgets. The `<CONSTRAINTS>` element in every widget specification defines the layout manager constraints that define the layout of each widget within its parent container. In this case, since the `<VIEW>` uses the `<GRIDBAGLAYOUT>` layout manager, the `<CONSTRAINTS>` element contains `<GRIDBAG>` constraints. Different layout managers will use different `<CONSTRAINTS>` elements, for example `<BORDER side=“North”>` can be used with a `<BORDER>` layout manager. In addition to the standard Swing attributes, XMLTalk gives each widget the `model` attribute, which links the widget to one of the models specified in the `<MODELS>`

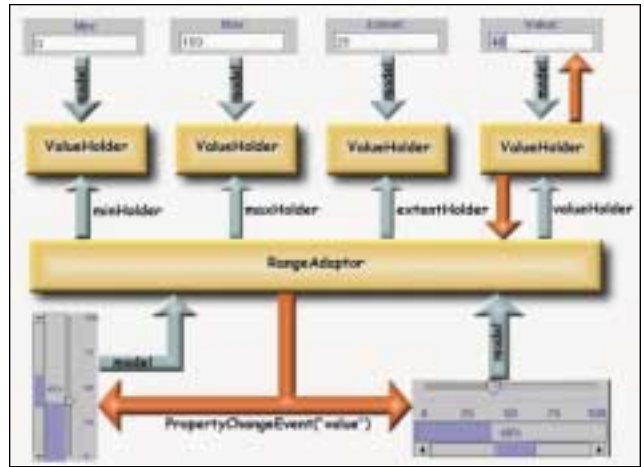


Figure 2. Models and connections for the Slider example.

section of the specification.

Listing 2 demonstrates the powerful concept of connecting very simple ValueModel objects together into more complex models that can drive the entire UI with the application, without requiring a single line of Java code. While developing several applications using XMLTalk, I have discovered several patterns in ValueModel connections that seem to occur in almost every application.

Connecting SelectionInList to Multiple BufferedAspectAdaptors

A frequent scenario occurs when a user makes a selection in a list of objects, triggering an update in multiple parts of the UI to reflect that selection. One example is a list of persons and a number of input fields displaying first and last names, and middle initial. The user can then edit these values and apply the change to the selected person by clicking an “Apply” button. Similarly, a “Reset” button should undo all the changes and retrieve the original values from the selected person. This entire scenario can be constructed using XMLTalk without writing a single line of Java code to drive this interaction. Let’s examine the models needed to implement this behavior in more detail.

SelectionInList. With Swing, the `JList`, `JComboBox`, `JTable`, and `JTree` widgets allow users to select an item within a list of items. With the exception of the `JTree`, all of these widgets in XMLTalk use the same model, a `SelectionInList` that aggregates two ValueModels to create a new one. One of the ValueModels—the `listHolder`—is used to contain the list of items, while the other—the `selectionHolder`—is used to contain the currently selected item. When users click an item in a `Listbox`, the `SelectionInList` will call `setValue` on its `selectionHolder`. As a result, the `selectionHolder` will fire a `PropertyChangeEvent`. The reverse is also true. When the application changes the value contained in the `selectionHolder`, the `Listbox` will react by changing the visible selection highlight. If the application changes the content of the `listHolder`, the `listbox` will react by completely repainting itself with the new content and resetting the `selectionHolder`. Note that the only programmat-

LISTING 2

View definitions for the Slider example

```

<VIEWS>
  <VIEW name="main" title="Test Slider">
    <LAYOUTMANAGER>
      <GRIDBAGLAYOUT/>
    </LAYOUTMANAGER>
    <SCROLLBAR name="verticalscroll"
      model="range"
      orientation="VERTICAL"
      blockincrement="25"
      unitincrement="5">
      <CONSTRAINTS>
        <GRIDBAG gridheight="REMAINDER"
          fill="VERTICAL"
          weighty="1.0"/>
      </CONSTRAINTS>
    </SCROLLBAR>
    <PROGRESSBAR name="verticalProgress"
      model="range"
      orientation="VERTICAL"
      paintstring="true">
      <CONSTRAINTS>
        <GRIDBAG gridheight="REMAINDER"
          fill="VERTICAL"
          weighty="1.0"/>
      </CONSTRAINTS>
    </PROGRESSBAR>
    <SLIDER name="slider_vert" model="range"
      orientation="VERTICAL"
      labels="true"
      ticks="true" track="true"
      major="25" minor="5" filled="true"
      snap="true">
      <CONSTRAINTS>
        <GRIDBAG fill="VERTICAL"
          weighty="1.0"
          gridheight="REMAINDER"/>
      </CONSTRAINTS>
    </SLIDER>
    <LABEL name="l1" text="Min:"/>
    <INPUTFIELD name="f1" model="min"
      decimalformat="###"
      type="Integer"/>
    <LABEL name="l2" text="Max:"/>
    <INPUTFIELD name="f2" model="max"
      decimalformat="###"
      type="Integer">
      <CONSTRAINTS>
        <GRIDBAG gridwidth="REMAINDER"/>
      </CONSTRAINTS>
    </INPUTFIELD>
    <LABEL name="l3" text="Extent:"/>
    <INPUTFIELD name="f3" model="extent"
      decimalformat="###"
      type="Integer"/>
    <LABEL name="l4" text="Value:"/>
    <INPUTFIELD name="f4" model="value"
      decimalformat="###"
      type="Integer">
      <CONSTRAINTS>
        <GRIDBAG gridwidth="REMAINDER"/>
      </CONSTRAINTS>
    </INPUTFIELD>
    <SLIDER name="slider_hor" model="range"
      orientation="HORIZONTAL"
      labels="true"
      ticks="true" track="true"
      major="25" minor="5" filled="true"
      snap="true">
      <CONSTRAINTS>
        <GRIDBAG gridwidth="REMAINDER"
          fill="HORIZONTAL"
          weightx="1.0"/>
      </CONSTRAINTS>
    </SLIDER>
    <PROGRESSBAR name="horizontalprogress"
      model="range"
      orientation="HORIZONTAL"
      paintstring="true">
      <CONSTRAINTS>
        <GRIDBAG gridwidth="REMAINDER"
          fill="HORIZONTAL"
          weightx="1.0"/>
      </CONSTRAINTS>
    </PROGRESSBAR>
    <SCROLLBAR name="horizontalscroll"
      model="range"
      orientation="HORIZONTAL"
      blockincrement="25"
      unitincrement="5">
      <CONSTRAINTS>
        <GRIDBAG gridwidth="REMAINDER"
          fill="HORIZONTAL"
          weightx="1.0"/>
      </CONSTRAINTS>
    </SCROLLBAR>
  </VIEW>
</VIEWS>

```

ic interaction is between the application and the models, using the simple and consistent ValueModel interface. The application programmer is not required to know and understand the API of each individual widget.

AspectAdaptor and BufferedAspectAdaptor. Sometimes a widget needs to display only *some* property of an object. This is accomplished by using an AspectAdaptor,² which is created with the name of a property (aspect). The

AspectAdaptor implements its `getValue()` method by using reflection to translate the name of the property to an appropriate `getProperty` method, following the standard Java beans naming conventions. For example, if the AspectAdaptor was created on aspect "firstName," it will call `getFirstName()` on its subject whenever the `getValue()` method is called. Similarly, it will call `setFirstName(object)` whenever the `setValue` method is called.

Often, a GUI will have several widgets that each display a different part (aspect) of the same subject. To facilitate this scenario, AspectAdaptor has a subjectChannel, which is another ValueModel containing the subject for the AspectAdaptor. Whenever the contents of the subject Channel change, the AspectAdaptor will retrieve the new value for its aspect (for example, by calling `getFirstName()` in the previous example). It then fires a `PropertyChangeEvent` to indicate its value has changed. By sharing the same subjectChannel among several AspectAdaptors, an entire GUI can be automatically updated whenever the content of the subjectChannel changes. In the above scenario, all the input fields showing `firstName`, `lastName`, and `middleInitial` will automatically update themselves when a new `Person` object is stored in the shared subjectChannel.

`BufferedAspectAdaptor` is a refinement of `AspectAdaptor`. Instead of directly updating the subject contained in the subjectChannel every time `setValue()` is called, it buffers the new value until it receives a `PropertyChangeEvent` from another ValueModel called the “triggerChannel.” When the `BufferedAspectAdaptor` receives the event from the triggerChannel, it gets its value, and when it is `Boolean.TRUE`, it propagates the buffered value to its subject. When the triggerChannel’s value is `Boolean.FALSE`, it retrieves the original value from the subject and signals to its views that the value has changed. This is used to undo a user’s changes. Multiple `BufferedAspectAdaptors` often share a single triggerChannel, which is triggered by an apply-button or cancelled by a reset-button. This allows multiple changes to be committed to the subject or undone in a single shot.

Putting It All Together

Now that we’ve seen the `SelectionInList` and `BufferedAspectAdaptor` models, let’s connect them to implement the scenario described earlier. We have a list of `Person` objects displayed in a Listbox, and every time the user selects one, we want the contents of the `firstName`, `lastName`, and `middleInitial` input fields to automatically update. In addition, we want to be able to replace the user’s changes with original values of the selected `Person` object. This scenario can be broken down as follows:

- Update the displayed values in input fields when a selection in a list changes.
- Commit or Abort all the changes made in the input fields in a single shot.

The first part indicates we want to use the `SelectionInList`’s `selectionHolder` as the `subjectChannel` of the input fields `AspectAdaptor` models. However, the second part indicates we want to use a `BufferedAspectAdaptor` with a shared `triggerChannel`, and not an `AspectAdaptor`. Since a `BufferedAspectAdaptor` is a subclass of `AspectAdaptor`, we can use a `BufferedAspectAdaptor` everywhere we use an `AspectAdaptor`, so combining both parts results in the following:

- A Listbox has a `SelectionInList` as its model.
- The `selectionHolder` of the `SelectionInList` is the `subjectChannel` for three `BufferedAspectAdaptors`.
- The `BufferedAspectAdaptors` share a single `triggerChannel`.

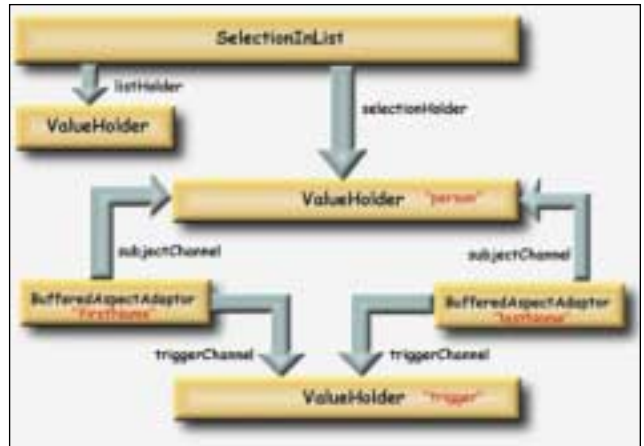


Figure 3. Connecting `SelectionInList` and multiple `BufferedAspectAdaptors`.

- The model for the three input fields are the `BufferedAspectAdaptors`.

Figure 3 illustrates the models and their connections, and Listing 3 shows how this is specified in the XML specification. The application code only has to provide the content for the listHolder of the `SelectionInList` (implement the `getPersons()` method). Note that Figure 3 only shows two of the three `BufferedAspectAdaptors` for clarity.

ValueModel Promotion

XMLTalk applications are reusable as components of other XMLTalk applications. Ideally, a component being reused should have no knowledge of the enclosing application or other components embedded in the enclosing application. But if components have no knowledge of each other, how then do they communicate with each other? To address this issue, XMLTalk introduces a new

LISTING 3

Example model connections

```

<MODELS>
  <VALUEHOLDER name="person"/>
  <VALUEHOLDER name="trigger"/>
  <BUFFEREDASPECTADAPTOR
    aspect="firstName"
    name="firstName"
    subjectChannel="person"
    triggerChannel="trigger"/>
  <BUFFEREDASPECTADAPTOR
    aspect="lastName"
    name="lastName"
    subjectChannel="person"
    triggerChannel="trigger"/>
  <BUFFEREDASPECTADAPTOR
    aspect="mi" name="mi"
    subjectChannel="person"
    triggerChannel="trigger"/>
  <SELECTIONINLIST name="persons"
    selectionHolder="person"
    list="getPersons"/>
</MODELS>
    
```

concept called “ValueModel Promotion.” Let’s examine ValueModel Promotion by looking at another example.

An XMLTalk application can choose to promote a ValueModel by specifying the promote=“true” attribute of the model specification. For example, a person editor might promote its ValueModel containing the Person object being edited as:

```
<VALUEHOLDER name = "person" promote="true"/>
```

XMLTalk uses this flag to publish the ValueModel to the enclosing application, making it a publicly accessible property of the embedded component. Some other embedded component might be used to select Person objects from a list or table of Person objects. It will promote its ValueModel containing the selected Person object as:

```
<VALUEHOLDER
  name = "selectedPerson"
  promote="true"/>
<SELECTIONINLIST
  name = "persons"
  selectionholder="selectedPerson"/>
```

At this point, the enclosing application has access to two promoted ValueModels. It can now use the delegatefor attribute to connect them together as follows:

```
<VALUEHOLDER
  name="globalPerson"
  delegatefor =
    "app1.person,app2.selectedPerson"/>
```

The value of the delegatefor attribute indicates that the ValueModel serves as a delegate for the promoted “person” ValueModel in the embedded application named “app1” and for the promoted ValueModel named “selectedPerson” in the embedded application named “app2.” The result of this connection is that the three ValueModels (“person,” “selectedPerson,” and “globalPerson”) will behave as if they are one and the same ValueModel object. When the value of one of them changes, the value of the other two will change automatically. This allows embedded components to communicate with each other without even knowing that the other one exists. Figure 4 illustrates the idea of ValueModel Promotion.

Conclusion

XMLTalk enables rapid development of UIs and allows these UIs to be modified without the need for code changes or recompilation of the application code. It promotes reuse of these applications by facilitating component-based development and loose coupling of these components. To use XMLTalk successfully, developers have to acquire a slightly different mindset. They have to think of

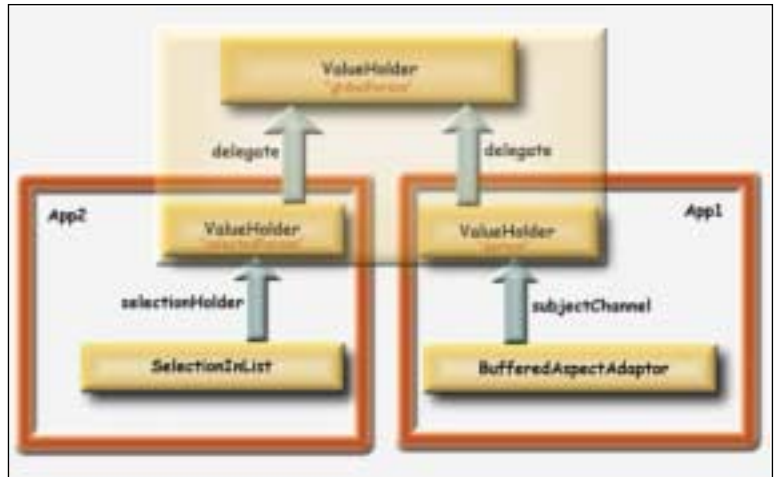


Figure 4. ValueModel Promotion.

UI development as “circuit design,” as if they are developing a digital circuit using off-the-shelf components. As in circuit design, the behavior of an XMLTalk application is “in the wires” in the way the models are connected to each other and to the UI widgets. With some training and proper documentation, non-programmers can maintain the UI of an application.

XMLTalk applications can be easily deployed to a large number of desktops, while still maintaining centralized control of the UI, if the XML specifications are maintained in a central repository. A change to the UI does not imply a complete redeployment to all the installed desktops, unless actual application code was changed. The bulk of code in an XMLTalk application is the XMLTalk framework itself, which will change very infrequently, if at all. The XMLTalk framework can be deployed once, while applications can be redeployed as needed. However, most changes to the application will consist of changes to the XML specification, which does not require a redeployment of the code at all. I have successfully deployed the XMLTalk framework as an extension to the JRE 1.3 and run XMLTalk applications as applets using the JRE as a plug-in to the browser. Because the framework is installed as an extension to the JRE, the applets themselves are extremely small. Often, entire applications are packaged in a jar of roughly 5KB, smaller than the average image on most Web sites. ■

Footnotes and Resources

1. Specifically, the Visualworks Smalltalk user interface frameworks originally conceived by ParcPlace.
2. The name “AspectAdaptor” is Smalltalk terminology. A more Java-centric name would have been “PropertyAdaptor.” The author is currently in the process of renaming most of the XMLTalk classes to be accessible to Java programmers.
3. For more information on XMLTalk, visit <http://www.trcinc.com>.

Frank Sauer is a infrastructure architect for the Technology Resource Connection, a wholly owned subsidiary of Perot Systems, in Tampa, FL. He can be contacted at frank.sauer@trcinc.com.