Open Issues by Kris Luyten and Jo Vermeulen

Following are two issues we worked out.

**How should UIML represent layout?**

Kris Luyten pointed out that the UIML 3.0 specification does not address graphical layout considerations. According to the language designers, this was intentional. Layout was considered a part of the vocabulary definition.

However, there is quite some difference in layout management support between different widget sets. Some only support absolute positioning (MFC/System.Windows.Forms) while others even allow the use of spatial constraints (Java Swing). Generalizing on layout managers is more complex than generalizing widgets (e.g. a Button, a Label, etc.).

In our opinion, it is better to separate the layout from the structure and style sections, in order to maximize the amount of reuse of a UIML interface. To illustrate the problem we will list two UIML documents that describe the same interface. One uses our GTK# vocabulary, while the other one uses our System.Windows.Forms (SWF) vocabulary.

Here is the code listing for the GTK# version:

```xml
<?xml version="1.0"?>
<uiml>
  <interface>
    <structure>
      <part class="Frame" id="Frame">
        <part class="HBox">
          <part class="Entry" id="leftentry"/>
          <part class="VBox">
            <part class="Button" id="copyleft"/>
            <part class="Button" id="copyright"/>
          </part>
        </part>
      </part>
    </structure>
    <style>
      <property part-name="Frame" name="label">Copy</property>
      <property part-name="copyleft" name="label">copy left</property>
      <property part-name="copyright" name="label">copy right</property>
      <property part-name="leftentry" name="text">          </property>
      <property part-name="rightentry" name="text">          </property>
    </style>
    <behavior>
      <rule>
        <condition>
          <event class="ButtonPressed" part-name="copyleft"/>
        </condition>
        <action>
          <property part-name="rightentry" name="text">
            <property part-name="leftentry" name="text"/>
          </property>
        </action>
      </rule>
    </behavior>
  </interface>
</uiml>
```
The actual rendered interface looks like this:

Illustration 1: Actual rendered interface with GTK#

This is the code listing for the SWF example:

```xml
<?xml version="1.0"?>
<uiml>
  <interface>
    <structure>
      <part class="Frame" id="Frame">
        <part class="Entry" id="leftentry"/>
        <part class="Button" id="copyleft"/>
        <part class="Button" id="copyright"/>
        <part class="Entry" id="rightentry"/>
      </part>
    </structure>
    <style>
      <!-- Absolute Positioning -->
      <property part-name="Frame" name="position">5,5</property>
      <property part-name="Frame" name="size">302,150</property>
      <property part-name="leftentry" name="position">5,50</property>
      <property part-name="leftentry" name="size">100,25</property>
      <property part-name="rightentry" name="position">195,50</property>
      <property part-name="rightentry" name="size">100,25</property>
      <property part-name="copyleft" name="position">125,25</property>
      <property part-name="copyleft" name="size">50,50</property>
      <property part-name="copyright" name="position">125,75</property>
      <property part-name="copyright" name="size">50,50</property>
      <!-- /Absolute Positioning -->
      <property part-name="Frame" name="label">Copy</property>
    </style>
  </interface>
</uiml>
```
And the actual rendered interface:

Illustration 2: Actual rendered interface with SWF

To highlight the differences between both UIML documents, we executed a diff on both files:
With a generic layout specification this UIML document would be completely reusable.

**Proposed Solution**

We extend UIML with a high-level way to describe the graphical layout. The designer is no longer bothered with widget-set dependent details. Our approach is based on the combination of spatial constraints and a constraint solving algorithm. The interface designer specifies the layout by defining constraints on the user interface components, such as `buttonA` left-of `labelB`. Afterwards, the constraint solver tries to find a solution that adheres to these constraints.

Constraints allow us to specify the layout in a declarative manner and integrate smoothly with UIML. The designer can focus only on what is the desired layout, rather than how this layout is to be achieved. Furthermore, constraints allow partial specification of the layout, which can be combined with other partial specifications in a predictable way. This is useful in our case to define the layout at several levels, taking advantage of an interface containment hierarchy. For example, we can define that container selection is left-of container content. The selection and content containers can then each on its own specify the layout of their children. When a change in this layout requires the containers to grow, shrink or move, the upper level layout constraints will be reevaluated.

More concrete, we added a `<layout>` tag, which can contain a set of constraints. Each constraint can have a priority, and contains a rule (which in our implementation is parsed...
by using a grammar) such as `<rule>label.top >= button.bottom</rule>` (meaning the label is positioned below the button) or a rule alias such as `<alias name="below">label,button</alias>`. Notice that `label.top` maps to the `top` property of the part with identifier `label`. The following example shows the copy interface re-implemented with our layout extension:

```xml
<?xml version="1.0"?>
<uiml>
  <interface>
    <structure>
      <part class="Container" id="hori">
        <part class="Entry" id="leftentry"/>
        <part class="Container" id="verti">
          <part class="Button" id="copyleft"/>
          <part class="Button" id="copyright"/>
        </part>
        <part class="Entry" id="rightentry"/>
      </part>
    </structure>
    <style>
      <property part-name="copyleft" name="label">copy left</property>
      <property part-name="copyright" name="label">copy right</property>
      <!-- set empty text for entries -->
      <property part-name="leftentry" name="text"></property>
      <property part-name="rightentry" name="text"></property>
    </style>
    <layout part-name="hori">
      <constraint><alias name="left-of">leftentry,verti</alias></constraint>
      <constraint><alias name="right-of">rightentry,verti</alias></constraint>
    </layout>
    <layout part-name="verti">
      <constraint><alias name="above">copyleft,copyright</alias></constraint>
    </layout>
    <behavior>
      <rule>
        <condition>
          <event class="ButtonPressed" part-name="copyleft"/>
        </condition>
        <action>
          <property part-name="rightentry" name="text">
            <property part-name="leftentry" name="text"/>
          </property>
        </action>
      </rule>
      <rule>
        <condition>
          <event class="ButtonPressed" part-name="copyright"/>
        </condition>
        <action>
          <property part-name="leftentry" name="text">
            <property part-name="rightentry" name="text"/>
          </property>
        </action>
      </rule>
    </behavior>
  </interface>
  <peers>
    <presentation base="swf-1.1.uiml"/>
  </peers>
</uiml>
```
Here is the resulting interface (rendered with the SWF renderer):

Illustration 4: The resulting interface with our layout extension

Of course, there are still some design issues to consider, but we wanted to keep this example as simple as possible. For example, there should be some space between the widget elements and they should be aligned properly.

That concludes the description of how we currently implement the layout specification in our UIML renderer. However, we still see some problems. For example, is it necessary to pollute the UIML interface element with the separate `<layout>` subelement? Another solution could be to map more closely with UIML's existing concepts:

```
<property part-name="hori" name="layout">
  <alias name="left-of">leftentry,verti</alias>
  <alias name="right-of">rightentry,verti</alias>
</property>
```

Here we treat the layout as another property. By putting all these properties in a separate template, we can simulate a layout tag. Of course layout will not be mapped on a physical property here, so it must be treated separately from other properties, which might be conflicting with the general idea of properties. It might confuse people who are new to the language, since we are making an exception here: there is a “special” layout property.

In our implementation we defined the aliases in the code. This is not ideal, since new aliases can not be added easily. Ideally we would need to have a part of the vocabulary that defines the different aliases. Here is a proposal:

```
<alias name="left-of">
  <d-param name="x"/>  
  <d-param name="y"/>
  <rule>x.right &lt;= y.left</rule>
</alias>
```

The `<d-param>` tags define the parameters that can be passed to the alias. They are then expanded in the rule.

We have also been investigating parametrized templates to allow a certain layout pattern to be applied to a part hierarchy. That is our take on the parametrization of templates, which is another open issue.
A layout template can be applied to a part hierarchy by inserting it as a child of a part hierarchy. Notice the class attribute defines the required type of interactor.

The part class Combo will be matched by the Choice from the layout template in the previous listing, since Combo is a possible instantiation for the Choice class according to the vocabulary that is used.

In general, we don't want to rewrite this new layout pattern when more than two parts have to be used. It should also be applicable to frames with three, four, or even twenty children. Therefore we need to be able to iterate through the parts to define constraints between them. We could use the xsl:for-each construction. However, we don't just want every part, we want to define constraints between two parts. In this example we want to define that every part is left of the next part. We haven't worked this out concretely yet, but it might be possible using the following-sibling axis. What about more complex layout managers, such as a FlowLayout? A flow layout allows the elements to move to the next “line”, if the horizontal space becomes too small.

There are also some general objections that can be made. Embedding spatial layout constraints in the UIML specification complicates its implementation. Renderers must include a constraint solver that is powerful enough to handle possible conflicts. Another objection could be that the layout mechanism we propose is restricted to the graphical layout of a user interface. It can therefore only be used with direct manipulation user interfaces. What about the notion of layout with a voice-based user interface? Our feeling is that a multi-modal layout specification still requires a lot of research, and cannot be completed within the provided time scope of the UIML Technical Committee.

**Affected Sections of the Specification**

5.3 - allow multiple layout elements within the interface element

5.4.2 - allow an inline layout to be added to a part, similar to the inline style element

7 – allow the template element to contain a layout element.
Of course a description of the layout element should also be added.

**Should rendering engines reference specific instances of external objects?**

It is sometimes desirable to have data preserved in the application logic so that the interface can be updated based on past user interaction with the system. Currently this is difficult to achieve as the logic element only specifies the class of object associated with a call element and not the specific instance. This implies that the objects are stateless or static and that the UIML rendering engine will instantiate the object each time the call is made. In order to overcome this, UIML may need a mechanism to specify which instance of an object will be referenced by a specific call. This would allow multiple instances of the same external object to exist within the system and be reference separately.

**Proposed Solution**

In our opinion it is not necessary to be able to reference each instance separately. This would complicate the application logic binding mechanism and the UIML language itself, since it would have to be augmented with a mechanism to uniquely identify each instance.

Our proposal allows arbitrary instances to be connected to a UIML document. When a call to the application logic is made, we first check if a call can be invoked on one of the connected object instances. When that fails, we’ll try to invoke it statically liked we did before.

Of course multiple instances can be connected to the UIML document. If we call a certain method, all connected objects will invoke this method. There's a catch though. When the call has to return a value, which return-value do we use? We suggest to use the return-value of the last connected object. Considering the fact that each invocation may have an effect on the other connected objects, this is the most sensible thing to do.

Following is a simple example with a modification of the UIML calculator interface. When you hit the '=' button, the contents of the output Entry is sent to the Write method of an instance of the CalcInspector class. Depending on a member "_times", this value is written out _times times:

```xml
<behavior>
  ...
  <rule>
    <condition>
      <event part-name="bsol" class="ButtonPressed"/>
    </condition>
    <action>
      <call name="CalcInspector.Print">
        <param>
          <property part-name="output" name="text"/>
        </param>
      </call>
    </action>
  </rule>
</behavior>
```
public class CalcInspector {
    int _times;

    public CalcInspector(int times) {
        _times = times;
    }

    /// <summary>
    /// Prints the output _times times.
    /// </summary>
    /// <param name="output"></param>
    public void Print(string output) {
        for (int i = 0; i < _times; i++) {
            Console.WriteLine(output);
        }
        Console.WriteLine("...Done!");
    }
}

The relevant code when the interface is rendered:

CalcInspector small = new CalcInspector(2);
CalcInspector large = new CalcInspector(5);

uimlDoc.Connect(small);
uimlDoc.Connect(large);

As you can see, we created two CalcInspector instances: small and large. One will write the output 2 times while the other will print it 5 times. Following are two screenshots, one just before pressing the '=' and one right after.
Since we connected small first, small's Write method gets executed first.

We also implemented a connection to other way around. It allows the application logic to subscribe to certain events from the UIML user interface. This was originally inspired by the connection mechanism of Glade¹. We wanted an easy and convenient way to connect with the rendered user interface. Uiml.net takes advantage of the .NET framework's basic support for aspect-oriented programming to realize this. Methods, classes, variables and the like can be augmented with so-called attributes². These attributes can then be queried using reflection. The instances are again connected in the same way as in the previous example.

The attribute UimlEventHandler specifies that a method can deal with user interface events. Parameters can be passed with the attribute that specify the information the method wants to receive together with the event.

```csharp
[UimlEventHandler("ButtonPressed")]
public void OnButtonPressed(Part sender, UimlEventArgs a)
{
    Console.WriteLine("Received event from a part:");
    Console.WriteLine("Apparently it's the "{0}" part", sender.Identifier);
    Console.WriteLine("It's class is <{0}>", sender.Class);
    Console.WriteLine("The UI object is of the type [{0}]", sender.UiObject.GetType());
}
```

¹ Glade is a User Interface Builder for GTK+ and GNOME. More info at [http://glade.gnome.org/](http://glade.gnome.org/)
² More information about attributes can be found at [http://msdn.microsoft.com/msdnmag/issues/05/05/BasicInstincts/](http://msdn.microsoft.com/msdnmag/issues/05/05/BasicInstincts/)
In this example, the method would receive every ButtonPressed event. It then writes out from which part it got the event. It writes out it's class, and the class of the actual underlying user interface object. So in fact the application logic queries a piece of the part tree, in which a lot of information is present. But we could go further:

```csharp
[UimlEventHandler("ButtonPressed", "fr", "l_recent")]
public void OnButtonPressed(Part sender, UimlEventArgs a)
{
    Part fr = a.GetPart("fr");
    Console.WriteLine("Part \"{0}\" of type <{1}> has" +
    "the following subparts:\", fr.Identifier, fr.Class);

    System.Collections.IEnumerator e = fr.GetSubParts();
    while (e.MoveNext())
    {
        Part sub = (Part)e.Current;
        Console.WriteLine("\t\"{0}\" of type <{1}>;", sub.Identifier, sub.Class);
    }

    Part l_recent = a.GetPart("l_recent");
    string propertyName = "text";
    Console.WriteLine("The [{0}] property of part \"{1}\" is: {2}",
    propertyName, l_recent.Identifier,
    l_recent.GetProperty(propertyName));
}
```

Since this is a lot of code, we'll go over it gradually. The first line:

```csharp
[UimlEventHandler("ButtonPressed", "fr", "l_recent")]
```

specifies that we are interested in ButtonPressed events. We also request the parts fr and l_recent from the user interface, so we can examine them. These parts will be collected in the UimlEventArgs parameter.

```csharp
Part fr = a.GetPart("fr");
```

This line gets the fr part from the event arguments. The rest of the code moves through all of fr's children, and writes them to the console. In a similar fashion, we request the l_recent part:

```csharp
Part l_recent = a.GetPart("l_recent");
string propertyName = "text";
Console.WriteLine("The [{0}] property of part \"{1}\" is: {2}",
    propertyName, l_recent.Identifier,
    l_recent.GetProperty(propertyName));
```

Here we write out l_recent's text property. So it is also possible to request properties
from a part, and query it's value. In fact, the whole Uiml.net API is available.

One might wonder, what if the application logic is only interested in certain events from a specific branch of the part tree? This is possible by adding an additional parameter to the connect method, specifying which part and its subchildren should be connected to:

```csharp
uimlDoc.Connect(this, "l_recent");
```

**Affected Sections of the Specification**

None, since all of this is implemented in the renderer itself. The language doesn't change at all.