UI Development Using XSL Beans

Table of Contents

Table of Contents **1**

Abstract **3**

UI Development and XSL Beans **4**

Discussion of Application Development Environments With A Focus On XSL Beans 4

GUI Development Environment Using XSL Beans **5**

Basic Development Environment Look and Feel 5
GUI Layout Models 7

Applications and Application Behavior **11**

What Is an Application Physically? 11
What Does an Application Do? 11
WWW Applications and Distributed Closures 11

Editing Application Behavior **13**

To Script, or Not to Script That is the Question. 13
Scripting, Functional Programming, and XML Instances 14
Editing the Code 16
Deploying defined code 18
Testing the application 19

Development Environment Extensibility **20**

Adding New Objects 20
Composing Existing Objects 21
Extending the Vocabulary Functional Composition 21
Extending the Development Environment 22

Development Environment Strategies **23**

Summary **24**

Glossary **25**

Bibliography **28**

Acknowledgements **29**

Table of Contents

Table of Contents

Abstract

GUI Development Environment Using XSL Beans

Basic Development Environment Look and Feel

GUI Layout Models

Applications and Application Behavior

What Is an Application Physically?

What Does an Application Do?

WWW Applications and Distributed Closures

Editing Application Behavior

To Script, or Not to Script That is the Question.

Scripting, Functional Programming, and XML Instances

Editing the Code

Deploying defined code

Testing the application

Development Environment Extensibility

Adding New Objects

Composing Existing Objects

Extending the Vocabulary Functional Composition

Extending the Development Environment

Development Environment Strategies

Summary

Glossary

Bibliography

Acknowledgements

Abstract

In a companion paper *'Platform Independent UI Objects'*, the author introduced XSL Beans, a technology that allows UI's to be instantiated using different display technologies, and different object systems. It is assumed that the reader is familiar with the XSL Bean concept.

The following is a summary of the advantages of the approach outlined in that paper:

- XSL Beans are platform independent.
- XSL Beans are standards based.
- XSL Beans allow documents to be brought to life as applications.
- XSL Beans exploit intelligent data, bringing new dimensions of reuse.
- XSL Beans are flexible.
- XSL Bean instances can be authored and understood by authors.
- XSL Bean are easy to use.

This paper discusses application development environments, and the background issues associated with them.

The intent of this paper is to provide a basis for discussion and development of an application development environment and in particular, one that is compatible with the XSL Bean approach to UI instantiation. Some possible deployment strategies are also discussed in later sections.

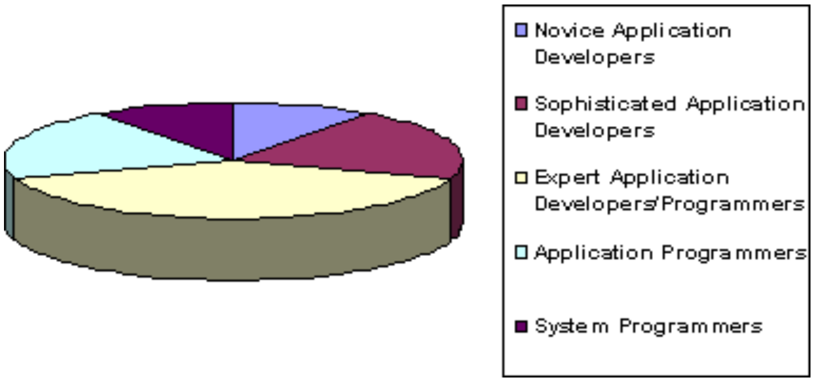
UI Development and XSL Beans

Discussion of Application Development Environments With A Focus On XSL Beans

GUI Development Environment Using XSL Beans

There are many different ways XSL Beans could be used within a development environment. This paper discusses, at a high level, how an application[1] development environment using XSL Beans might look and feel, and the issues associated with it. The discussion here is necessarily specific to an imagined implementation, but the concepts can be carried over to other implementations. In particular, the exact GUI depends very much on the layout and scripting paradigms chosen.

The primary goal for an XSL Bean development environment should be to support rapid application development or RAD, in much the same way that Visual Basic does: Visual Basic does not claim to be comprehensive, or necessarily the best tool, but for many cases, it is sufficient. Anecdotally, it provides 80% of the value for 20% of the complexity.

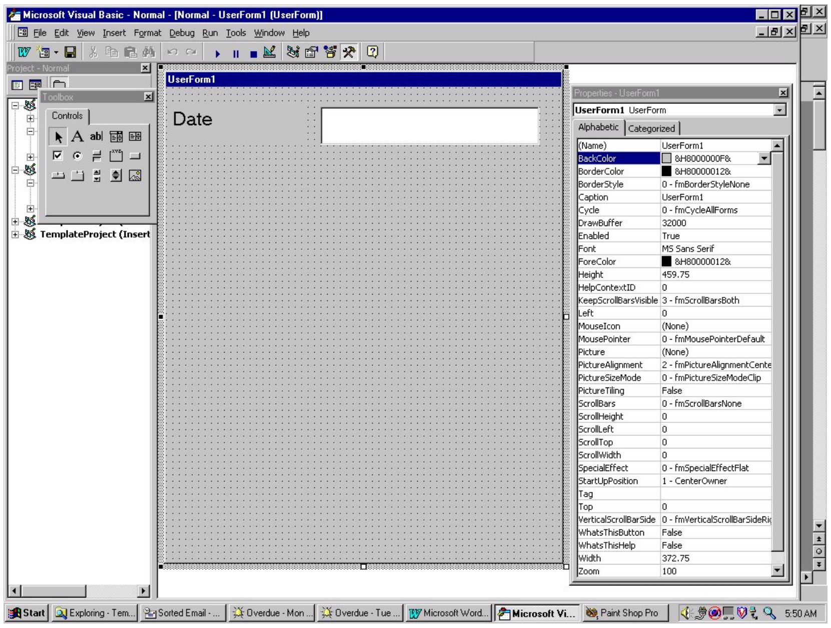


Target Users for UI Development Environment

An application development environment should strive for a balance similar to that shown above: it should be easy and intuitive to accomplish the tasks at hand for the majority of users, with a minority feeling constrained or overwhelmed (users at the edges of the bell curve).

Basic Development Environment Look and Feel

As stated above, an XSL Bean based UI development environment should strive to be roughly on par with Visual Basic. It is also reasonable to say that the basic paradigm for an XSL Bean development environment itself should be similar those for Visual Basic and JavaBeans. The Visual Basic for Applications development environment is shown below.



(Name)	UserForm1
BackColor	&H8000000F&
BorderColor	&H80000012&
BorderStyle	0 - fmBorderStyleNone
Caption	UserForm1
Cycle	0 - fmCycleAllForms
DrawBuffer	32000
Enabled	True
Font	MS Sans Serif
ForeColor	&H80000012&
Height	459.75
HelpContextID	0
KeepScrollBarsVisible	3 - fmScrollBarsBoth
Left	0
MouseIcon	(None)
MousePointer	0 - fmMousePointerDefault
Picture	(None)
PictureAlignment	2 - fmPictureAlignmentCente
PictureSizeMode	0 - fmPictureSizeModeClip
PictureTiling	False
ScrollBars	0 - fmScrollBarsNone
ScrollHeight	0
ScrollLeft	0
ScrollTop	0
ScrollWidth	0
SpecialEffect	0 - fmSpecialEffectFlat
StartUpPosition	1 - CenterOwner
Tag	
Top	0
VerticalScrollBarSide	0 - fmVerticalScrollBarSideRi
WhatsThisButton	False
WhatsThisHelp	False
Width	372.75
Zoom	100

Visual Basic for Applications Development Environment

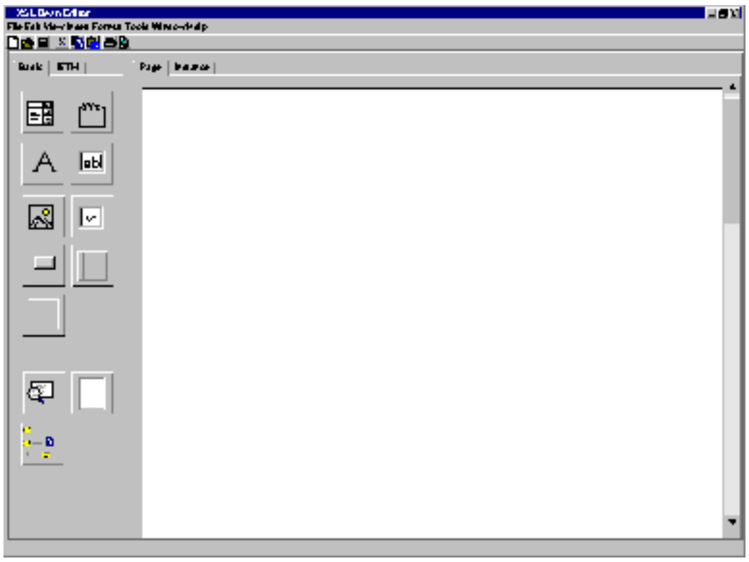
The main features of such an environment are:

- A palette of *predefined* objects, which a user can combine to form a UI. These are typically called controls in the MS Windows world, and widgets, or gadgets elsewhere.
- An edit area for composing a UI. This is typically called a form in the MS Windows world, and a pane, or panel elsewhere.
- The ability to drag and drop components from the palette to the form, enabling a form of WYSIWYG editing of the form.
- A property editor, which allows properties of the controls to be edited. Such environments are usually built on top of the ability to query the set of available properties at runtime.
- The ability to test the constructed UI, and debug it.

Most basic GUI−builder interactions with the components take place via property editing. Programmed blocks of code that respond to events (one can look at these blocks of code as properties of the object as well) handle behavior (such as calling out to ODBC to get the value for a property) that cannot be described by properties alone. This paradigm is widely accepted by developers as being powerful and easy to use.

It should be noted that in general, environments such as the above *focus on rapid development of the GUI layout*, rather than the application. This is a subtle and important distinction. This paper discusses application development in addition to GUI development.

Given the desire to emulate Visual Basic, a development environment using XSL Beans might look like the following (and all development GUI mock−ups will use this as a base from now on).



Example GUI for Web Page Layout Editor using XSL Beans

The GUI provides a palette of available XSL Beans (left) that can be used to layout a page (right) in a semi−WYSIWYG manner. XSL Beans can be dragged from the palette onto the page. Once instantiated on the page, double clicking produces a property list that can be edited. Setting some properties may involve interaction with a wizard.

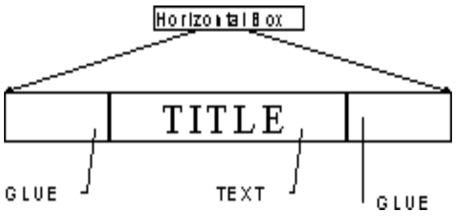
GUI Layout Models

Layout of the GUI objects is a fundamental action in application development. While it could be argued that a page−centric layout model is applicable to the WWW the paradigm espoused here is application−centric[2] as it is the author's (possibly mistaken) belief that Web *application* development is an overarching goal for many WWW developers. However, the issues involved with laying out GUI controls *are*> very similar to those in text formatting.

In Visual Basic, objects are placed in fixed positions on a fixed−size form. When a window is resized, or if the form is displayed at different resolutions, objects are generally scaled linearly (though programmers can override the default behavior). This results in forms that either look ungainly at certain sizes/resolutions (for example, toolbars that are oversized), or that require significant amounts of coding.

The model espoused here differs in that it dynamically reconfigures itself to handle arbitrary window sizes and resolutions. This is particularly suitable for the WWW, where screen resolution, available fonts, and locale can differ greatly between clients. This is also superior for I18N, because labels, menus, etc. will resize according to the size needed by a given language. The model is based on one of the most famous text formatters every produced: TeX, by Donald Knuth. It has been used with great success in the Fresco and InterViews GUI frameworks, and was the basis of the layout model used in Nemo, the prototype GUI builder for DynaWeb.

The basic model is that there is a top−level box, which can contain other boxes or glue that can stretch and shrink. Different combinations of this can result in different effects. For example:

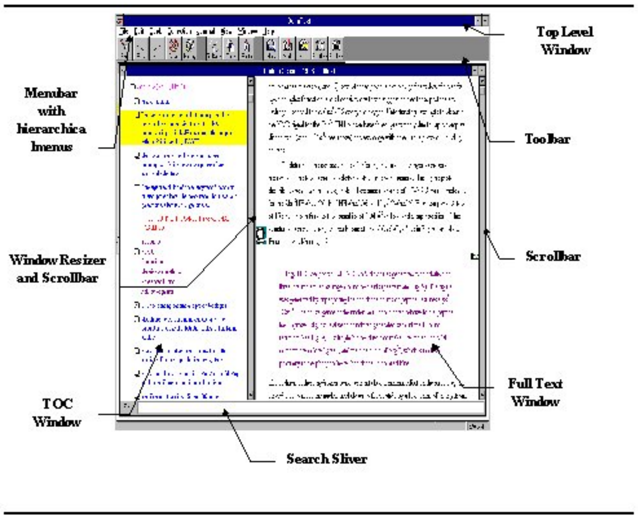


Sample of Box and Glue Layout

This is a horizontal box, containing three components, 2 pieces of glue, and one piece of text. If the pieces of glue are given an infinite desired size and an infinite amount of ability to stretch and shrink, the text component will be centered within the horizontal box, no matter what size the horizontal box becomes. This is often called constraints based layout, because constraints on sizes determine object dimensions. It is also often called the boxes and glue model.

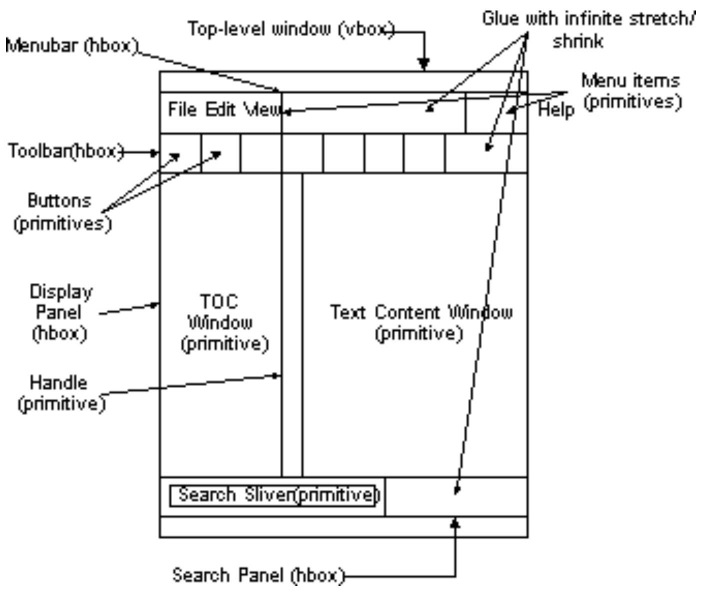
More complex controls can be built out of the combination of boxes, glue, and primitives, such as buttons. In InterViews, even primitives such as buttons were built from a number of finer grained objects, such as icons.

For example, a browser window such as the following



A Sample UI

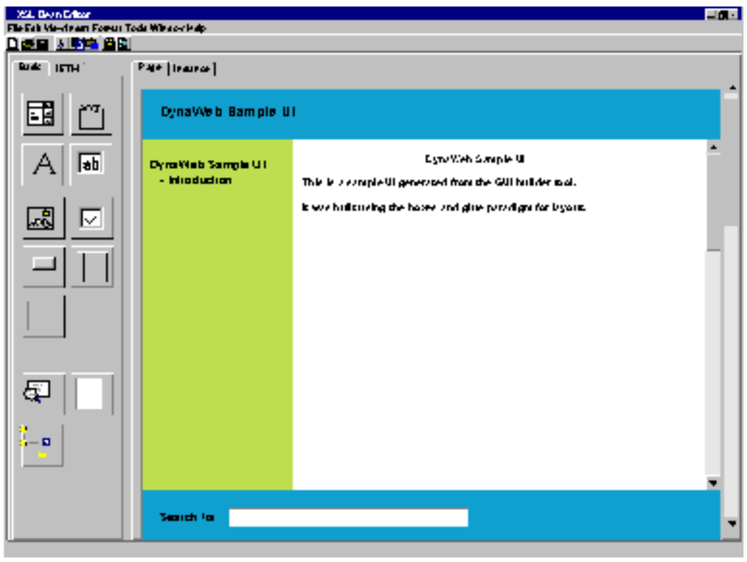
could be built using boxes and glue, as shown in the next example.



Example of Complex Boxes and Glue Layout for GUI

Obviously, because the boxes and glue model is purely hierarchical, it can *easily* be represented using XML, as a flow object tree (FOT), or as a hierarchy of JavaBeans (Swing already does something like this).

In a development environment, the boxes and glue model provides a simple and consistent model for development. In the Jupiter information delivery environment, most GUI's will not be anywhere near as complex as the preceding example. For example, the following is the GUI layout of DynaWeb, as it would appear within the example editor shown earlier.



DynaWeb Sample UI
- Introduction

DynaWeb Sample UI

This is a sample UI generated from the GUI builder tool.

It was built using the boxes and glue paradigm for layout.

Example GUI for Web Page Layout Editor using XSL Beans

This GUI could be created using the following steps:

1. Double click on the page to set the data source property, etc.
2. Drag a vertical box bean onto the page. It grows to fill the page.
3. Drag a label bean onto the page. It grows to fill the entire vbox.

    - ♦ Double click on the label, bringing up the property page. Set the height to 10%. Also alter the background color and the text content of the label.. The label resizes itself to fill the top 10% of the page.

4. Drag a search sliver bean onto the page. It grows to fill the entire remaining 90% of the page
    - ♦ Double click on the search sliver, bringing up the property page. Set the height to 10%. Also alter the background color. The search sliver resizes itself to fill 10% of the page, immediately after the label.
5. Drag a horizontal box bean onto the page in between the label and search sliver. It grows to fill the entire remaining 80% of the page, filling the middle region.
6. Drag a TOC bean into the hbox. This results in a wizard popping up, guiding the user through setting TOC properties such as look, elision, ranking algorithm, and transformer (the property list is retrieved from the object itself via introspection). The TOC fills the entire hbox.
    - ♦ Double click on the TOC bean, bringing up the property page. Set the width to 30%. Also alter the background color. The TOC resizes itself to fill the left 30% of the hbox.
7. Drag a content bean into the hbox, just after the TOC. This results in a wizard popping up, guiding the user through setting content properties such transformer (the property list is retrieved from the object itself via introspection). The content bean fills all remaining space. When the user first drags the content bean into place, and associates a data source, the raw XML view is displayed.
    - ♦ Click on the `stylesheet` property resulting in a (possibly empty) list of stylesheets, and the ability to edit them, including creating a new stylesheet from scratch. Editing takes place in the stylesheet editor (whose UI is consistent with the development environment).

Given a suitable set of XSL Beans with knowledge of one another, the user need do little more than this for most cases, because the objects will automatically synchronize themselves. For example, a standard set of objects that comprise a DynaWeb UI (TOC, content, search sliver) might be provided that knew how to look for, and update one another. This logic would be coded into the default action handlers supplied with the objects.

While this is sufficient for many simple tasks, it soon proves to be limiting. Some means of modifying the runtime behavior of an application is also needed.

Applications and Application Behavior

While simple layout and simple configuration changes can be easy to make, editing application behavior is more complicated. Before discussing behavioral editing in detail, it is useful to have some notion of what precisely an application *is*[3].

Like objects, an application can be split into two pieces: what it *is* physically, and what it *does* (this should not be surprising, because an application is an object in itself). Each of these can be analyzed separately:

What *Is* an Application Physically?

As with objects, an application can be represented by its state (though obviously the code is a necessary component too). This state is the set of objects and their properties and values *at a particular point in time*. This is proven by the fact that it is possible to serialize and then deserialize an application. The deserialized application is essentially identical to the application as it was serialized (at least for the split second at which the deserialization was complete, because timers etc. could alter application state almost immediately).

What *Does* an Application *Do*?

The behavioral aspects of an application can be seen as actions causing state transitions[4]. Not *all* aspects of behavior can easily be described this way however, because a state transition implies an atomic action, and in some cases, a sequence of actions needs to be performed, possibly with control flow. These can be described using single transitions, but such fine−grained descriptions of a system are very hard to create and comprehend.

WWW Applications and Distributed Closures

Given the above, a reasonable definition of an application is:

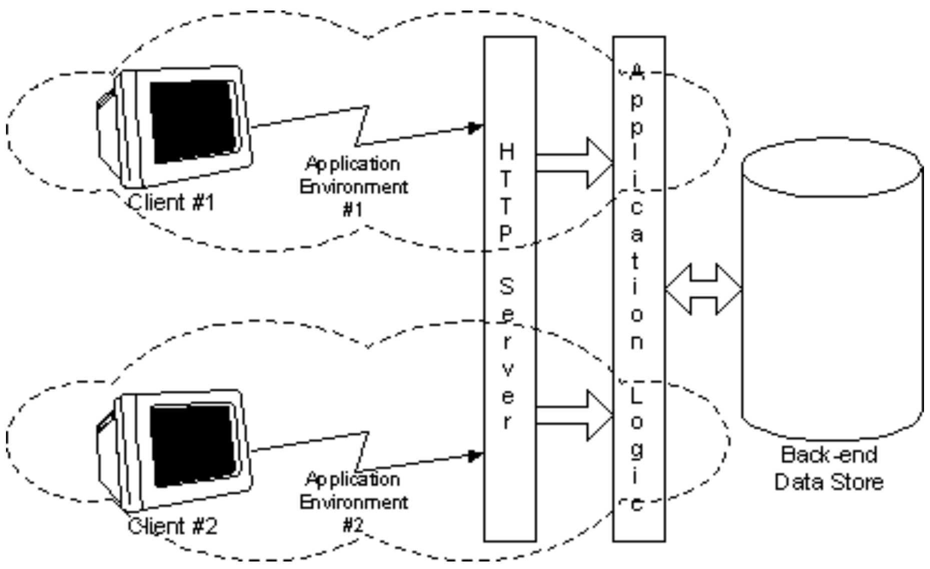
> An application consists of a network of objects making up the state of the application, and logic that results in changes to that state (state transitions).

The state of an application can also be looked upon as a closure, as commonly found in LISP systems.

The notion of a closure is foreign to many people, and indeed to many programming languages, but falls out naturally from the way LISP systems manage objects. In most LISP systems, there is an object called the environment. The environment grows and shrinks as procedures are called, because frames are added/removed during program execution to provide lexical scoping for parameters etc. (similar to the stack in other languages).

All active objects in the system are found in the environment or within objects in the environment (including procedures, numbers, constants, symbols, etc.). As such, a closure is essentially a snapshot of the environment at a given point in the execution of the application.

For WWW applications, and particularly those involving sessions, the notion of closure is extremely useful, because a closure *is* the Application State frozen at a point in time[5]. When multiple simultaneous clients are connected to a WWW server, it is effectively running multiple applications concurrently. Closures and environments provide a good conceptual (and possibly actual) way of managing application data, and maintaining the correct scoping relationships between applications, as shown in the following diagram.



Application Environments In The WWW

In effect, an environment spanning the client and the server[6] is being maintained, and each application environment is logically separate from any others (though some things in the environment might be shared, such as system−wide objects like logging services).

In cases where there is no explicit session (for example, no cookies are used to manage session id's), a closure is transmitted back and forth (i.e. the application state is held in the URL's, URL parameters, etc.). Clicking on a link in client results in an altered closure being sent to the server, which in turn returns a modified closure to the client[7].

In cases where sessions are used, deltas to the environment are transmitted back and forth such that the client and server, together, maintain the environment. For example, some local DHTML might alter the environment such that the color of a button is changed. At the point where the server next needs to know the button color, it needs the updated information. The exact way on which the environment is synchronized depends very much on the protocol and object system in use. For example, CORBA would allow the server−maintained Button State to be updated directly (effectively having the environment always live within the server address space), while HTTP does not.

The important thing is that any change to the environment is constrained to the environment *within that application*: so applications cannot interfere with one another. This is very important for reliability, security, and many other things. While environments can be simulated in CORBA, it is always possible for one application to interfere with another, because applications are scoped through the naming service. With environments, this cannot happen, because an application can only see what is in contained within its environment.

Editing Application Behavior

Given the above, the problem of editing application behavior becomes one of describing responses to state transitions (changes to the environment), that in turn cause other state transitions (changes to the environment). For example, when a button is pushed, the `pushed` property of the button changes from `FALSE` to `TRUE`. The task is to associate an action with the event[8] generated by the change.

Obviously, the objects and properties affected during a state change are those to be found in the environment of the application (within its closure), so within the development environment, it is easy to determine the objects that can be affected. This makes it relatively easy to provide functionality akin to the Visual Basic auto−completion feature, where a 'word wheel' of objects and properties pops up when a few characters are typed (this could be called an environment browser).

Having identified an object or property, there are only 3 actions that need to be performed directly:

- *Get* fetches the value of the object or property. This can happen either explicitly or implicitly as part of a comparison, etc.
- *Set* change the value of the property, resulting in an opportunity for another event handler to be triggered.
- *Call* call out to a procedure[9], providing an escape mechanism. This is not strictly necessary, because a call can be simulated by setting properties to pass arguments, and having setting or getting a property value to trigger some side−effect. For example, getting the name of a file can be seen as a call (`file_dialog.getFilename()`) or as a set followed by a get (`[get [set file_dialog.show TRUE]]`).

In addition to these primitives, some means of combining them is necessary (and naming the combinations), and some means for altering control flow, are necessary. The first of these tasks provides one axis upon which to extend the vocabulary for describing the application behavior (this is essentially adding verbs, the second axis is to add objects/nouns).

To Script, or Not to Script That is the Question.

One of the recurrent questions regarding application development is whether scripting is necessary, or desirable. From a manageability, maintainability and reliability perspective, scripting is certainly undesirable (witness the rising costs of WWW sites!), so a general guideline is that there should be no scripts.

However, from a purely practical perspective, some scripting is necessary, because it is very hard to comprehend large state transition networks, and very difficult to build design tools for them that scale well. Programming with transition networks is beyond the capabilities of most developers, who are used to procedural programming languages, or ad hoc scripting languages such as Perl.

In addition, there are cases where managing a set of actions as a unit allows greater modularity, and maintainability etc. Thus, the question is not *whether* scripting needs to be provided but rather, what *form* the scripting should take. A development environment must provide some form of scripting capability to be used in 'gluing' the application together that does not reduce the ability to manage and maintain the application.

Computer programs are difficult for both humans and computers to analyze well. Many procedural programming languages allow procedures to modify global state (global variables) which compounds the already difficult problem, effectively rendering analysis impossible. It is well know that global variables lead to bugs, and generally cause grief to all that encounter them. In order to support the goals given above, global variables should be avoided.

An older form of programming, called functional programming gives at least equivalent expressive power[10], but can be more readily analyzed by computers, and can be built with GUI's. Functional programs can also be represented meaningfully in XML (in other words, XML structure can meaningfully be used to represent the structure of a functional program), as explained in the next section[11].

Scripting, Functional Programming, and XML Instances

The following is a fragment of JavaScript and HTML that reacts to events on a selection menu, and then changes the background color of a document.

```
<SCRIPT LANGUAGE="JavaScript1.2">
  function setColor() {
    var choice;

    choice = document.colorForm.color.selectedIndex;
    switch(choice) {
      case 0: document.bgColor = "FF0000"; break;
      case 1: document.bgColor = "00FF00"; break;
      case 2: document.bgColor = "0000FF"; break;
    }
  }
</SCRIPT>

<SELECT NAME="color" onChange=setColor()>
  <OPTION VALUE="red">Red
  <OPTION VALUE="green">Green
  <OPTION VALUE="blue">Blue
</SELECT>
```

JavaScript Fragment for Background Color Selection

This uses the value of one property to decide the value of another, which is a very common pattern in GUI programming.

The important thing to note about the script here is that even though it is written using a procedural language, it is almost entirely declarative. For declarative statements, functional and procedural languages generally look very similar. For example, here is the equivalent script in Scheme:

```
<SCRIPT LANGUAGE="scheme">
  (define setcolor
     (let choice (document.colorForm.color.selectedIndex)
        (cond
           (eq choice 0) (define document.bgColor "FF0000")
           (eq choice 1) (define document.bgColor "00FF00")
           (eq choice 2) (define document.bgColor "0000FF"))))
</SCRIPT>
<SELECT NAME="color" onChange="(setcolor)">
  <OPTION VALUE="red">Red
  <OPTION VALUE="green">Green
  <OPTION VALUE="blue">Blue
</SELECT>
```

Scheme Fragment for Background Color Selection

Obviously, the syntax is different, but the control flow is identical, as is the gross structure of the script.

The object and behavior definition examples above actually reflect a poor design, because the script is defined outside the scope of the object using it. This complicates namespace management (what if a user wants to have 2 different `setcolor` procedures?), and also raises the question of *which* environment the script should be executed in[12].

The definition style shown above is almost entirely due to the models of HTML and JavaScript being brought together[13]. In more object−oriented systems, the `setcolor` function would be a method on the object. For example, in Java, this is almost certainly how it would be modeled. Another way of looking at this is that DHTML has all the MVC (model view controller) components in a single place.

An alternative representation of the above, written entirely in scheme, could be written:

```
(radio−button−group
  (define options '("Red" "Green" "Blue"))
  (define OnChange
    (lambda ()
      (cond
        (eq choice 0) (define document.bgColor "FF0000")
        (eq choice 1) (define document.bgColor "00FF00")
        (eq choice 2) (define document.bgColor "0000FF")))))
```

Alternate Scheme Fragment for Background Color Selection

Which is clearer[14], shorter, more maintainable (because of increased modularity), and does not have the scoping problem discussed above.

More importantly, this can *easily* be represented in XML:

```
<select type="radio−button'>
  <options>
    <option name="Red"/>
    <option name="Green"/>
       . . .
  </options>
  <method name="OnChange">
  <switch>
    <expr><get name="choice"/></expr>
    <case expr="1">
      <set name="document.bgColor" value="FF0000"/>
    </case>
    <case expr="2">
      <set name="document.bgColor"value="00FF00"/>
    </case>
     . . .
  </switch>
  </method>
</select>
```

Background Color Selection Using XML

This is somewhat verbose, but the important point is that *equivalent functionality can be achieved using an XML−based syntax*, and that in XML, the scoping of variable names (lexical scoping) is intuitive, because it reflects the hierarchy of the document tree. The variables that can be seen from within a given expression are those variables defined at some point within the ancestry of the expression.

The use of the `call` primitive to call functions is reasonable in such cases, because the scope of changes a function can make are limited to the scope of the environment in effect at the time the call was made. Another way of saying this is that a function (lambda) can be expanded inline (similar to macro expansion) with no change of behavior in the application. This point is a subtle, but very important one.

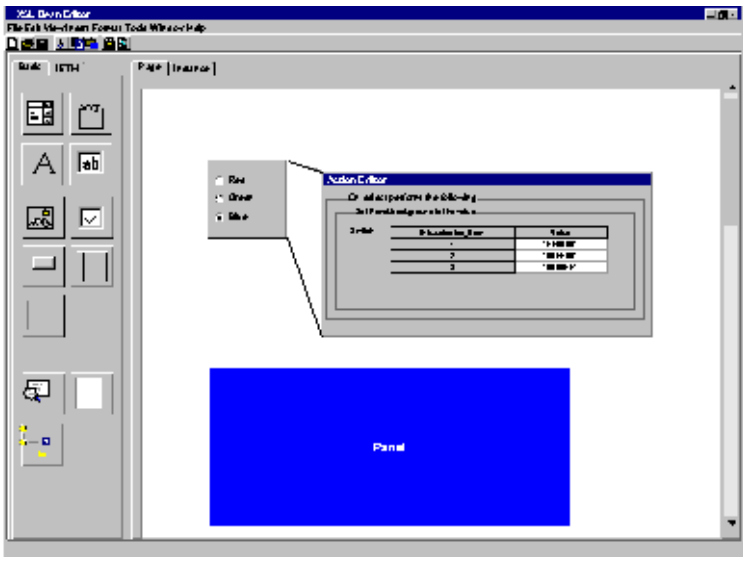
Note also that a minimal LISP implementation only includes about a dozen language constructs, and relies heavily upon functional composition, so a simple XML−based language could be made very small, but very powerful.

That said, it should be noted that pure functional programming is not a panacea: many programmers find it foreign and counterintuitive. It is the authors' contention however, that within the scope of UI programming, and UI object composition in particular, functional programming is not only desirable, but critical.

Editing the Code

In Visual Basic, and other similar environments, the development environment does little to aid in the task of actually writing code. Such environments will generally produce a procedure or function skeleton that the developer must flesh out, and debuggers to allow the programmer to easily debug the code they've written. Writing and debugging the code fragments is where most development effort occurs.

Obviously, code such as the above can be written using a text editor, but it is equally obvious that this is not an ideal environment for developers. An application development environment should provide tools that not only make it easy to produce GUI and event handler skeletons (as outlined earlier), but also offers tools for actually producing code as well. The following figure shows one potential UI used for editing the code above[15].



Example GUI for Action Editing

This figure shows a dialog box for editing the action associated with selecting something in a radio−button menu. The steps leading to this display might be something like:

1. Drag a `panel` object into the edit window, essentially creating a layout area similar to Visual Basic

(fixed rather than constraints−based layout). Set the background and other properties.
2. Drag a `selection` object onto the panel. A wizard guides the user through control type, list of items, and other parameters.
3. Drag a `panel` onto the panel object. Set the size and shape.
4. Double click on the selection object, bringing up its property list. One of the lists is of events that can be handled. Choose the `OnSelect` event.
    - ♦ A wizard pops up, prompting for the type of action to perform. The types of actions are `get`, `set` and `call`. The developer selects the `set` option, resulting in the wizard opening an environment browser, which allows the developer to select objects and properties to be set.
    - ♦ Having selected the properties to be set, the wizard now prompts for the value. The developer has the choice between various forms of literal expression, and compound expressions. The developer chooses the `switch` expression.
    - ♦ The switch expression wizard now comes into play, and prompts for an expression to be used as the switch value. The developer uses the environment browser to select the `selected_item` property of the current object. The wizard then prompts for pairs of match expressions, and return values, creating a grid.

This is a very simple example, but demonstrates most of the pertinent components:

- *Wizards* these guide the developer through the process of creating the switch and assignment.
- *Environment browser* which provides access to the environment available to the developer in the context of an object. Visual Basic provides something similar to this, but due to the possible use of global variables, it cannot provide *exact* information, and references to undefined variables can only be detected at runtime (one reason a debugger is needed).
- *Expressions* and their associated wizards. The main thing to note is that the power of the expressions comes not from the breadth of possible expression types, but the number of ways in which they can be combined, and the uniformity of the combination (essentially any expression can occur in any place a value could occur).

Notably absent from the example above is the ability to add user−defined functions and procedures through composition. This is discussed in detail in a later section.

For some developers, such an environment will prove limiting or confusing, because as expressions become more complex, the number of open expression wizards will conceal the overall control flow. This results in something akin to the 'lost in hyperspace' feeling, which is caused by too many open windows, or lack of link context.

As such, some ability to escape out of the wizards, into an editor, or development environment of choice is needed. For most systems, code written at this point is essentially a black box, and can no longer be edited, or analyzed by the development environment. A good development environment should not only support such code, but make it easy for the programmer to develop it. For example, rather than forcing the programmer to escape to an external application, the development environment could provide an 'expert mode' interface that allowed expressions to be typed in directly. This 'expert mode' should provide hints along the way, indicating possible functions to use, available variables, syntax errors, etc.

However, with the type of declarative language being discussed here, even external environments can be supported directly. The main difference between standard development environments using procedural languages, and what is discussed above, is in the handling of variables. Procedural languages generally allow variables to occur at a global level (as discussed earlier), and also allow references to variables to be passed to, and returned from procedures or functions. This leads to many situations where it is simply impossible to determine what effect changing a variable will have, because it is impossible to determine where in the application it is being used. In many cases, it is even possible for the exact same memory region to have multiple names (variable aliasing). These problems make it essentially impossible to analyze arbitrary code contributed from a developer.

In the type of language discussed above, arbitrary fragments of code can be contributed, because lexical scoping makes it impossible for the code to modify any variable not directly within the expression ancestry. As such, the development environment can parse and make sense out of contributed code because it knows all of it's dependencies.

Deploying defined code

Obviously, and XML−based, or even functional scripting language is useless if no widely deployed interpreters for it exist. An important point about the language discussed above is that the syntax is largely irrelevant: XML, scheme, or any number of other representations could have been used. In other words, it can *easily* be transformed into alternate syntaxes.

For example, given the following code snippet

```
<select type="radio−button'>
   <options>
   <option name="Red"/>
   <option name="Green"/>
   . . .
   </options>
   <method name="OnChange">
   <switch>
     <expr><get name="choice"/></expr>
     <case expr="1">
       <set name="document.bgColor"value="FF0000"/>
     </case>
     <case expr="2">
       <set name="document.bgColor"value="00FF00"/>
     </case>
     . . .
   </switch>
   </method>

</select>
```

Background Color Selection Using XML

It is obvious that it can be transformed into something like the following:

```
Function select As Integer
   Dim dialog as Dialog
   dialog = Selection("Red","Green","Blue")
   Switch dialog.select()
     Case 1: document.bgColor="#FF0000"
     Case 2: document.bgColor="#00FF00"
     Case 3: document.bgColor="#0000FF"
   End Switch
   select = dialog.selected()
End Function
```

Background Color Selection Using Basic

Even much more complicated expressions, even function declarations, can be converted to alternative syntaxes. This is entirely consistent with the XSL Beans approach to UI objects, which are platform independent. The above represents a platform independent scripting language.

Testing the application

Once the GUI has been built, and actions edited, the developer will need to test the application. In most environments, there is some way to switch from edit mode, to run mode. In run mode, the application executes as normal, though in most environments, the application executes within a debugging environment. An XSL Bean development environment also needs to minimally provide a run mode, and should also provide some form of debugger.

Note that with a functional language of the form outlined above, debugging is a much simpler task because application state changes are localized. As such, it becomes obvious where the bugs lie.

Perhaps the most complicated part of debugging applications such as those discussed in this paper, is that the events, and state transitions can happen in a distributed manner. Indeed events could be handled in any number of different environments (DHTML, Java Applets, CORBA objects, etc.). The two basic approaches to this problem are:

1. While debugging, constrain the number of possible environments, and tightly integrate with them. For example, a custom language interpreter might interpret all scripting, which would provide a very tightly bound and controllable debugging environment.
2. Provide as much of the deployment environment as possible, for example, a WWW browser might be integrated into the debugging environment, allowing a 'live' view of the application to be interacted with. The fidelity of the debugging environment here is almost purely a function of how closely the debugger can work with the WWW browser.

Development Environment Extensibility

Extensibility within a GUI builder, or development environment generally falls into four categories:

1. Adding new objects to the palette.
2. Composing existing objects into a new object.
3. Adding procedures/functions to the environment
4. Extending the development environment itself.

Each of these is discussed below.

Adding New Objects

The ability to add new objects is obviously desirable and indeed crucial to the success of the development environment: it is impossible to create a set of objects that will meet everyone's needs. The exact amount of direct support for *creating* new objects that should be provided is a difficult and important decision to make. Development environments such as JBuilder and Visual C++ provide wizards that guide people through the creation of a new object (either from scratch or as a derivation of an existing object). Such functionality should exist in an XSL Bean development environment as well, but adding more than just this quickly increases the complexity of the development environment.

In an XSL Bean based development environment, the primary task will be creating new XSL stylesheets, but at times, creating new flow objects might be necessary. For example, if a user wanted to create a variant of a TOC Bean that merged ODBC data and XML data together, one possible implementation strategy would be to write a flow object that did this. A sample XSL stylesheet for such cases might look like:

```
<xsl:rule pattern='h1'>
   <h1><xsl:process-children/></h1>
</xsl:rule>

<xsl:rule pattern='div'>
   <foo:odbc query='..'/>
</xsl:rule>
```

Sample XSL Stylesheet Using Custom Flow Objects

The important point in the above being that namespaces provide a perfectly reasonable way of partitioning the object space such that customizations do not conflict.

In cases where XSL functionality alone is insufficient, custom flow objects provide an 'escape hatch' to code, that is still well contained, and maintainable.

It is imagined that system programmers will deal with XSL Bean components (instances, stylesheets etc.) as discreet objects, and possibly directly implement custom flow objects to achieve their goals. Such programming can greatly benefit from an integrated development environment or IDE, but the low return on investment (in terms of productivity), and complexity of developing such IDE's makes it infeasible to either describe or implement a comprehensive solution for this. As such, a COTS product might be used: for example, if development takes place in Java, then JBuilder, Visual J++, or some such environment could be used.

Minimally, a development environment should make it easy to integrate with such third party tools. For example, a menu item might be provided that allowed a programmer to launch an application to develop a flow object (i.e. a flow object might have an associated application project), and/or provide wizards to guide through the initial skeleton creation phase.

Composing Existing Objects

The boxes and glue model espoused here leads to the obvious ability to compose objects and name them. From a developmental perspective, the main capabilities the development environment needs to provide are:

- The ability to add the object to the palette of objects, with a name, and an icon.
- The ability to coordinate property values for all objects within a composed object. For example: a user composes the centered text example (earlier) and saves it to the palette. When the user creates a new instance of this object, double clicks, and gets the property field editor, changing the background color should do so for all sub−objects as well (by default).
- The above is actually a subset of the larger problem of event handling, or event delegation. When some state transition occurs (mouse click, property change, etc.), it must be possible to control which objects handles events, and there must be reasonable defaults.

When composing a new object from existing objects, there needs to be the ability to create a composite object instance, or a template. A template is something that can be named, and later instantiated again (reused), whereas a composite object is instance specific. Ideally, a user should be able to save any composite object, as a template at will.

The development environment should alert the user of things that might make reuse difficult. For example, if all the objects within a single composite object had hard−coded, and different data source properties, instantiating a new object from the template would involve a great deal of work (i.e., the system should help users define templates in an object−oriented manner).

Extending the Vocabulary Functional Composition

As noted earlier, the power of a language is not really in it's breadth, but rather in the ways in which it can be extended, and the ways in which constructs can be combined. For example, even though English has well over 500,000 words in the language, only extraordinary people have a usable vocabulary exceeding a few thousand words.

As noted earlier, one axis for extension is the addition of first−class objects with properties (i.e. adding nouns to the language). The other axis for extension is through packaging combinations of expressions into a unit that can be referred to. This is often called functional composition. Expressions in an XML−based functional language could occur anywhere a literal value could occur, so combinations can easily be created, and given that a function call is another form of expression, functions cal also be used at arbitrary points.

In procedural languages, there is something akin to functional composition called procedures, or functions.

In an XML−based language[16] parameter scoping should be purely lexical in nature, making it fairly easy to write functions independent of their usage. A GUI could even be build for this, though again, a GUI environment will generally be constraining to more sophisticated users, so some escape mechanism would be needed.

Extending the Development Environment

While this is a desirable feature, it can be very difficult to provide a seamless way to extend the development environment. Perhaps the key thing here is to provide hooks for all areas of functionality the application development environment provides. For example, hooks that allow custom object inspectors to be 'plugged in', or that allowed some code to catch debugging events, would be extremely useful.

Any extension to the environment should require system programming, and as such, only minimal support needs to be given to it.

Development Environment Strategies

Basically, there are three approaches to providing a development environment:

- *Buy it* the development environment would be purchased from somewhere, and modified to suite. The probability of finding a suitable development environment for sale is small, and the probability that modifying it would prove cost effective is low.
- *Build it* obviously, a development environment could be built given suitable resources. The development environment discussed here is limited enough in scope that it would not be a large development effort, especially if implemented in Java. A number of freeware JavaBean editors exist today, so certain parts of development could be finished apace.
- *Integrate* here some JavaBeans that could be used in JBuilder, Visual J++ etc. would be developed. Those environments would be used to create an applet (i.e. perform the layout), entirely using Java. A tool that loaded up the applet, and then serialized it to XML would need to be provided. With special JavaBeans, greater control over the serialized format can be gained (especially if *all* Beans used were supplied), though XSL bindings for the serialized form would need to be provided. The greatest problem here is fidelity: it is hard to make an application development environment suit one's needs without total control of the environment.

The author's personal preference is to develop the environment.

Summary

This paper has looked at WWW applications and development environments for them. The main points can be summarized as follows:

- Development environments should provide a comfortable and productive environment for the majority of users, and provide some escape hatches for 'power users' so that they do not feel constrained by the environment.
- The basic paradigm for a development environment should be something akin to Visual Basic: basically the drag and drop, property editing paradigm.
- A convenient and powerful GUI layout paradigm for the WWW is that of boxes and glue, as found in TeX.
- Application behavior *can* be edited directly within a development environment, but doing this well depends on a functional scripting language.
- XML can be used as the syntax for defining a functional language.
- The notion of closure, and environments, is crucial to application development environments, and provides a good conceptual framework for managing WWW Application State
- The most likely development strategy is to develop the environment internally, though purchasing the rights to one is obviously preferable.

Glossary

Application development environment
: An environment used to develop applications. In particular, an environment integrated with an application server allowing WWW applications to be built.

Application
: An application consists of a network of objects making up the state of the application, and logic that results in changes to that state (state transitions). In the WWW, application also refers to the combination of client and server capabilities that allows users to interact with data (for example, group scheduling).

Application behavior
: The set of state transitions, and their effects, that occur as an application executes, and users interact with the application.

Application server
: A WWW server that also includes functionality allowing WWW−based applications to be developed and deployed.

Attributes
: In XML, name+value pairs on an element, and within CORBA and COM IDL, data members of an interface. Attributes are commonly mapped to get/set method pairs when IDL is used to generate language−specific bindings.

Clone (of an object)
: An exact copy of an object such that comparison of it's properties shows that they are all the same.

Closure
: A first class object representing the entire state of an application at a given point in time. Commonly found in LISP, but rarely in other languages.

Component
: A reusable software object.

Compound expression
: An expression that is made up of other expressions (possibly literal and compound).

Constraints−based layout
: A form of layout based not upon exact positioning, but rather on placing constraints upon objects. For example, an object might have the constraint 'at least 30% of the screen size' placed upon it. A common form of this is the boxes and glue model.

Continuation
: A closure and an anonymous function (akin to a function pointer) to be called when the continuation is executed. The function is executed in the environment packaged by the closure.

Deserialization
: The act of creating (instantiating) a new object from a serialized form.

Desktop Interface
: A UI that interacts directly with a desktop. For example, common word processing applications are all desktop applications.

Document−centric application
: An application whose paradigm is much the same as that of a document. OpenDoc was probably one of the early leaders in this arena, but the WWW brings it to the forefront.

Edit mode
: A mode in a development environment where objects do not react to events, making it possible for a developer to interact with the development environment to edit their layout etc.

Environment
: A data structure (possibly conceptual) that includes all named objects at a given point in the application execution. The exact contents of the environment change over time as variables come

into, and go out of, scope.

FOT
: Flow object tree, or tree of flow objects. This is essentially a tree of abstract objects representing objects to be 'flowed' into a page composition engine.

Functional composition
: The act of taking a number of expressions, or a sequence of steps, and packaging it with a name, thereby creating a function that can be invoked. Functions might also have arguments defined, which allows a function to 'hide' or 'alias' variables in the environment.

GUI
: Graphical User Interface. A User Interface exposed to the user using some graphical form.

GUI Development Environment
: An environment used to rapidly build GUI's. Typical examples are Visual Basic, Visual C++ etc.

IDL
: Interface Definition Language. Used in CORBA and COM to define interfaces for objects.

Instantiate (an object)
: To create (allocate) a new object in memory.

Interface
: The set of methods and attributes that an object must make publicly visible in order for it to 'implement' the interface. A single object might implement more than once interface.

Lexical scoping
: A feature of many programming languages is that variables are scoped to the block level (lexical scoping). In procedural languages, ways of working around lexical scoping exist.

Literal expression
: An expression comprised solely of a literal value, such as the number 1.

Look and feel
: The way a UI presents itself to a user. Most windowing systems have an associated look and feel style guide.

Object migration
: The ability for an object to move between processes.

Properties
: Within CORBA and COM IDL, data members of an interface. Properties are commonly mapped to get/set method pairs when IDL is used to generate language−specific bindings.

RAD Environment
: Rapid application development environment.

Run mode
: The mode in a development environment where the application is executing under the supervision of the development environment. Used mostly for testing.

Serialization
: The act of taking the set of properties comprising an objects state, and writing them out in a serial form.

State (application state)
: The set of properties that make up the environment of an application at a given point in time. See closure.

UI
: User Interface. That which controls the human interaction with a computer, including, but not limited to, the layout of the screen, and the behavior enforced by the application.

WWW interface
: An application interface that is delivered to, and executed within, a WWW browser.

XSL Bean
: An XML instance and a set of associated XSL stylesheets, that allow objects serialized into XML, to be instantiated in a variety of ways.

Bibliography

The following is a list of recommended reading only. Compiling a full bibliography for the ideas in this paper is an almost impossible task.

CORBA
    http://www.omg.org/
Java and JavaBeans
    http://www.javasoft.com/
Object Oriented System Development
    Dennis de Champeaux, Douglas Lea, and Penelope Faure
The Structure and Interpretation of Computer Programs
    Abelsen and Sussman
UI Development Environment Using XSL Beans
    Gavin Nicol
Why Functional Programming Matters
    John Hughes, 1984
XML 1.0
    http://www.w3.org/XML
XSL Draft Specification
    http://www.w3.org/XSL

Acknowledgements

The author wishes to gratefully acknowledge the early reviewers of this paper for their conscientious review and thoughtful comments. The reviewers, in no alphabetical order were:

- Chip Pettibone Product Manager, Information Delivery
- Cynthia Heitzman Product Manager, IETM Vertical Solutions
- Don Stinchfield Director of Engineering, Information Delivery
- Sebastian Holst VP of Product Management
- Scott Gordon Architect, Information Devlivery

---

[1] It has become common to call interactive WWW sites applications, and it is in this sense that the term is used here.

[2] The author feels that both have a place. Page−centric layout is crucial for producing acceptable print output, but application−centric layout is better for WWW pages. Within certain objects on a WWW page however, the page−centric model may still be applicable.

[3] A complete discussion of this is beyond the scope of this paper, so only a brief discussion is given. This may be the topic of a future paper.

[4] The topic of behavioral specification using transition networks as described in chapter 5 of 'Object−Oriented System Development) is very applicable here.

[5] This is not quite true. The true state of the application includes the state of the process itself. In some LISP systems, there is another first−class object called a continuation, which is a closure and an associated (lambda) function to be called to restart processing.

[6] The term environment was often used in this sense in DynaWeb 1.0.

[7] Actually, the server keeps sending an environment with continuations to the client. Clicking on a link effectively activates a continuation. It is not expected that readers will understand this.

[8] Note that the term event here differs from the commonly accepted usage, but encompasses all the functionality of it.

[9] This might be better called `apply`.

[10] An excellent paper discussing functional programming is 'Why Functional Programming Matters', by John Hughes.

[11] Note that first−order functional (applicative) languages are better.

[12] In some cases, global functions/procedures are necessary, but in UI programming, this is less common than is normal in programming, and generally restricted to simple functions.

[13] There has been some work by Microsoft within the W3C to standardize such practices under the label HTML Components. In some ways, these are similar to JavaBeans.

[14] The use of `lambda` may not be obvious. A lambda function is effectively an unnamed procedure, so in the example, we are assigning an unnamed procedure to `onChange`, and it will be executed when `onClick` is called. In many LISP systems, this is actually how procedures and functions are stored.

[15] Note that much of the UI discussed here could be used in other places. For example, many rules engines are essentially a functional language interpreter, and the UI discussed here is equally applicable.

[16] Indeed, this is desirable in most languages, and is part of most functional programming languages.