XML for the Criminally Insane: an Introduction to the Schema-Based / Model-View-Controller Design Framework

Michael Corning, Microsoft Corporation

Introduction

This talk introduces schema-based programming to developers, and outlines the features of an application that demonstrates this new programming model. What follows this is a demonstration of such an application in action, in the form of an XML-based newspaper. Finally, Michael will walk through code for a polymorphic newspaper, and ebooks.

Michael Corning, mcorning@microsoft.com

Michael is a Memetic Engineer on the Application Server Team at Microsoft. Currently, his job is to automate the App Server test process using XML and ASP. Corning's fifteen minutes of fame came from being fortunate enough to have coauthored (with Steve Elfanbaum and David Melnick) the first book on ASP, Working with Active Server Pages (Que, 1997). Since then Corning has written extensively on topics of interest to ASP and XML developers. One European reviewer of his book aptly noted that "the author could scarcely contain his enthusiasm." Corning brings that patented passion to the podium before audiences the world over whenever he gets a chance to speak about software that, he believes, will have a measurable impact on the course of human development

SB/MVC Product Mandate

Introduction

Web development, like all previous forms of human enterprise, has experienced a division of labor. This process of stratification began with client/server and then three-tier programming. On the client the most significant hallmark of this process began when the data inside web pages was factored away from the presentation of that data. The best example of the effects of this segregation is a continuum from XML to XSLT to HTML. Along that continuum you will find specialists who can focus on a particular skillset, yet continue to work in concerted fashion. The result has been a workflow process in the industry that is highly efficient and equally profitable.

The advent of XML also heralded innovations in programming, itself. One of the more visible manifestations of this is the Microsoft BizTalk Server 2000, especially its Application Designer. At the other end of the spectrum is a grassroots effort known as "schema-based programming" (SBP). In both cases, the slogan is "The schema is the API." In schema-based programming, however, we have discovered that "The schemas are the APIs", but that's a different story! The application documented here is based on SBP, but it's also based on a very old programming paradigm, one dating all the way back to Smalltalk and the early 80's. That paradigm is called the Model-View-Controller design framework. The name of this hybrid design framework, then, is called SB/MVC, and its sample application is called *SBPNews*.

Rationale

The rest of this section will layout the rationale for developing SB/MVC in general, and *SBPNews* in particular. In a nutshell, SB/MVC is an attempt to apply the lessons learned from server-side and client-side Web development to the art of programming itself. That is, a well-staffed programming team consists in architects, developers, testers, and with sufficient resources, technical writers. SB/MVC is designed to maximize the division of labor without jeopardizing coordination. Benefits accrue to those who adopt SB/MVC on two fronts.

First, SB/MVC benefits from loosely coupled applications. This makes it possible to assign different functionalities to different teams without having to worry about real-time cooperation and coordination. The benefits here are similar to those produced by three-tier designs, only in SB/MVC labor is partitioned horizontally, not stratified vertically as in three-tier development. SB/MVC and three-tier architectures suggest a three-dimensional matrix organization. SB/MVC also shows some promise in making the code itself more compact and cogent (a neat trick whenever you can pull it off). Compact code is possible because SB/MVC aggressively leverages XSLT, a declarative programming language.

Second, SB/MVC benefits from tightly coupled documentation. The goal of SB/MVC is "change the spec and you change the code; change the code and you change the spec." This feature is what gives SB/MVC its cogency. In the traditional software development process the documentation library left behind consists of functional specs, program specs, test specs, and other documents such as user guides and FAQs. One problem with a traditional system of program development is that documents tend to die an early death, if they are written at all; a kind of "write once, use never." One notable exception to this is the team of programmers and testers in charge of NASA's Space Shuttle software. Documentation produced by that team is of heroic proportions and is meticulously cited and maintained. SB/MVC is a humble attempt to bring some of the advantages of careful documentation and testing to programmers who may not have the resources of world-class software teams.

One example of the power of tightly-coupled documentation is the original form of this document. As you read through it you will see actual source code appear in the text as appropriate. If I change that source code, the next time I open this document (the HTML

version), the code cited will be current. The original HTML document is to be found in the relevant code folder.

There are additional advantages in using SB/MVC. First, the MVC design framework itself has proven remarkably resilient; it began its life with Smalltalk, was generalized by the Gang of Four, implemented in Java, and now it provides the foundation for script-based web applications, including apps built with schema-based techniques. This suggests continued investment of intellectual capital in SB/MVC will continue to return dividend; for example, I'm working with the Microsoft Intentional Programming team to implement the SB/MVC design framework using their revolutionary technology.

To the extent parts of the SB/MVC design framework are permanent, you will not need to tweak that code the next time you implement a design with the SB/MVC framework. Likewise, other parts of the framework require little coding to use on subsequent programming engagements. I will cover the details of this feature later in this document.

Also you will see in a moment that once you move script into the Model making it data you can do all kinds of special things like creating tutorials or implementing code auditors. The sky is no longer the limit, just your imagination.

Finally, a significant advantage is that though no two SB/MVC applications will look alike in appearance, they will be virtually identical at the programming level. Nearly all the differences between any two SB/MVC apps will be in the data that drives the application. This makes sustained engineering much easier because two different people can create and maintain an SB/MVC application and the maintenance programmer won't have to learn how the creator wrote the app. This is one of the secrets of the Space Shuttle programmers: they reserve their creativity for the testing phase and they all make their original programs as simple and pedestrian as possible. It's more important that the code work than that it be a tour 'd force of programming style. SB/MVC just works, baby…

Criteria

So, we take it on faith that developers need and want software and systems that are:

- Efficient (requiring the least time for the highest quality in all aspects of development, testing, and documentation producing the highest return on investment for both client and developer)
- Effective (giving the client what they need on the first try, and being flexible enough to keep up with changing client needs and wants)
- Robust (including design frameworks that survive strategic inflection points such as the Web in general, and the .NET initiative from Microsoft in particular).

Let us now proceed to the functional spec.

Functional Specification

Introduction

The purpose of the *SBPNews* application is designed to show the essence of SB/MVC programming. *SBPNews* is the successor to the Model-View-Controller tutorial presented at the 1999 Wrox Web Developers Conferences in Washington D.C. and London. Both applications are true to McLuen's dictum of the modern age, "The medium is the message." That is, *SBPNews* is an example (including documentation) of how to build SB/MVC applications using SB/MVC techniques.

Functionality

This section outlines the basic mechanisms at work in an SB/MVC application. This section also documents how data flows throughout an SB/MVC design. Details are provided in the Program Specification below. The descriptions below are based on the *SBPNews* application, so will not cover all possibilities covered by industrial-strength SB/MVC applications. Keep in mind that the *SBPNews* application highlights the essential features that set SB/MVC designs apart from traditional MVC architectures and from conventional applications of procedural techniques.

SBPNews demonstrates the most basic of applications, but perhaps the most ubiquitous style of page seen on the Web. By default, *SBPNews* displays all headlines and story abstracts. Selection of a headline in either the headline list or the headline above an abstract displays the full text of the selected story. Once a story is selected, *SBPNews* also displays any reference data stored with the story. This reference data may include external and internal hyperlinks. In version 1.0 of *SBPNews* those hyperlinks are simple HTML anchor tags; later versions will use XLink techniques. Click either the Headlines heading or the Story heading to return the application to the start up state (where all abstracts are displayed and no headline is selected).

So, three mouse clicks are all that's needed to run this page, but don't let that simplicity fool you. There's lot's of very interesting things going on in those three clicks. Let's take a closer look.

SB/MVC Mechanics

Views

SBPNews has five views. If you have installed the XML utilities from MSDN you will be able to see the HTML source of *SBPNews*, and if you scroll down to the bottom of the source code you will see five HTML tags that include "view" at the beginning of their `id` attribute. These HTML tags get their HTML exclusively from an XSLT transform (described later).

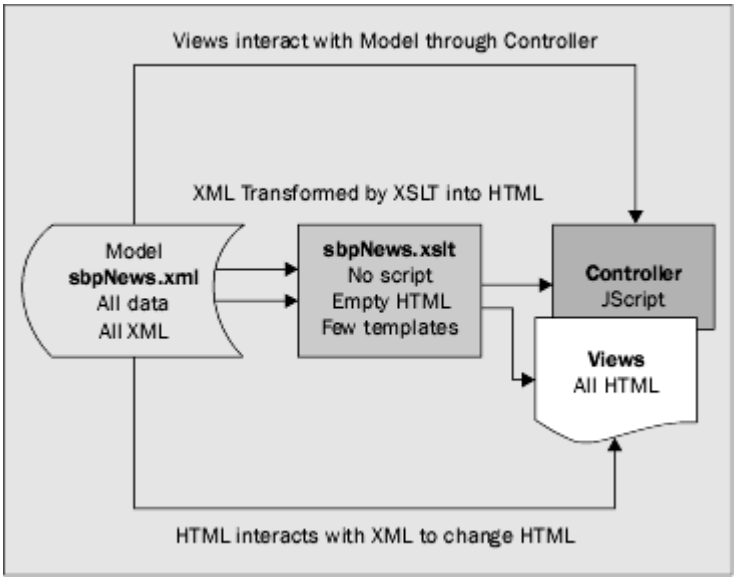
Controller

While you have the source code open, scroll up to the beginning of the script block. All the script in *SBPNews* is dedicated to the Controller function. The Controller processes interactions between the user and the app, and the Controller updates the application's data, which is found in the third part of an MVC scheme, the Model.

One of the innovations of SB/MVC is that Controller script is stored in XML wrappers inside the Model. In other words, *script becomes data*. This enables SB/MVC to do some very interesting and helpful things, as we shall see later.

Model

The Model is all data. The Model is all XML. As you just read, in an SB/MVC application the Model takes on extra duty not intended for conventional implementations of the Model, but extra duty that is perfectly consistent with the spirit of the MVC law. In SB/MVC the Model assumes responsibility for script, events, and XSLT that produces the HTML displayed by the Views.



This connects the Model back to the Controller. When a user selects a headline, the `headlineSelected` event fires in *SBPNews*. Inside the Model you will find XML nodes containing the events designed into the system. (Conventional MVC systems, on the other hand, have the Views register (and unregister) their interest in events. In this version of *SBPNews*, that "registration" is done at design time when the XML nodes enumerating events are entered by the programmer. Later versions will permit the Views to unregister themselves - a step sometimes necessary to ensure proper functioning of the entire application).

The event tags have a "notify" attribute, and the IDREF(S) string binds the event to one or more XSLT transforms that will manipulate the Model's data when the event (recorded in the "id" attribute) fires. See the Program Spec for additional details.

Next inside the Model you see those "views" (note the lower case) that wrap the XSLT transforms. These view tags have a `boundTo IDREF` attribute that connects them with the HTML View (note upper case) tags in the master `.XSLT` file that creates all the app's HTML.

The last point of interest in the Model is the application data, itself. Since the *SBPNews* applications is an electronic newspaper, the data is found in the `edition` tag. The `edition` tag contains `story` tags, and the `story` tags contain the tags whose data is (finally) displayed in the journal.

One final note before I summarize the feature set and move on to the Program Spec: there are only two files in an SB/MVC application, and both of them are XML files (even though one has an xslt suffix). Dynamic data is under the preview of the Model, the `.xml` file; and the `.xslt` file contains the static data, data that goes to work but once, at startup, data that renders the application. An SB/MVC app can query/modify either file at any time, should the designer so choose.

Features

SB/MVC is based on MVC, but it takes some liberties with the approach, and adds two important innovations. For those unfamiliar with MVC, the table below lists the four features of MVC first. Following them are the features of schema-based programming. Last in the table are the features of the hybrid design framework, SB/MVC.

Feature	***Description***	***Relevance***
Model	Single repository of application and external data (including program script, events and documentation).	Efficient
Views	Objects that display Model data.	Efficient, Effective
Controller	Script that mediates between the user and the application's Views (sensing mouse clicks, for example), and between the application's Views and Model.	Efficient, Effective
Loosely-coupled/Event-driven	MVC applications are composed of independent Views (that know nothing of each other) that respond to events raised when Model data (displayed by the View) changes.	Efficient, Effective
XML-based/data-driven	Schema-based programming, as the name suggests, is based on XML and related standards (such as XSLT and XPath). Schema-based programming applications tend to be aggressively data-driven.	Efficient
Declarative	XSLT is a declarative language driven largely by the data signatures of the XML upon which the XSLT transform operates. With XSLT, even the program is data.	Efficient
Polymorphic	SB/MVC applications are based on JScript to take advantage of function pointers used to create polymorphic functions in the Controller so different XSLT transforms can be bound to specific Views at runtime.	Efficient
XSLT-based Views	SB/MVC Controllers modify XSLT variables and fire XSLT transforms (in a one-to-many relationship with Views).	Efficient
XML-based Events	In traditional MVC applications Views register their interest in events with the Model. Functions in the View objects then transform the Model's data as necessary to properly represent the data to the user. In SB/MVC events, like almost everything else, are data and are bound to XSLT transforms at design time. This mechanism is simpler than dynamic registration and retains virtually all the power of traditional MVC designs.	Efficient
Controller notifies Views	This is the other place where SB/MVC departs slightly from conventional MVC designs (where the Model notifies the Views).	Efficient
Controller script stored in XML	Though SB/MVC must rely on a procedural language like JScript, it obtains some of the advantages of programs as XSLT by wrapping each function in XML and storing that function-cum-data with all the other XML data related to an SB/MVC application (including its documentation). Among the advantages of this technique listed below, SB/MVC documentation can use pointers to XML-wrapped functions always displaying the most up-to-date version of the code. Alternatively, since all code changes are "rev'd" with XML tags, documentation can reproduce the code at any given moment in its lifetime.	Efficient
Data stored in XML	Except for the HTML tags used by Views to render data, all other parts of an SB/MVC application are stored in XML, including the application's JScript. XSLT transforms used by Views to manipulate application data are also kept in XML nodes and act like little XSLT objects.	Efficient

Feature	*Description*	*Relevance*
Master XSLT transforms XML into HTML	Single external XSLT file moves the most current JScript from the XML file into HTML at runtime. JScript revisions (or "revs") are left behind in the XML and act like an audit trail of bug fixes and additional features. Each rev is date stamped so any previous version can be reconstituted. The XML that wraps JScript functions can provide test metadata for modeling purposes and can also include test case data using both to automate software testing.	Efficient
Separate CSS stylesheets control HTML presentation	Presentation work can be ongoing independent of programming and data management.	Effective, Effective
XML-wrapped functions will continue to work when C# replaces JScript	Next-generation programming languages based on COM+ 2.0 and the Universal Runtime have built-in documentation as a first-class feature of the language. In addition, APIs in C# can be exposed as XML by the compiler. SB/MVC is ready for such programming language innovations ensuring a long and productive life from the SB/MVC design framework.	Robust

I will now go into the programmatic details of each of these features.

Program Specification

Introduction

The *SBPNews* Program Specification lays out the specific design strategies used to build the application. It also justifies those decisions by weighing alternatives. The one thing that programmers who embrace MVC learn quickly (and not without pain) is that to code in the MVC design framework requires discipline. When I extended MVC to include schema-based programming techniques I had to grope to make progress; every time I pushed the envelope of MVC decorum I had to think hard about whether I was just being different or if the changes actually improved the framework. One of the hallmarks of a truly great framework is that it elicits the best thinking of its adherents, and it embraces that thinking, evolving along the way, if that thinking warrants such trust.

I can say with some confidence that the result of merging the nascent in schema-based programming with the permanent in MVC has made something better. With schema-based programming, MVC can do things it couldn't do before; and that's always been my best definition of an asset.

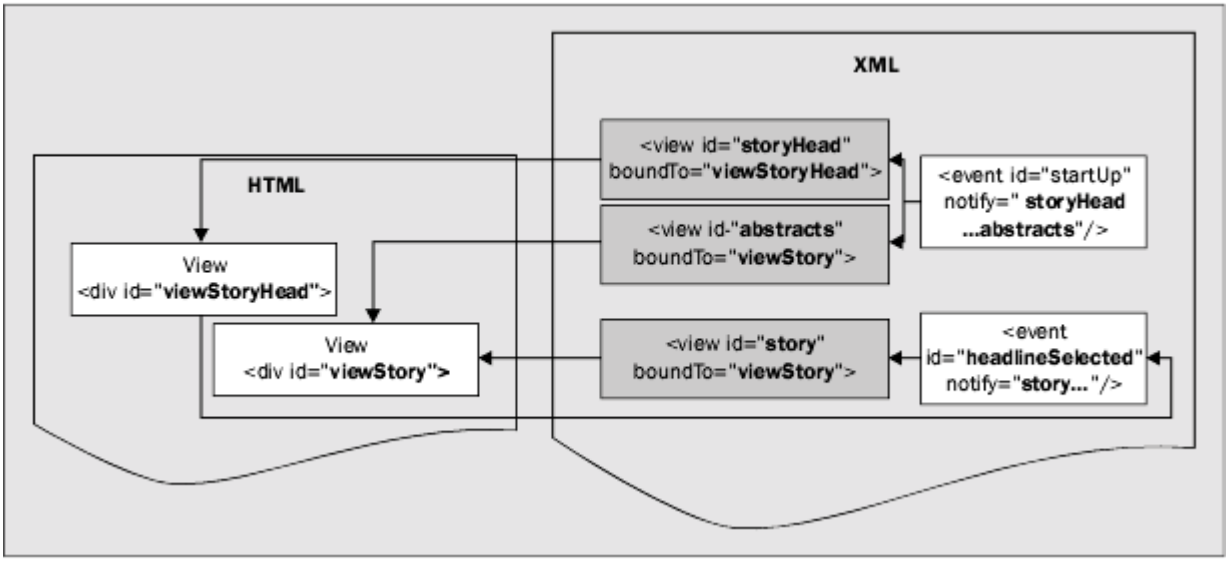
New Features

Let's begin with an example of the power that comes when you make script into data. Have you ever found an application's user interface doing something that's undocumented and wondered what code was responsible for the visual effects? In the case of *SBPNews*, let's say that the anomaly is that a View displays nothing. Since virtually everything in SB/MVC is data, the SB/MVC framework gives the programmer the ability to drill directly into the code for any selected View. In an SB/MVC app you can use XML virtually anywhere you want. For example, *SBPNews* also uses XML to create tutorials (displayed by clicking the *SBPNews* heading).

The remainder of the Program Spec will describe how an SB/MVC app works, in general.

The Thread That Binds

I will now describe in detail how SB/MVC binds Views to events to transforms. This simple design is extremely flexible. The next figure depicts two events. The first is when you start *SBPNews*. The second occurs when you select a story headline.



The `initialize()` function that is bound to the `window_onload` event fires the `startUp` event. The `startUp` event notifies the views (that hold the XSLT that transforms the XML into HTML) so that *SBPNews* displays something (confirm that nothing would be displayed by selecting the "View XSL Output" IE context menu - you will see empty HTML tags). For example, abstracts of each story along with its headline are listed down the middle of the web page. Follow the lines right to left, starting from the "start-up event box", to see how an event can notify several views; each view is responsible for one HTML tag (or HTML View).

If you select a story headline you fire the `headlineSelected` event. This event notifies the `story` view that, in turn, updates the same HTML tag that the `startUp` event updated. So even though each XML view is bound to only one HTML View, several XML views can share one HTML View (though only one XML view should update an HTML View at a time - otherwise there's probably a bug in your app).

These two examples serve to demonstrate the essence of the binding strategy. All other cases are a variation on this theme. Follow this model when you troubleshoot an SB/MVC app or when you want to add a View (as the *SBPNews* tutorial illustrates).

Code Overview

An SB/MVC app has three codeBlocks (see the `codeBlock` tag in the `SBPNews.xml` file). One is run once at start up, one is run every time a user clicks a View in the browser window, and the third codeBlock contains event handlers that run when their View is clicked.

The first codeBlock, named the "permanentBlock," contains the app's variable declarations and includes the `initialize()` function that runs at `window_onload`. The second codeBlock holds the `notify()` function and a handful of helper functions that the SB/MVC programmer should never have to change.

The third codeBlock, called "eventHandlers", is always created by the SB/MVC programmer. In this codeBlock the programmer implements functions referenced by the function pointers in the

initialize() function. Generally, these eventHandlers manage the onclick event of HTML Views.

Before I dig into code, it's worth noting a few things about the structure of an SB/MVC Model. First, each function is wrapped in a code tag; and that tag contains enough information that the initialize function could be created at runtime by XSLT. Second, the code tag also binds the function to the event(s) so you can create an XSLT file that does an internal audit on your code; it can flag functions that have no event and events that have no eventHandler. Each of these "features" is an example of the benefits that accrue when programs become data.

The notify() function is the workhorse of any SB/MVC app. The notify() function is called by the initialize() function and by all eventHandlers with this prototype:

```
notify([event])
```

One of the variables declared when the SB/MVC app starts is a nodeList named nlstEvent. This nodeList uses a cached IXMLDOMSelection object. When notify() receives a call it uses the getEvent() helper function to look up the passed-in event in the nlstEvent property, getting a node object back; the node object is one of the event tags depicted in the previous figure above. Here's where that call is made:

```
var aNotify=getEvent(event).getAttribute("notify").split(" ")
```

SBPNews is built without explicit schemas so it can't take advantage of the simpler and faster code that comes with the ID/IDREF(S) data types. For that reason, the function above splits the text of view ids into an array. This array is then iterated and some polymorphic JScript code is called:

```
view=model.selectSingleNode("/model//view[@id='"+aNotify[each]+"']")
document.all(view.getAttribute("boundTo")).transform(view.firstChild)
```

The first line selects the view node in the Model with the id of the current item in the event's notify attribute. Notice the "//" in the data signature - this permits the parser to find a view node even if another node (such as a tutorial node as in the case of the debugTransform view).

The Code Path

The second line uses this view and it's boundTo property to get a DHTML reference to the actual HTML tag that will receive the transformed XML. One limitation to the current design of *SBPNews* is that the XSLT transform node must be the firstChild of the current view node. The reference to the transform() method is possible because during the initialize() function the HTML tag obtained an expand property named "transform" that was a function pointer to a JScript function. Generally, this JScript function is simply:

```
this.innerHTML= model.transformNode(xslView)
```

where "xslView" is the XSLT transform stored inside the current view node (of course this refers to the current object which is the actual HTML tag getting its innerHTML property updated).

That is, essentially, all there is to the most powerful part of an SB/MVC application. This is yet another benefit to having a declarative language like XSLT do all the heavy lifting. The scripting engines have but to glue the declarative code together with the output mechanism.

Your Job

I will bring this Program Spec to a close with an examination of two eventHandlers. One is the most complicated; the other is the simplest. The complicated eventHandler is the `storyHeadlineSelected` codeBlock. As I noted above, when you create an SB/MVC application you will spend a good deal of time in this eventHandlers codeBlock.

The `storyHeadlineSelected()` function has to manage several things. First it has to sense when the user Alt-clicked a story headline (it will then display the XSLT that is called when a story headline is clicked normally). Second, since the View that's bound to the `storyHeadlineSelected()` function is a container, the eventHandler has to sense whether a click occurred on a heading or on a story headline; only in the case of the latter will the eventHandler respond by firing the `headlineSelected` event (it's third responsibility).

Finally, the `storyHeadlineSelected()` function had a bug. I rev'd out the fix so that you can see the bug in action. All I have to do is change the rev tag's deprecated attribute to be an added attribute, and the XSLT will send the bug fix to the client at runtime.

By contrast, the `tutorialRequested` eventHandler has but one duty: to call `notify()` with the `tutorialRequested` event.

Overview

An application is schema-based programming (in part) to the extent that it relies on XSLT. The remainder of this program spec will explain how the various XSLT transforms used in *SBPNews* work. Most of these transforms animate the application, but a handful of transforms are called once, at startup, to render the default HTML. This section will cover all these instances.

SBPNews has seven XSLT transforms working on behalf of five events and six Views. Let's begin by exploring this algebra a little. As you may recall from previous remarks, every transform is bound to one View, but one View can be served by more than one transform. Each transform produces HTML based on part of the XML data in the Model, so it makes sense that there is a 1:1 relationship between transform and View (otherwise the same HTML will be duplicated by additional Views). On the other hand, Views are merely containers that expose events, nothing more; so it's equally obvious that, depending on context, and/or the intention of the user, that different HTML can appear in the same place in the browser (though, of course, not at the same time). Finally, a single event can notify any number of transforms so it's not surprising that the cardinality of events is smallest.

The work a transform is called upon to do can vary widely (and so can the challenge of programming the transform). The level of difficulty is a function of the amount of data processing the transform must do and the complexity of the underlying XML schema. For example, the (implicit) schema used by SBPNews is vastly simpler than a similar schema used by The `Abolitionist` electronic journal. This is a good thing since coming to terms with SBPNews will take less time and provide a good foundation when you take on the more complex schema in The Abolitionist.

A Wide Range of Complexity

The simplest transform in *SBPNews* is found inside the `viewStoryHead` node in the Model. Here's what that transform looks like:

```
<view id="storyHead" boundTo="viewStoryHead" >
  <xsl:transform version="1.0" >
  <xsl:template match="/" >
  <xsl:value-of select="/model/@view" />
  </xsl:template>
```

```
  </xsl:transform>
</view>
```

As you can see, this transform has but one duty: to display the current value of the @view attribute in the Model.

At the other extreme is the debugTransform view. Here's the XSLT for that transform:

```
<view id="debugTransform" boundTo="viewDebugTransform" >
  <xsl:transform version="1.0" >
    <xsl:variable name="debugEvent" select="/model/@debugEvent" />
    <xsl:variable name="debugNotify"
select="/model/event[@id=$debugEvent]/@notify"     />
    <xsl:output method="xml" indent="yes" omit-xml-declaration="yes" />
    <xsl:template match="/" >
      <div >
      <b >For Event:
        <xsl:value-of select="$debugEvent" />
      </b>
      </div>

XSLT Transform(s)

      <xsl:value-of select="$debugNotify" />
      <hr style="color:white" />
      <xsl:call-template name="tokenList" >
        <xsl:with-param name="list" select="$debugNotify" />
      </xsl:call-template>
    </xsl:template>
    <xsl:template name="tokenList" >
      <xsl:param name="list" />
      <xsl:variable name="nlist" select="concat(normalize-space($list),' ')"
/>
      <xsl:variable name="first" select="substring-before($nlist,' ')" />
      <xsl:variable name="rest" select="substring-after($nlist,' ')" />
      <xmp >
        <xsl:copy-of select="model/view[@id=$first]" />
      </xmp>
      <xsl:if test="$rest" >
        <xsl:call-template name="tokenList" >
          <xsl:with-param name="list" select="$rest" />
        </xsl:call-template>
      </xsl:if>
    </xsl:template>
  </xsl:transform>
</view>
```

This transform does some comparatively heavy lifting. First, it sets up some variables (actually constants whose value may be different each time the transform is called). The first variable, debugEvent, gets its value by looking to the debugEvent attribute created by the eventHandler when the Alt key was sensed. The second variable, debugNotify, picks up the nodeList of view ids managed by the event captured in the debugEvent variable. The transform then displays these variable values and proceeds to tokenize the space delimited string of view ids. The tokenization takes place in a called template, tokenList. This is the only transform in *SBPNews* that uses more than one XSLT template.

The tokenList transform uses param and variable XSLT tags to separate out each word in the string and uses that word as the variable in the data signature of the copy-to XSLT tag. The

transform works as many XSLT transforms do, by recursion (rather than iteration). Each call to the tokenList transform gets the XML of another transform in the Model and renders it literally using the <xmp> tag.

Schema-Based Programming Walkthrough

The Tutorial. The tutorial view is the next most complex piece of XSLT code. Here's the transform:

```
<view id="tutorial" boundTo="viewTutorial" >
  <xsl:transform version="1.0" >
    <xsl:output method="xml" indent="yes" omit-xml-declaration="yes" />
      <xsl:template match="/" >
      <xsl:for-each select="model/tutorials/tutorial" >
        <xsl:variable name="t" select="@id" />
        <table border="0" width="100%" style="table-layout:fixed" >
          <col width="5%" />
          <col width="95%" />
          <caption >Tutorial:
            <xsl:value-of select="@name" />
          </caption>
          <xsl:for-each select="//tutorial[@elucidates]" >
            <xsl:sort select="@step" />
          <tr >
            <td >
              <span class="tutorial" >
                <xsl:value-of select="@step" />
              </span>
            </td>
            <td >
              <span >
                <xsl:value-of select="@description" />
              </span>
              </td>
            </tr>
            <tr >
              <td />
              <td >
                <xmp >
                  <xsl:copy-of select="." />
                </xmp>
              </td>
            </tr>
          </xsl:for-each>
        </table>
      </xsl:for-each>
    </xsl:template>
  </xsl:transform>
</view>
```

This transform needs two related bits of data in the Model. First, the id and name of each tutorial in the Model (though for simplicity, there is only one tutorial (I leave it as an exercise for the reader to add a tutorial of their own). Second, based on the tutorial's id, the transform iterates the collection of tutorial nodes for the current tutorial object in the Model, regardless of their location (this is what the recursive descent operator "//" does). Before the transform displays the contents of the tutorial nodes, it sorts the collection on the step attribute. If there were two or more tutorials in the collection the sort would be on the elucidates and step attributes. The transform completes by rendering all the content of the tutorial tag, including

any XML or text surrounded by the `tutorial` tag itself. Again, it's left as an exercise to embellish the `tutorial` tag contents with more detailed instructions (within their own `<instruction>` tag, perhaps).

Twin Views

The `story` and `references` views are practically identical. This is not surprising since the content they expose is dependent on the same node of XML in the Model, the `selectedStory` attribute. These two transforms use a data signature that's almost identical. The `references` view uses the expression:

```
model/edition/story[@id=/model/@selectedStory]/reference
```

and the `story` view uses the same expression without the "/reference" reference. One difference between the two transforms is that the `story` view only gets one node when it calls on the `<xsl:for-each>` element and the `references` view gets a collection. The technique `story` view uses is another common XSLT trick to set the context in order to make multiple references to different aspects of that context; story needs to render the headline and the content elements of the story.

Another Set of Twins

The `headlines` and `abstracts` views are also similar. Both transforms are based on the data signature, model/edition/story, but the `headlines` view needs to drill one level deeper because it is listing all the headlines in the current edition and the `abstract` view is interested in the collection of stories in the edition. Both views, however, need to display the headline for each story, so the `headlines` view looks "up" a level for the `id` attribute assigned to the new h4 tag while the `abstracts` view looks directly at the current story to create the same heading. Finally, the `abstract` view also needs to render the text of the abstract so it uses an extra `<xsl:value-of>` element.

The Rest of the Story

The other XSLT transforms are in the `sbpNews.xslt` file. The first transform is the one that renders the web page itself. This transform contains all the HTML you will see if you use the "View XSL Output" utility I mentioned earlier. One interesting detail is in the expression used to render step one of the tutorial. That expression is `<xsl:copy-of select="model/tutorial[@step='1']" />` and it moves the tutorial node stored in the Model (so it can be included by the transform inside the `tutorial` view) out into the client.

Of course one of the most important functions of the master template is to fetch all the client-side script that is also stored in the Model (again, so that the information implicit in the functions themselves can be manipulated by the Model). So what parts of the client-side script are copied out of the Model? Well if you look at the first template after the master template you will see a template that identifies the spec, limitation, and any rev tag that contains a deprecated attribute (rev tags with the added attribute survive transformation). This template ignores these tags, and any data they contain remains in the Model. The next two templates manage the comments maintained in the Model. The final template manages the code that's kept in the Model. This final template is recursive. For example, a codeBlock contains code, so the apply-templates inside the template processes the code children contained within the codeBlock. Likewise, the templates I just described process any comments, rev, or limitation tags in the code tag.

That completes the tour of schema-based programming techniques used in *SBPNews*.

Summary

As you can see, a typical SB/MVC application is not complex, yet it is extremely flexible. The hardest thing to follow is the relationship between eventHandlers, events, views, and Views. Much of the JScript in the Model is very basic. You pay for this simplicity when you invest time in developing your XSLT transforms. But in the end, you have a robust, lucid, and extensible application, which was what I set out to achieve when I first laid down the blueprints for a new way to create client-side code, schema-based/model-view-controller applications.

Note

This is a conversion of the original polymorphic HTML document, the original version of which can be found in the code folder.