XML Response Patterns

By David Leibs

While the Web is rapidly becoming a vast repository for ecommerce data such as price lists and product specifications, HTML documents lack the structure required to make them useful data inputs to business applications. By providing a standard way of specifying structure, describing metadata and capturing complex information, XML represents a significant step forward for Web applications.

XML enforces a logical tree structure for the organization of information. It has a root element, branch elements and leaf elements that allow for "in-context" information management. It provides a syntax to define the attributes of each element, so that XML documents gain many of the database features required for sophisticated information management.

In a project at Neometron we used dynamically built XML patterns to communicate rich information from a application server to various clients. Initially we build the XML to be returned by http requests by passing text streams around a set of Smalltalk methods that would build the text representation of the XML to be returned while accessing the information model to retrieve the variable part of the information to be returned. The problem with this approach was that changing our XML representation also required finding and modifying some very ugly code all written by different people in somewhat different styles. We found that this difficulty encouraged living with representations longer than needed.

Our experience with Active Server Pages encouraged us to build a similar mechanism for our Smalltalk framework. One thing we had found in working with Active Server Pages was that the pages could become very ugly. The primary problem with the active server pages approach is that the classic control structures from languages like Visual Basic and JavaScript are carried over and mixed into what should be a declarative structure.

So while simple filing in of attributes is understandable as in:

```
<IMG SRC="<%=somefunction()%>"><P>
```

code that has iteration and branching will degenerate to:

```
<%
      For Each Item in Request.Form
            if Request.Form(Item.Count Then
                  For initLoop = 1 to Request.Form(Item).Count
%>
<%Response.Write(Item & ": Index = " & intLoop & " Value ="_
      Request.Form(Item) (intLoop)) %> <BR>
<%
  Next
  Else
%>
<% = Request.Form(Item) %>
  <% End If
     Next
  %>
```

We chose to restrict ourselves to well formed XML with code fragments in the leaves. We would use proper XML to implement conditionals, and would have a template-based iteration. We called our system of embedding Smalltalk code in XML "Response Patterns".

The following XML fragment shows most of the interesting features of our response pattern language.

```
<notes>
  <rp:maplist list="self notes">
    <note id="[self id]" type="[self type]">
      <field name="type"><rp:eval>self type</rp:eval></field>
```

```
      <textfield name="message"><rp:eval>self message</rp:eval></textfield>
      <from><role id="[self ownerId]" name="[self ownerName]"/></from>
      <field name="created" type="timestamp">
         <rp:eval>self created</rp:eval>
      </field>
      <rp:switch case="self remove">
       <rp:case of=""/>
       <rp:otherwise>
        <field name="remove" type="timestamp">
          <rp:eval>self remove</rp:eval>
        </field>
       </rp:otherwise>
      </rp:switch>
   </note>
</rp:maplist>
</notes>
```

The evaluation strategy is to walk the structure copy of a tree of XML nodes and replace special evaluation nodes with the results of the evaluation.

> Attributes that have a beginning and ending square bracket are evaluated and the value converted to a string.
>
> The rp:maplist node will evaluate an expression that produces a sequencable collection of objects. For each object a copy of the subtree will be evaluated with self bound to each successive object. The collection of new subnodes finally replaces the rp:maplist node.
>
> The rp:switch node has a case attribute that is evaluated that provides a tag that is used to select one of the cases or otherwise if none match. The switch node is replaced with the evaluated case.
>
> The rp:eval node is replaced by a PCData node whose text is the text that resulted from the evaluation of the expression contained within. An optional attributes isText allows for having the rp:eval return tagged xml node structures instead of PCData. Further overloading of rp:eval allowed for specifying a file, url, or method to be fetched externally, parsed and then evaluated. This external fetching facilitated better modularity and also allowed for recursive response patterns.

Our response patterns system was so successful that the developers rewrote large parts of our server in the response pattern style. This allowed for easier understanding of the XML that was being shipped to remote systems because of their self documenting nature. Remote modification of a server could be done by file editing since the server automatically checked modification times.

Profiling indicated that the simple evaluation done at tree walk time was not efficient enough to implement a full application server. This is where our choice of Smalltalk really paid off the most. We were able to turn the evaluator into a compiler by walking the tree of nodes and compiling each Smalltalk fragment into either a copying block or a compiled method. Runtime evaluation then became: structure copy the tree and replace embedded compiled code with the invocation of the compiled code using a suitable receiver.