Platform Independent UI Objects

Table of Contents

Table of Contents

Abstract

XSL Beans are a unique technology for developing UI's that exploits XML and XSL, and that provides a number of benefits, chief of which are:

- *Media independence* XSL Beans are media independent, meaning that they can be used in Java, Dynamic HTML, or even standard GUI environments.
- *Standards−based* XSL Beans rely upon, and exploit, XML and related standards, providing an open UI framework.
- *Intelligent data* XSL Beans rely upon, and take advantage of, intelligent data. In particular, well−designed XML instances make it easy to bring sophisticated application behavior to documents in a transparent manner.
- *Documents as applications* using XSL Beans, *any* document can be brought to life as an application, bringing new dimensions of reuse to ostensibly static data.
- *Reusability* a direct effect of the above two points is that XSL Beans can be reused easily, in new ways. XSL Beans could lead to entirely new ways of approaching old problems, such as forms handling.
- *Instance syntax independent* a number of different XML instances can result in ostensibly the same UI, resulting in maximum flexibility in data representation.
- *Human legible, human editable* a direct result of the above is that XSL Bean instance data can be understood, and edited by people with little skill beyond that required for authoring XML documents. This brings application development within the realm of document authors (programming for the masses).
- *Ease of use* in combination with a UI builder, XSL Beans can be easy to use. More importantly, they can be easy to use even *without* a UI builder.
- *Flexibility* XSL Beans are entirely data−driven, meaning that components can change the data format, or the way the data format is interpreted, easily, at any time.

XML Beans could play a significant role in many areas, and could revolutionize the document−centric application arena.

This paper describes the thoughts leading to the XSL Bean concept, and the XSL Bean concept in detail. The primary purpose of this document is to provide a background to development groups that might be producing cross−platform UI's (in particular, WWW−based UI's) or UI builder applications that could benefit from XSL Bean technology.

The paper explains the XSL Bean concept at a fairly high level. There is a companion paper discussing development environments using XSL Beans, which also discusses some parts of XSL Beans in more detail.

.
.
.
.

Platform Independent UI Objects

Bringing XSL Beans to the World

Introduction

It is widely felt that having a comprehensive, state of the art UI is critical for the success of applications. Application UI's have many components, some of which are:

- *Administrative UI's* for most major projects, some administrative UI will be necessary. For example, web site management tools require extensive administration UI's for configuring the system for optimum performance.
- *Development UI* an information delivery component needs to be delivered with a development environment that allows people to rapidly prototype and deploy applications (analogous to application builders for WWW applications).
- *Application UI* the UI developed for the application in the development environment needs to be instantiated from within the application.

It is also widely felt that many UI's need to be available through both desktop and WWW interfaces. [1]

This paper outlines a platform independent UI technology based on XML and related standards[2]. The technology described is suitable for developing an application development environment that supports concurrent development of desktop and WWW−based applications, and also for deploying UI's developed in such an environment.

Architectural Requirements and Assumptions

The key requirements and assumptions leading to the espoused architecture are outlined below.

- *Low impact* given the amount of work already required for developing modern applications, the mechanism used for handling GUI's needs to minimize the impact on already overextended development resources.
- *Reusability* the UI manipulation framework should be usable in as many places as possible.
- *Media independence* given that many UI's need to be delivered both over the WWW, and as a desktop application, any architecture chosen should allow a *single* UI (or specification thereof) to be deployed to multiple environments. It is important to note that this is very much the same desire that drives people to use generic markup.
- *Consistency* many studies have shown that consistency in UI's is crucial for maximizing user productivity[3]. UI's within and developed an application, should have a consistent look and feel. Ideally, the look and feel should be selectable, and configurable[4].
- *Object systems* the UI framework should work with, and if possible exploit, COM, CORBA, and JAVA: it is almost certain that one of these three will be widely used in the environment an application is deployed in.
- *XML−related standards* given the prevalence of XML technologies, the UI framework should exploit XML standards as much as possible.

Components of a UI

The question of what exactly comprises a UI is central to the approach outlined in this paper. For the remainder of this paper, the following subjective definition is used.

A UI is 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.

Again, this is highly subjective, but the intent is to emphasize that a UI is far more than just a GUI, and includes desired patterns of activity and processing.

The key phrases in the above definition are *'...the layout of the screen'* and *'...the behavior'*. These phrases provide a basis for dividing a UI into two major components:

- *Structural/Visual* how the UI is presented.
- *Behavioral* how the UI acts.

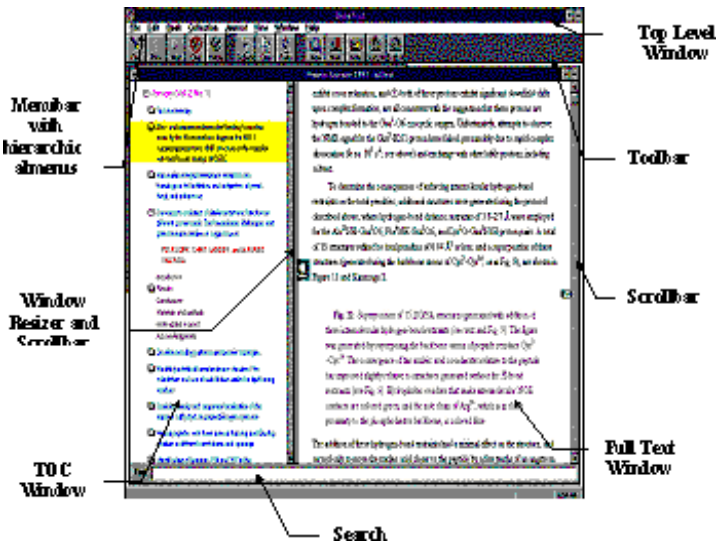
Both of these are described in more detail below.

The Structural/Visual Component

Perhaps the most obvious component of a UI is the way the UI is presented to the end user. For most applications, it includes menubars, toolbars, icons, and other such things, and is referred to as a GUI. On most windowing systems, there are guidelines to the overall GUI structure (often called look and feel guidelines).

The important thing to note about GUI's, is that they are intrinsically *hierarchical*, because there is an obvious containment relationship that can be reflected as a tree. In most modern GUI's, the windowing system supports the hierarchy directly, with windows that can be nested within one another[5].

Take the following example:



Example of a GUI

This GUI is made up of a number of display regions (windows) nested inside one another. Ignoring the

physical implementation of the GUI, one can come up with a *logical* description:

There is a top−level window that contains a menubar, followed by a toolbar, followed by two windows, side by side. One is a TOC window, and the other, a text window. Those two windows are separated by a bar used for changing the amount of space each takes up, and each has a scrollbar. A search sliver follows these two windows.

The fact that a succinct description of the GUI can be made so easily, and that the description uses purely declarative statements, is important to note, as this opens up a realm of possibilities.

The Behavioral Component

A very important part of UI's is how components interact with one another and with other parts of a system. This is often referred to as the behavior of the application.

At first, it would appear that there is no way of defining behavior other than by writing code, but there are four observations that can be made:

- Each UI component has a limited number of operations that are applicable to it, so the problem is somewhat constrained.
- Programming can be looked upon as a language creation task: each object function etc. a programmer creates is a new word that can be used for describing a problem and/or it's solution. By starting with a suitable predefined vocabulary of sufficient size, programming can be minimized (because there is no pressing need to add new words to the language).
- Each behavioral interaction can be regarded as causing a state transition in one or more components[6].
- The level of behavioral specification required amounts to something similar to either a *property change* or a *method invocation* (where a property change can be seen as calling a method for that purpose). Method calls will contain most of the actual processing, and will perform the actual state change (this implies that each UI component have a well−defined API, and that there be a way of invoking methods dynamically).

These observations lead to the conclusion that it is indeed possible to define application behavior in a purely declarative manner. This conclusion is certainly not intuitive for most programmers[7], which is why there are few environments that attempt to exploit this.

What is less intuitive is that any declarative statement can also be represented in XML.

It is reasonably well known that declarative programming is much faster, more reliable, and maintainable than procedural programs, so any environment that exploits the ability to declaratively define application behavior, is likely to reap significant benefits from it (or more correctly, people using the tool will reap the benefits). However, environments that provide such benefits by exploiting declarative behavioral definitions are scarce.

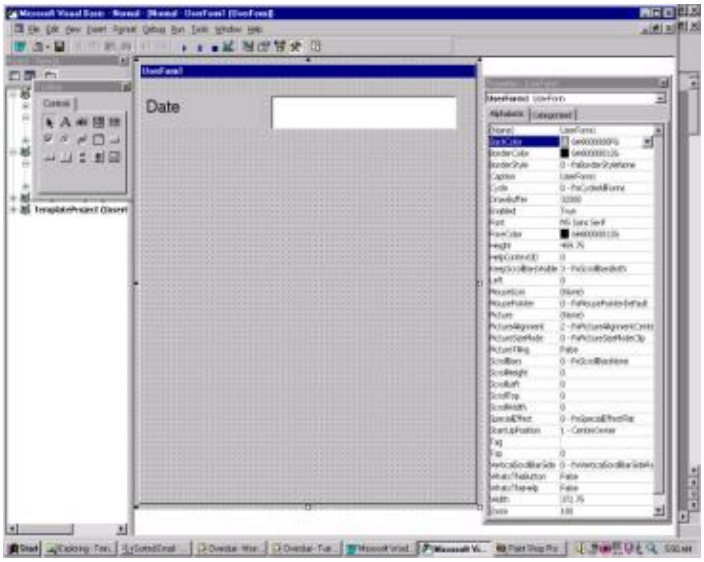
The WWW, where scripting prominence is rising at the same rate as site maintenance costs, represents a perfect opportunity for exploiting the above, especially if XML is used to represent the declarative definitions.

Java Beans, Visual Basic, CORBA and COM

Most modern RAD (rapid application development) environments are really GUI development environments, because they do little to help develop the application logic itself, and instead focus on GUI building and project management. Such evelopment environments generally provide the following capabilities:

- A palette of predefined components, and a drag−and−drop, property−editing paradigm for development.
- An interface for grouping files and other resources into a project.
- A way to generate skeletons for handling property changes, events etc., and a way for managing the code placed into the skeletons by developers.
- A debugging environment allowing programmers to debug their procedural programs.

A typical example (the Visual Basic Editor from MS Word) is shown in the following figure.



Visual Basic for Applications Development Environment

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. The most important aspect of this paradigm that there is a *predefined* set of components that each consists of a look and feel and a set of properties.

JavaBeans, one of the new and wildly successful Java technologies from Sun, is based on a convention for object interfaces. Like Visual Basic objects in the above example, a JavaBean is a component with a set of properties that can be manipulated. JavaBean technology standardizes how an application can get information about the properties (using introspection and special interfaces), and how properties can be manipulated (through get/set method pairs and property−specific editors). Programmed behavior is added either through action handlers, or in the implementation of the get/set methods.

JavaBeans were originally widely used for GUI objects, but recently, Enterprise Beans have gained wide

acceptance. An Enterprise Bean is simply a JavaBean that wraps up some part of a business in a bean interface (for example, a legacy database might be exposed as an Enterprise Bean). Enterprise Java beans aim at tackling many of the problems CORBA and DCOM are aimed at.

CORBA and COM are two object technologies that allow programmers to specify the interfaces[8] of a component, and provide a cross−language framework for manipulating objects via the defined interfaces[9]. They allow one to define both attributes (properties), and methods, using an Interface Definition Language, or IDL. These technologies only specify the *interface* of a component, not how it is implemented. They can be seen to complement JavaBeans, in that JavaBeans basically conform to a 'bean interface specification', which could be defined in either CORBA or COM. A JavaBean is more than just interfaces however: it is an object, and as such has code implementing the interfaces.

Under Windows, COM objects are used for much the same purposes as JavaBeans: COM provides a basic interface that objects must conform to, and provides a way of asking objects what other interfaces they support. This is put to good effect in many of Microsoft's object technologies such as ActiveX and OCX controls, where the normal way of interacting with the objects is through a GUI development tool. Such tools can use COM to discover the methods and attributes of an object at runtime, and hence can be written independent of the set of objects they will deal with (Bean editors provide the same capabilities for Java).

CORBA offers a superset of COM functionality in a platform and language independent environment.

All these technologies were (and are being) developed with the express purpose of providing a simple, flexible way of defining objects and their properties, and a way of manipulating the properties of instantiated objects[10]. All also provide some way of adding programmed behavior to objects.

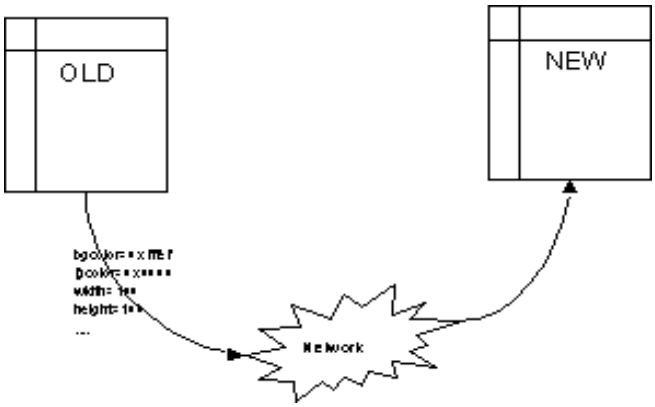
Serialization and Object Instantiation

The properties of an object represent its state. By retrieving the values of all properties, and creating a serial representation of them, an object is said to be serialized, and its state is saved. An object can be serialized to a file for persistant storage, to a network connection for object migration or communication, or to any other media desired.

An object can be deserialized, meaning that a new object is created (instantiated), and the serialized properties are used to set the values of the properties in the newly instantiated object. The newly created object is a clone of the original, and in some cases, both the original and the clone can coexist.

Note that this is explicitly different from object migration, where an object is serialized, but once deserialized, takes on the identity of the old object (the old object is destroyed). With object migration, it is an error for both the original and deserialized objects to exist in the same application scope, though with pure serialization and deserialization, this is not *necessarily* an error (in general it is, but in some cases, such as cut buffers, it is not).

Serialization and deserialization are depicted in the following figure.



Object Serialization and Deserialization

Note that only the properties of the object are being serialized, not the objects' code (though in a pure OOP system, the code would be a property of the object itself, allowing for true object migration).

The single most difficult thing about object serialization is that many objects have other objects as a property, so the entire object hierarchy needs to be serialized, and some ordering maintained so that deserialization can be performed in a stream−like manner. In some cases, the objects do not even form a hierarchy, and instead form a graph, further complicating serialization algorithms.

Serialization can be used for individual objects, or for an application, *whereby the entire state of an application can be saved and restored at some point in the future*. [11] This is an important point that has far−reaching ramifications.

Using XML for Serialization

An obvious application of XML in the above is as a replacement for the IDL (Interface Definition Language) used by CORBA and COM. This leads to an interesting form of literate programming for interface definitions, and has been put to good use in the W3C DOM, where the HTML documentation, CORBA IDL, and JAVA and ECMAscript bindings are generated from the XML source[12]. A number of efforts are underway, some within the OMG (the group managing CORBA development), to standardize an XML mapping for IDL, and for using XML in interface repositories.

A less immediately obvious[13] use of XML is as the serialization format used for data interchange between applications. RPC and various other data interchange efforts all have an "on the wire" format which is used for communication, with objects being serialized into and deserialized from it (usually this is called marshalling in this context). XML can be used for as the standard "on the wire" format (which is what WebMethods tools do).

For example, the logical description of a UI given earlier could very easily be rephrased (serialized) using XML, as the following example shows.

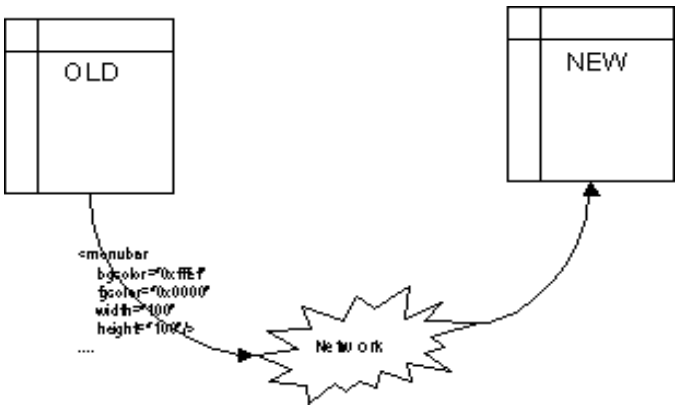
```
<UI>
  <MENUBAR>
     <MENU>
        <ITEM>Open</ITEM>
        . . .
     </MENU>
     . . .
```

```
  </MENUBAR>
  <SPLITPANE GEOM="30%,70%">
    <TOCVIEW/>
    <TEXTVIEW/>
  </SPLITPANE>
  <TEXTFIELD/>
</UI>
```

Logical UI Description In XML

One could just as easily serialize any object into XML: XML is rapidly gaining a foothold as part of a middleware layer between legacy enterprise information repositories (such as mainframe databases) and the WWW, which is the first step in this direction.

Toolkits for object serialization to XML are also beginning to appear (Coins and Koala being two). In such situations, the serialization picture changes slightly to become:



Serialization/deserialization using XML as the data format

This is obviously not a massive departure from existing practice (just a simple change in serialized data format), and so object serialization into XML should become commonplace over the next few years. This is especially true given the capabilities of JAVA, CORBA, and COM for introspection, which allows generic serialization engines to be developed.

However, when using XML simplistically as a serialization format, one of the following difficulties is likely to occur:

- *Hard coded serialization* in effect, each object would have a special constructor written for it that understands XML instance syntax (schema) specific to the internals of the object. Any change of property lists will require changing the serialization code. This is basically the approach that Coins takes.
- *Unintelligible XML instances* in order to develop a generic XML serialization engine, object names, property names, and XML instance syntax in general, will reflect the internals of the object being serialized. In effect, this will render it meaningless to humans, and difficult to reuse for anything other than deserialization.
- *Object system dependent* serialization as used above, tends to be object system dependent, again hindering reuse.
- *Media dependent* reflecting the object system dependence, the serialized data is also generally media dependent, making it hard to retarget.

These all basically reflect the fragility of simplistic serialization: small changes can break many parts of the system, and it can be very hard to recover from the mistakes, or to repair such serialized instances.

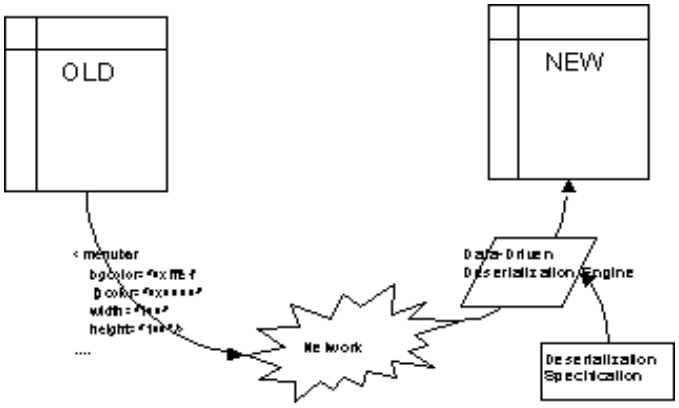
For example, a simple serialized EDI message might look like the following XML fragment.

```
<code−list>
 <desc>
   <date>Apr 09,1998</date>
   <time>08:00</time>
   <source>ISO 4217</source>
   <title>3 Character Country Codes</title>
   <usage>ASC X12 D.E. 100 − Currency Code</usage>
   <author>FORESIGHT Corp.</author>
 </desc>
 <codes>
   <code><name>ADP</name><defn>Andorran Peseta</defn></code>
   . . . . . .
   <code><name>AOK</name><defn>Angolan Kwanza</defn></code>
   . . . . . .
   <code><name>ZWD</name><defn>Zimbabwe Dollar</defn></code>
 </codes>
 </code−list>
```

As most people could point out, this is fine for certain cases, but is certainly not the most flexible because much of the information is left implicitly defined. Indeed, there are a number of ISO, and ad hoc working groups, whose sole purpose is to provide an XML syntax for EDI messages that is easy to process by humans *and* machines.

The Next Step

To get around the problems above, a degree of extensibility and intelligence can be introduced into the deserialization engine, in effect creating a data−driven deserialization engine.



Serialization and Data−driven Deserialization

By adding the extra level of intelligence (the data−driven deserialization engine and deserialization specification) the following benefits, among others, can be accrued:

- *Instance syntax independence* the format used for serialization can be anything at all, so long as a deserialization specification exists. Note that the deserialization specification is data, and so no code needs to change even in the face of radical changes to the serialization format. The deserialization

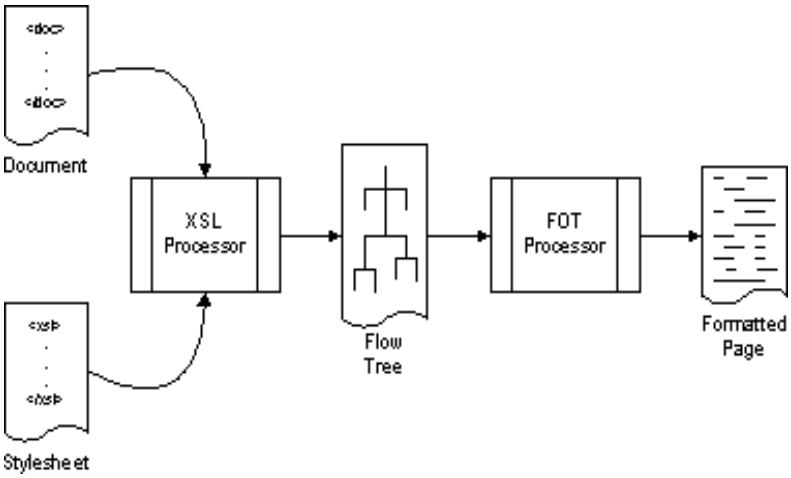
specification is necessarily tied to a serialization format, however.

- *Intelligible serialized data* a direct benefit of the above is that serialized instances can be intelligible to humans, and indeed could be documents, rather than serialized objects. As such, a document could be used to instantiate the UI used for interacting with it, based purely on the structure of the document itself (document as UI). This brings a new dimension of reusability to documents, and makes the notion of document−centric application literally true.
- *Object system independence* a given serialized instance can be instantiated using different object systems. For example, a serialized instance could be instantiated using CORBA, Java, or COM.
- *Media independence* a given serialized instance can be instantiated on vastly different media by using a different deserialization engine, different deserialization specifications, or a combination of both. For example, an object could be instantiated as a button in Java, a button in HTML (using DHTML), or as a voice selection for aural readers.

These benefits to object instantiation are basically the same as those that generic markup brings to the world of documents.

XSL/DSSSL Processing Models

The processing model for DSSSL, and for XSL, bears a striking resemblance to the data−driven deserialization model discussed above. In DSSSL/XSL, a document instance is combined with a stylesheet inside a DSSSL/XSL processor, producing a tree of flow objects (commonly called a flow object tree, or FOT). The flow object tree represents a tree of *logical* objects to be rendered that are independent of the actual rendered page. This is depicted in the following diagram:



DSSSL/XSL Processing Model

Each flow object is nothing more than a logical specification of an object instance, and it is the responsibility of the FOT processor to actually instantiate the corresponding objects. Different FOT processors can be used to achieve a degree of media independence: for example, James Clark's JADE, an implementation of a DSSSL subset, has FOT processors that produce HTML, RTF, TeX and MIF. Others have developed PDF and Postscript FOT processors for Jade.

A key insight with regards to XSL and the ideas espoused in this paper is that a flow object is really nothing more than a named object that can have children, and a set of properties. As such, *any object that conforms to such an interface can be considered a flow object*, meaning that almost any object can be a flow object, or instantiated as one[14]. For example, any JavaBean could be instantiated as a flow object, allowing a Swing UI to be instantiated as a flow object tree.

The main difference between the preceding diagram and that of a data−driven deserialization engine, is the intermediary FOT processor. If the objects output by the FOT processor, and the objects in the FOT are the same (i.e. if the FOT processor is an identity transform), then the FOT processor and its output are superfluous, and can be removed as an optimization (though the conceptual model remains).[15]

After removing the FOT processor the models are almost identical, and it becomes possible to instantiate arbitrary object hierarchies directly from within the XSL processor, rather than from the FOT generated by it (i.e. the FOT and the object hierarchy are one and the same). For example, a Java UI can be directly instantiated from an XML instance, an XSL stylesheet, and a generic XSL processor, as noted above. This optimization is possible, and desirable in two cases:

1. *Media specific processor* in cases where the XSL processor is limited to a single media (single FOT object set), and where the FOT needs no further processing. For example, the output of the XSL processor might be fixed for DHTML, HTML, Java, or perhaps XML.

2. *Dynamic object processor* in cases where the XSL processor can use an underlying object system, and XML namespaces, to instantiate arbitrary flow objects, and where the FOT needs no further processing. The earlier example of instantiating a UI is one good case in point (this is also one possible implementation of an intermediate FOT representation as well).

XSL Beans − a new approach to an old problem

To summarize the main points made this far:

1. XML can, and will be, used as a serialization format for objects and their properties. Koala, Coins, and the use of XML in middleware are clear indicators of this trend.
2. A GUI is intrinsically hierarchical, and can be serialized as an XML instance.
3. UI behavior can be defined declaratively, and can be represented in XML. This is discussed in more detail in the companion paper "UI Development Environment Using XSL Beans".
4. Simplistic serialization, and object specific serialization both have a place, but also have a number of problems:
    - *Hard coded serialization* any change of property lists will require changing the serialization code.
    - *Unintelligible XML instances* rendering it meaningless to humans, and difficult to reuse for anything other than deserialization.
    - *Object system dependent* serialization tends to be object system dependent, hindering reuse.
    - *Media dependent* reflecting the object system dependence, the serialized data is also generally media dependent.
5. Data−driven deserialization solves the problems with these approaches.
6. The processing model of XSL/DSSSL is almost identical to that of a data−driven deserialization engine.

Taking these all together, it is apparent that XML and XSL can represent the object instance data and deserialization specification respectively. These together are labeled an XSL Bean.

There is one important distinction between XSL Beans and other serialization and deserialization mechanisms (such as Java serialization): XSL Bean instance data does not necessarily have to be XML data generated from a serialized object, but can be *any* XML document instance. Indeed, it is the author's contention that in many cases, *the instance data will be authored directly by a human*, especially as common DTD's and corresponding deserialization specifications are developed.

Creating XSL Beans

As stated earlier, there are two parts to an XSL Bean: the XML document representing the instance data, and the XSL stylesheet specifying how the instance data is to be mapped onto a flow object tree (instantiated). Creating an XSL Bean involves (minimally) creating these two things:

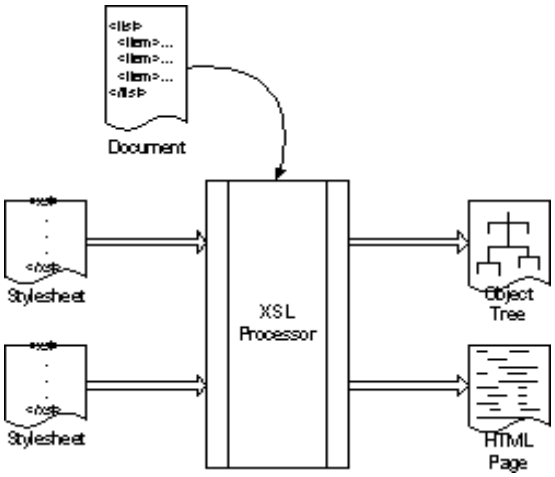
1. *XML instance* the instance can be created in a text editor, in an XML editor, as part of object serialization etc.
2. *XSL stylesheet* this too, could be created in a text editor[16]. However it is imagined that some form of XSL stylesheet editor would be used. More discussion of the environment is to be found in the companion paper "UI Development Using XSL Beans".

Instantiating XSL Beans

As discussed earlier, the process of instantiating an XSL bean is effectively that of a data−driven deserialization engine, or of an XSL processor with a (possibly) optimized FOT processor chain.

Object System/Media independence

A given XSL processor might be implemented such that it can instantiate arbitrary objects (very likely with COM, CORBA and/or Java), making it possible to instantiate an XSL Bean in a number of different ways. For example, in the following diagram, a `<list>` element is being instantiated as an object (for example, a `ListBox` JavaBean) or as a `<ul>` list in HTML.



XSL Bean Processing and Instantiation Model

It is equally possible to produce a high−quality formatted version for print, and in a web delivery component it should be possible to specify whether a UI should be sent to a browser as a set of applets, or in DHTML.

As such, an XSL Bean is best described as XML instance data with one or more associated XSL stylesheets for controlling the instantiated form. *The ability to support multiple media and looks and feels through style selection is a key differentiator*.

Deployment Strategies for XSL Beans

Deployment of XSL Beans falls into 3 major categories:

1. Providing an XSL Bean engine.
2. Providing a development environment.
3. Providing a standard set of XSL Beans.

Providing an XSL Bean Engine

Assuming an XSL processor written in Java, the creation of a generic XSL Bean engine capable of instantiating arbitrary objects is fairly trivial. In Java it is possible to instantiate classes by name: using XML namespaces, and possibly other hints in the stylesheet, the XSL processor can determine the name of the flow object it wishes to instantiate, and do so. The flow objects instantiated could literally be any Java object that conformed to the interface.

It is possible to do something very similar using CORBA as well.

The only problem with instantiating objects by name is that the operation is *slow*. For performance−sensitive applications (such as a stylesheet editor), it is probably unacceptable to incur this overhead; so application−specific instantiation code might be needed for an otherwise generic XSL engine (i.e. hard coding the FOT vocabulary).

Note that an XSL Bean engine could be used over and over again within applications. Almost all of the application GUI's could be built using an XSL Bean Engine, and they would be available as Java applets or applications, or as WWW−based GUI's. Forms based editing can be seen as a useful extension of the basic XSL Bean environment (in effect, this is distributed XSL Bean authoring).

Providing a Development Environment

XSL Beans development environments are discussed in detail elsewhere, but there are basically three choices :

1. *Buy it* a development environment could be purchased from somewhere, and modified to suite. The probability of finding a suitable development environment for sale is small, and even if one were found, the probability that tailoring it would prove cost effective is low.
2. *Build it* obviously, one could build a development environment given suitable resources.
3. *Integrate* here some application−specific JavaBeans that could be used in JBuilder, Visual J++ etc. would be written. Those environments would be used to create an applet (i.e. perform the layout), entirely using Java. A tool would be provided that loaded up the applet, and then serialized it to XML. When using application−specific JavaBeans, greater control over the serialized format can be gained (especially if *all* Beans used were from one place). XSL bindings for the serialized form would still need to be developed.

The author's personal preference is to develop a native XSL Bean environment.

Providing a Standard Set of XSL Beans

In both of the likely ways a development environment can be supplied (building or integration), JavaBeans would be used during the actual drag and drop layout editing. As such, the task of providing a standard set of XSL Beans can be broken into two major phases:

- Determine the set XSL Beans that need to be provided (ListBox, Menu, TOC, etc. etc.). This requires a substantial amount of analysis of the reference applications for the technology. The task of deciding exactly what set of XSL Beans needs to be supported belongs to product management, or the market.
- Provide JavaBeans and sets of XSL stylesheets that can be used to instantiate the XSL Beans. The main issue here is making sure that the beans work correctly in multiple environments (Java application, Java applet, DHTML).

Summary

This paper has given the background and introduction to a platform independent UI framework called XSL Beans. XSL Beans are a combination of an XML instance, and one or more XSL stylesheets, which are used to drive object instantiation. This leads to a number of benefits over existing frameworks:

- 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 Beans are easy to use.

XML Beans could play a significant role in many applications.

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, and to initialize it.

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
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 Delivery

---

[1] Here the term desktop interface describes an UI that runs on a desktop, such as MS Word, whereas a WWW interface is an interface running within a WWW browser, such as Netscape Navigator.

[2] Many of the ideas espoused here (and some parts of the text) are based on a paper "UI Description Using XML" written by the author in August of 1997. Those ideas, in turn, are based on ideas derived from work in the early 1990's on distributed hypermedia.

[3] Many studies have also shown that consistency of formatting in printed books results in speed and comprehension gains for readers.

[4] The Swing toolkit developed by JavaSoft allows one to choose the look and feel of a Java application at runtime. This is by no means a new idea: various X11 toolkits have had such capabilities, and LISP machines had the capability before them.

[5] MS Windows has a more complicated, and less intuitive model.

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

[7] Though there is a close relationship to functional programming.

[8] An interface is a set of methods and attributes/properties that an object exposes publicly. An interface differs from a class or object, in that interfaces are independent of implementations and class hierarchies.

[9] CORBA has always offered the ability for remote method invocation, and DCOM adds this capability to COM. As such, both COM and CORBA support distributed application architectures.

[10] It is interesting to note that most of the early OOP systems such as Smalltalk provided such capabilities as an intrinsic part of the object system, and so needed no special mechanisms.

[11] In most LISP dialects there is a first class object called a closure or continuation that effectively represents this.

[12] The author spearheaded this effort.

[13] The author has espoused this idea for a number of years as part of the "Document as UI" concept.

[14] Basically any object that can be constructed without needing data passed to the constructor can be

instantiated as a flow object, with some caveats.

[15] This may seem to be a bad thing to do, but a similar situation exists with I18N in HTML, where the logical transcoding to Unicode does not actually happen in most browsers.

[16] Note that there could be *multiple* XSL stylesheets per DTD.