XML: A Delphi Implementation, 2

by David Baer

In the two months since I started to write part one of this article, the XML juggernaut has continued to gather momentum. At that time there were ten or twelve books on the subject at my local technical bookstore. Today the number is closer to twenty. So, it looks like we're on to something here.

We'll pick up where we left off last time, adding some additional capabilities to the class framework begun in part one. The main order of business will be adding an XML input capability to the framework, but we'll supply a couple of other convenient features as well. I won't attempt to recap the material discussed in part one here, other than to remind you that the focus of the framework's capabilities is on the use of XML for inter-application data communications. There are any number of browser oriented capabilities which we'll be ignoring in this context.

Before we get started, I need to address two errors in part one. The first was simply a case of typing on auto-pilot and not spotting the error until after the magazine had gone to press. I said that XML names were like Delphi names, but could also contain hyphens, periods and semicolons. That should have been *colons*, not *semicolons*.

A second correction is needed for something that was an outright case of my not having all the facts. In the discussion of Document Content Declarations (DCDs), I stated that for elements identified as containing dates, the format of such items must comply with an ISO standard that mandated a format of YYYYMMDD. I didn't learn until later that there are a number of compliant formats, of which the preferred is YYYY-MM-DD. The standard allows the hyphens to be dropped and even allows for omission of the century. How quaint. In all, this was a disappointing discovery, but the situation is still preferable to the usual chaos of the myriad of localized date formats found in the world today.

Finally, there's one generalization made in part one that merits clarification. This is not exactly relevant to building a Delphi XML capability. On the other hand, this subject is extremely important in the evolution of XML, and it's closely tied in with the foundations upon which Microsoft is basing their own XML E-commerce functionality, which is being promoted in the BizTalk initiative.

Data Redux

Recall that I described DCDs as an emerging and preferable alternative to Document Type Definitions (DTDs) for use in XML document validation. I had a chance to hear Adam Bosworth, Microsoft's principal XML evangelist, speak in May of this year, and even had a brief opportunity to ask him a few questions directly after the talk. One of those questions had to do with the viability of DCDs as implemented in IE5 regarding the degree of compliance with whatever the W3C eventually blessed as a standard. His reply was that he was growing optimistic that they were close (in IE5) to what that standard would look like upon final approval (but he also emphasized that Microsoft would definitely support whatever specification prevails in the end).

About a month later, I received an email from *Our Esteemed Editor*, saying that he had just returned from the Microsoft Tech Ed Europe conference. At that event, MS Vice President Paul Maritz had emphasized one theme. He told the attendees that if they took only a single message home with them, it should be that Microsoft was resolutely committed to supporting XML and Data Reduced on all fronts. *OEE*'s question: what is this *Data Reduced* business? Is it the same thing as the DCD capability?

Well, I didn't know. I had never heard of this thing called XML Data Reduced, and I'm fairly confident that Adam Bosworth never used that phrase in his presentation only a few weeks earlier. It was off to the web to see what I could find. The answers were not easy to come by. For all of Paul Maritz's stated MS commitment to XDR, the term was not in evidence on MS's MSDN XML pages (a good source of information, at http://msdn.microsoft.com/xml/c-frame.htm#/xml/default.asp at the time of writing). Nor could I find a concise explanation at the W3C XML site, other than that this whole topic comes under the general heading of *XML Schemas*.

I finally found an authoritative source of information in Lee Buck, the principal technologist of a company called Extensibility. Extensibility produces a product that allows organizations who want to begin working with XML schemas to do so while minimizing the risk of too-early adoption (check www.extensibility.com). To make a long story short, Lee filled in the gaps of my understanding, and the following is what seems to be going on.

MS and others submitted a proposal in early 1998 for an improved alternative to DTDs. This capability was called *XML Data*. As a superset of DTDs, XML Data was a large and fairly ambitious specification that was unlikely to see any kind of rapid adoption. But the improvements over DTDs were significant.

First of all, the declarations were in straight XML, not the confusing, foreign syntax of DTDs. Furthermore, a number of serious deficiencies of DTDs were rectified. Improved capabilities include data typing, min/max cardinality and value range constraints. Significantly, the specification allowed for a flexible type of inheritance for declarations (these are, after all, specified in XML, making them naturally extensible).

But complexity was still a major concern. As a result, a group, which again included Microsoft, submitted an alternative schema proposal later in 1998 which simplified the XML Data specification. This alternative was called Document Content Declarations.

So what's Data Reduced? You've probably already guessed it: it refers to an XML Data specification that's been, well, reduced. It too is a simplification of the original XML Data proposal. To the casual observer, the two specifications, DCDs and XDR, are rather similar, and they address many of the same concerns.

But if the broad strokes are the same, the details are not. Confusion is compounded by the fact that MS's web pages usually reference these XML developments generically as 'XML schema'. So what's with MS? Can't they make up their minds? Certainly I'm in no position to answer that, but it's evident that the DR flavor of XML schema specification is currently very much in favor at MS. Whatever proposal gains the final W3C blessing (and there are still other proposals kicking around as well, if only as dark horses at this point), I'm convinced that the schema capability will play an extremely important role in XML's future.

A Parcel Of Parsers

Well, enough of ongoing XML developments. Let's get down to the business of adding a parse/load capability to the framework. The first item on the agenda, needless to say, is to determine where the parsing capability is going to come from.

One alternative is to write one. If we wanted a basic non-validating parser that did not need to resolve external entities, this really wouldn't be all that difficult. However, if validation is added as a requirement, that's quite another matter. If we also add the requirement that the parser must support whatever schema validation flavor wins the day (when that day comes), then we can kiss more than a few of our weekends goodbye over the next year.

No, it's clearly preferable to integrate an existing capability. Fortunately, there's a good deal of technology available for this, even when we add the obvious requirement that it must be freely available. IBM, Sun, Microsoft and others have solid implementations available for the cost of an online download session.

So, let's add another rather obvious requirement that it must easily integrate with our Delphi-based class framework. That pretty much narrows the field to one. The offerings of IBM *et al* are either Java or C++ based. Microsoft's capabilities are predictably accessed via COM.

So, let's explore the MS possibilities a bit further. The W3C offers recommendations for two XML parsing standards. One is the Document Object Model, within which a parser obviously exists, but that parser is not exposed to clients. The other standard is known as *Simple API for XML* (SAX, for short). Clients of a SAX parser must supply their own storage and relationship management capabilities. The SAX parser is responsible for many things: parsing, validating, resolving external entity references, and so forth. But once the client is handed pieces of the incoming document, its job is done.

So, SAX would appear to be the way to go. We've already got the storage and management issues addressed, and we don't need the overhead of a redundant parallel set of capabilities such as those we'd get if we called upon DOM services for parsing.

There's just one small problem, though. The msxml.dll does not currently include a SAX offering. DOM is there in full glory, but there's no SAX support. Going back to my May encounter with MS's Adam Bosworth, he said that the SAX capability would in all probability be forthcoming, but he didn't know when that would be.

So, we're stuck with DOM. Although this isn't ideal, it's not really too onerous. We will have a momentary bloat of redundantly maintained document nodes during the loading operation. But once the data is transferred into our Delphi class nodes, the DOM objects will be destroyed.

Start Your Parsers

To prepare your machine to work with what follows, you'll need to ensure that you have the latest copy of msxml.dll. If you've installed IE5, then you almost certainly have it. If not, see if you have this file installed. If it's there and its size is in the neighborhood of 500Kb, then that's the one. If it's there but its size is only about 100Kb, then that's an old version that doesn't contain the support for DOM. You'll need to get the current version, which is available from MS's website.

I didn't do any of these things. Although I know it's normally considered unsafe to install a dll by simply copying it from another machine, I decided to throw caution to the wind and try this shortcut. I picked up the dll from a machine upon which IE5 had been installed, and loaded it onto both Windows95 and NT systems. After two months, I've yet to see any adverse effects. However, I can't guarantee your experience would be as trouble free.

If you need to install it, then you'll also need to ensure that it's been registered. To register it, execute `regsrv32 msxml.dll` from the `Run` prompt. The other thing that will be needed is the type library definition for the COM classes contained in msxml.dll. If you wish, you may use Delphi to import this declaration, or you may just use the copy I've included with the accompanying source files on this month's CD-ROM.

I have to admit that I was somewhat intimidated the first time I took a look at the type library file. At over 1,700 lines, there's a lot to digest. But if you've become familiar with the DOM methods (visiting the MSDN site previously mentioned is one way to do so), it soon starts to make sense and is not nearly so overwhelming. I also must say that in writing and testing the code for this article, I was constantly amazed at how trouble free the process was. Most things simply worked the first time out,

```
type
  TCharEntity = (ceLt, ceGt, ceQuot, ceApos, ceAmp);
  TCharEntities = set of TCharEntity;
  TSubstituteCharEntitiesEvent = procedure(Sender: TObject;
    var Text: String; var SkipTranslation: Boolean) of
      Object;
  EXmlDError = class(Exception);
  EXmlDParseError = class(Exception)
    FErrorCode:   Integer;
    FReason:      String;
    FSrcText:     String;
    FLine:        Integer;
    FLinePos:     Integer;
  public
    constructor Create(ParseError: IXMLDOMParseError);
    procedure ShowParseError;
    property ErrorCode: Integer read FErrorCode;
    property Reason: String read FReason;
    property SrcText: String read FSrcText;
    property Line: Integer read FLine;
    property LinePos: Integer read FLinePos;
  end;
  TXmlDStructureNode = class(TXmlDNode)
    private
      FAttrList: TXmlDAttrList;
      ...
    public
      ...
      property AttrList: TXmlDAttrList read FAttrList;
    end;
  TXmlDDocument = class(TXmlDStructureNode)
    private
      ...
      FAttrCharEntities: TCharEntities;
      FTextCharEntities: TCharEntities;
      FOnOutputAttrValue: TSubstituteCharEntitiesEvent;
      FOnOutputTextValue: TSubstituteCharEntitiesEvent;
      DiscardUnsupportedItems: Boolean;
    protected
      ...
      procedure DecodePrologAttrs(S: String);
      procedure LoadFromDOMDocument(Doc: IXMLDOMDocument);
      procedure LoadChildNodes(ParNode: TXmlDNode;
        ParDOMNode: IXMLDOMNode);
      procedure LoadAttributes(Node: TXmlDElement;
        DOMNode: IXMLDOMNode);
      procedure AssignNodeToTreeNode(XmlNode: TXmlDNode;
        TreeNode: TTreeNode);
      procedure AssignAttrNodesToTreeNodes(ParXmlNode:
        TXmlDNode; ParTreeNode: TTreeNode);
    public
      ...
      procedure AssignTo(Dest: TPersistent); override;
      procedure LoadFromStream(Stream: TStream;
        ValidateOnParse: Boolean = True;
        DiscardUnsupportedItems: Boolean = False);
      procedure LoadFromFile(const FileName: String;
        ValidateOnParse: Boolean = True;
        DiscardUnsupportedItems: Boolean = False);
      ...
      property AttrCharEntities: TCharEntities
        read FAttrCharEntities
        write FAttrCharEntities;
      property TextCharEntities: TCharEntities
        read FTextCharEntities
        write FTextCharEntities;
      property OnOutputAttrValue:
        TSubstituteCharEntitiesEvent
        read FOnOutputAttrValue write FOnOutputAttrValue;
      property OnOutputTextValue:
        TSubstituteCharEntitiesEvent
        read FOnOutputTextValue write FOnOutputTextValue;
  end;
  TXmlDAttrList = class(TPersistent)
  private
    List:     TStringList;
    FOwnerNode: TXmlDStructureNode;
  public
    ...
    property OwnerNode: TXmlDStructureNode read FOwnerNode;
  end;
```

➤ *Listing 1*

and the few hitches encountered were minor and brief.

Load 'Em Up

Adding an input capability to the class framework was done by adding two new public methods to the `TXmlDDocument` class, `LoadFromFile` and `LoadFromStream`. Listing 1 contains some snippets of the frameworks class definitions, mostly showing only additions and modifications to the full set of declarations.

Note both methods have three parameters. The first is the source file name or the stream reference. In the case of `LoadFromFile`, the name can actually specify a URL, not just a local or network accessible file. The second parameter, `ValidateOnParse`, allows any validation during parsing to be skipped.

The third parameter, `DiscardUnsupportedItems`, requires a little explanation. Recall from last time that there were several XML document component types that we weren't bothering to support. For our load capability, if any of these types are encountered, we offer the choice of simply discarding them with no further ado, or of raising an exception (the default choice).

Listing 2 contains the implementation code for all the routines participating in the parse/load process. As you can see in both load methods, invoking the DOM parse services requires very little code. `LoadFromFile` obtains an `IXMLDOMDocument` interface reference and executes the DOM method `load`, the parameter of which specifies the URL of the input source.

`LoadFromStream` has to do a little more setting up. An alternative to the DOM `load` method is `LoadXML`, which takes a string parameter. Here, if the input stream is not a `TStringStream`, we create one and copy the contents of the input stream into it. At that point we're ready to call the `LoadXML` method. If you examine the type library declarations, you'll note that all strings are `WideStrings`. We don't need to concern ourselves with this, as Delphi takes care of the details.

Most of the code needed for our parsing/loading is involved in transferring the document nodes from DOM nodes to our own Delphi object nodes. The method `LoadFromDOMDocument` gets the ball rolling. Transferring nodes from DOM to our nodes is a process handled by the recursively called method `LoadChildNodes`. Before calling this to get the recursion started, `LoadFromDOMDocument` needs to confirm we've got something to transfer.

The msxml load routines do not raise an exception upon encountering erroneous input. Instead, the DOM document object supplies a property, `parseError`. This is an interface which provides several properties containing error information. One of those properties, `errorCode`, will be nonzero if an error was encountered. When this happens, our framework will transform the condition into a Delphi exception which passes the error information through in an `EXmlDParseError` object (Listing 1 has the declaration). We pass the DOM `parseError` interface to the exception constructor, which does the transfer.

While we're on the subject of `EXmlDParseError`, note that I've included something a little nonstandard. This is a convenience method, `ShowParseError`, which formats and shows an error dialog with pertinent information.

Excavating DOM Nodes

Apart from one little surprise in how the MS facilities deal with

information in the XML prolog, transferring the information from DOM into our framework is quite straightforward. We'll get to that in a moment.

First let's dispense with prolog business. In the XML examples presented in the previous article, the first XML line always looked liked this:

```
<?xml version="1.0"?>
```

where the version is specified in a way that looks identical to an element attribute.

In fact there are two other pieces of information that can go in the prolog: *encoding* and *standalone*. For example:

```
<?xml version="1.0"
  encoding="UTF-8"
  standalone="yes">
```

The XML specification doesn't define these as attributes or as anything else. They're just optional parts of the prolog statement. So what does the msxml DOM implementation do with them? It takes the lot of them and groups them into an XML component type called a *processing instruction* (PI).

You may recall that in part one, I stated I didn't think PIs had any business being used in inter-application data communications. They are highly application specific and can be employed for any number of things: formatting instructions, compiler pragmas, you name it. PIs have a simple representation in the DOM framework, and it would have taken little effort to add this extra node type to our framework. But I truly think they should be avoided in most cases, so I'm stubbornly resisting providing support for them.

Instead, I've taken these prolog 'attributes' and re-mapped them from the supplied PI into attribute-like properties of the `TXmlDDocument` class. To do so, I moved the `AttrList` property from the `TXmlDElement` class up one level in the hierarchy to `TXmlDStructureNode`. This alteration also necessitated a few modifications to the methods participating in document save processing. If you're interested in the details, examine the code for this article on the CD-ROM.

➤ *Listing 2*

```
constructor EXmlDParseError.Create(ParseError:
  IXMLDOMParseError);
begin
  inherited Create('XML Parse Error');
  FErrorCode := ParseError.errorCode;
  FReason := ParseError.reason;
  FSrcText := ParseError.srcText;
  FLine := ParseError.line;
  FLinePos := ParseError.linePos;
end;
procedure EXmlDParseError.ShowParseError;
var S:  String;
begin
  S := 'XML Parse Error:' + FReason + 'Line=' +
    IntToStr(FLine) + ' LinePos=' + IntToStr(FLinePos);
  MessageDlg(S, mtError, [mbOK], 0);
end;
procedure TXmlDDocument.LoadFromFile(const FileName: String;
  ValidateOnParse, DiscardUnsupportedItems: Boolean);
var Doc:  IXMLDOMDocument;
begin
  Doc := CoDOMDocument.Create;
  Doc.validateOnParse := ValidateOnParse;
  Self.DiscardUnsupportedItems := DiscardUnsupportedItems;
  Doc.load(FileName);
  LoadFromDOMDocument(Doc);
end;
procedure TXmlDDocument.LoadFromStream(Stream: TStream;
  ValidateOnParse, DiscardUnsupportedItems: Boolean);
var
  Doc:  IXMLDOMDocument;
  SS: TStringStream;
begin
  Doc := CoDOMDocument.Create;
  Doc.validateOnParse := ValidateOnParse;
  Self.DiscardUnsupportedItems := DiscardUnsupportedItems;
  if Stream is TStringStream then
    SS := TStringStream(Stream)
  else begin
    SS := TStringStream.Create('');
    SS.CopyFrom(Stream, Stream.Size);
  end;
  SS.Position := 0;
  Doc.loadXML(PChar(SS.DataString));
  LoadFromDOMDocument(Doc);
  if SS <> Stream then
    SS.Free
  else
    SS.Position := 0;
end;
procedure TXmlDDocument.LoadFromDOMDocument(Doc:
  IXMLDOMDocument);
var Err:  IXMLDOMParseError;
begin
  Clear;
  Err := Doc.parseError;
  if Err.errorCode <> 0 then
    raise EXmlDParseError.Create(Err);
  NodeName := Doc.nodeName;
  LoadChildNodes(Self, Doc);
end;
procedure TXmlDDocument.LoadChildNodes(ParNode: TXmlDNode;
  ParDOMNode: IXMLDOMNode);
var
  ChildDOMNode: IXMLDOMNode;
  NewNode:  TXmlDNode;
begin
  ChildDOMNode := ParDOMNode.firstChild;
  while ChildDOMNode <> nil do begin
    NewNode := nil;
    case ChildDOMNode.nodeType of
      NODE_ELEMENT :
      begin
        NewNode := CreateElement(ChildDOMNode.nodeName);
        LoadAttributes(TXmlDElement(NewNode), ChildDOMNode);
      end;
      NODE_TEXT :
        NewNode := CreateTextNode(ChildDOMNode.nodeValue);
      NODE_CDATA_SECTION :
        NewNode :=
          CreateCDataSection(ChildDOMNode.nodeValue);
      NODE_PROCESSING_INSTRUCTION :
        DecodePrologAttrs(ChildDOMNode.nodeValue);
      NODE_COMMENT :
        NewNode := CreateComment(ChildDOMNode.nodeValue);
      NODE_DOCUMENT_TYPE :
        TXmlDDocument(ParNode).DocumentTypeDefinition :=
          ChildDOMNode.xml;
      else
        if not DiscardUnsupportedItems then
          raise EXmlDError(
            'XML document contains unsupported ' +
            'node type of ' + ChildDOMNode.nodeTypeString);
    end;
    if (NewNode <> nil) and (ParNode <> nil) then
      ParNode.AppendChild(NewNode);
    LoadChildNodes(NewNode, ChildDOMNode);
    ChildDOMNode := ChildDOMNode.NextSibling;
  end;
end;
procedure TXmlDDocument.DecodePrologAttrs(S: String);
var I: Integer;
begin
  I := Pos(' ', S);
  while I > 0 do begin
    FAttrList.Add(StringReplace(
      Copy(S, 1, I - 1), '"', '', [rfReplaceAll]));
    S := TrimLeft(Copy(S, I + 1, $7FFF));
    I := Pos(' ', S);
  end;
  FAttrList.Add(StringReplace(S, '"', '', [rfReplaceAll]));
end;
procedure TXmlDDocument.LoadAttributes(Node: TXmlDElement;
  DOMNode: IXMLDOMNode);
var
  I:  Integer;
  Attributes: IXMLDOMNamedNodeMap;
  Item: IXMLDOMNode;
begin
  Attributes := DOMNode.attributes;
  for I := 0 to (Attributes.length - 1) do begin
    Item := Attributes[I];
    Node.FAttrList[Item.nodeName] := Item.nodeValue;
  end;
end;
```

As for the rest, the story is fairly simple. For all but element attributes, we use the DOM `firstChild` and `nextSibling` properties to snorkel through the document, creating the appropriate types on our side of the fence and copying the DOM node contents into them. The code that does this can be seen in methods `LoadFromDOMDocument` and `LoadChildNodes`.

The case statement in `LoadChildNodes` uses constants that come courtesy of the type library to identify the DOM node types. Apart from the aforementioned special treatment of PIs, there's only one other technique to note. DOM attribute nodes are not accessed via the usual child and sibling properties. Instead, DOM defines an interface, named `IXMLDOMNamedNodeMap` in the MS world, which is used to access attributes. Copying attribute values into our element's `AttrList` is performed in method `LoadAttributes`.

So there you have it. We've gained powerful parse/load capabilities with little effort. But let's not stop there. There's a couple of other things we can do to add value to the class framework.

All This And AssignTo Too?

One thing that would certainly be convenient on occasion is the ability to easily load a tree view with an XML document. After all, the document tree structure is a nearly perfect mapping. So how do we go about doing that? It's easy.

We simply implement a class derived from `TTreeView` in which the `Assign` method knows how to transfer the contents of a `TXmlDDocument` instance into the tree view. OK, all of you who are waving your arms and shouting 'No! That's not how you do it!', give yourselves a pat on the back, go outside and take a bonus fifteen minute coffee break. As for the rest of you, it's time for a lesson in the clever, elegant `TPersistent.AssignTo` method.

When writing a class of any complexity, it's usually easy to identify what other commonplace classes have a compatibility with respect to data structures that can be held or represented. For such classes, we can provide a transfer mechanism within our class as a service within the `Assign` method. But we clearly cannot anticipate every possible compatible class which would benefit from this assignability.

To accommodate this situation, the VCL provides us with a highly flexible alternative. It works as

➤ *Listing 3*

```
function CharEntitiesReplace(const S: String;
  CE: TCharEntities): String;
begin
  Result := S;
  if ceAmp in CE then
    Result := StringReplace(Result, '&', '&amp;',
      [rfReplaceAll]);
  if ceLt in CE then
    Result := StringReplace(Result, '<', '&lt;',
      [rfReplaceAll]);
  if ceGt in CE then
    Result := StringReplace(Result, '>', '&gt;',
      [rfReplaceAll]);
  if ceApos in CE then
    Result := StringReplace(Result, '''', '&apos;',
      [rfReplaceAll]);
  if ceQuot in CE then
    Result := StringReplace(Result, '"', '&quot;',
      [rfReplaceAll]);
end;
procedure TXmlDDocument.AssignTo(Dest: TPersistent);
var
  TV: TTreeView;
  TN: TTreeNodes;
  TreeNode: TTreeNode;
  procedure AddChildNodes(ParXmlNode: TXmlDNode;
    ParTreeNode: TTreeNode);
  var
    XmlNode:  TXmlDNode;
    TreeNode: TTreeNode;
  begin
    XmlNode := ParXmlNode.FirstChild;
    while (XmlNode <> nil) do begin
      TreeNode := TN.AddChild(ParTreeNode, '');
      AssignNodeToTreeNode(XmlNode, TreeNode);
      AddChildNodes(XmlNode, TreeNode);
      XmlNode := XmlNode.NextSibling;
    end;
  end;
begin
  if Dest is TTreeNodes then begin
    TN := TTreeNodes(Dest);
    TV := TTreeView(TN.Owner);
    TV.SortType := stNone;
    TV.ReadOnly := True;
    if TV.Images = nil then
      TV.Images := TCustomImageList.Create(TV);
    TV.Images.Clear;
    TV.Images.GetResource(rtBitmap,
      'XMLTREEVIEWNODES', 0, [], 0);
    TV.Images.BkColor := clBlack;
    TN.BeginUpdate;
    TreeNode := TN.AddChild(nil, '');
    AssignNodeToTreeNode(Self, TreeNode);
    AddChildNodes(Self, TreeNode);
    TN.EndUpdate;
  end else
    inherited AssignTo(Dest);
end;
procedure TXmlDDocument.AssignNodeToTreeNode(XmlNode:
  TXmlDNode; TreeNode: TTreeNode);
begin
  case XmlNode.NodeType of
    xntDocument:
    begin
      TreeNode.Text := 'XML Document';
      TreeNode.ImageIndex := 0;
      AssignAttrNodesToTreeNodes(XmlNode, TreeNode);
    end;
    xntElement:
    begin
      TreeNode.Text := XmlNode.NodeName;
      TreeNode.ImageIndex := 2;
      AssignAttrNodesToTreeNodes(XmlNode, TreeNode);
    end;
    xntText:
    begin
      TreeNode.Text := XmlNode.NodeValue;
      TreeNode.ImageIndex := 4;
    end;
    xntCDATASection:
    begin
      TreeNode.Text := XmlNode.NodeValue;
      TreeNode.ImageIndex := 5;
    end;
    xntComment:
    begin
      TreeNode.Text := XmlNode.NodeValue;
      TreeNode.ImageIndex := 6;
    end;
  end;
  TreeNode.SelectedIndex := TreeNode.ImageIndex;
  TreeNode.Data := XmlNode;
end;
procedure TXmlDDocument.AssignAttrNodesToTreeNodes(
  ParXmlNode: TXmlDNode; ParTreeNode: TTreeNode);
var
  I:  Integer;
  S:  String;
  TreeNode: TTreeNode;
  XSN: TXmlDStructureNode;
begin
  XSN := ParXmlNode as TXmlDStructureNode;
  for I := 0 to (XSN.FAttrList.Count - 1) do begin
    S := StringReplace(XSN.FAttrList.List.Strings[I],
      '=', '="', []) + '"';
    TreeNode := ParTreeNode.Owner.AddChild(ParTreeNode, S);
    if ParXmlNode.NodeType = xntDocument then
      TreeNode.ImageIndex := 1
    else
      TreeNode.ImageIndex := 3;
    TreeNode.SelectedIndex := TreeNode.ImageIndex;
    TreeNode.Data := XSN.FAttrList;
  end;
end;
```

follows. Assign is a virtual method, which is called on the target object and is passed a reference to the source object. The class's Assign may know how to service the request, in which case it does so and we're done. Otherwise it calls its inherited Assign which goes through the same process.

If no inherited Assign steps forward to take responsibility, we end up at TPersistent.Assign. It causes an exception to be raised indicating no assignment compatibility exists. But it doesn't do so directly. Instead, it calls the virtual method AssignTo on the source object, which gets the target object as a parameter. If the source object's class knows how to transfer its contents to the target, it does so. If not, we go down the inheritance chain of AssignTo. If we end up in TPersistent.AssignTo, then the exception is finally raised there.

In other words, we can effectively extend an Assign target class by providing the assignment processing in our own class. As Listing 3 shows, that's what's happening in the AssignTo method of TXmlDDocument.

The navigation of the document structure should start to be looking familiar by now. We use the FirstChild and NextSibling properties to navigate through the document structure. When at an element node, we first pick up any attributes and make tree view nodes of them.

To facilitate finding our way back into the document from a tree view node, the node reference is added to the tree node's Data property (thanks to Mark Chambers, my Australian correspondent and informal beta tester of this code, for that suggestion). For an attribute 'node', there's nothing to point back to, so instead we record the AttrList reference in the Data property.

Figure 1 shows the results of this operation. The document shown is the same one built with the sample code in part one of this article. I'll take responsibility for the graphics here, and I'll be the first one to admit that the images for prolog pseudo-attributes and regular element attributes are a little lame.

Can We Quote You?

Before putting this to bed, there's one more small improvement we can add to the framework to make it more bulletproof. Recall from part one, I spoke of five predefined character entities in XML that could be used instead of characters which could confuse the parser. These were <, >, ", ' and &, which respectively stand in for less-than, greater-than, quote, apostrophe and ampersand.

Some of these characters are more 'dangerous' than others. A less-than character in open text is a real parse wrecker. Upon finding one of these, the parser thinks an element has been encountered. Within attributes values (which may be enclosed in either quotes or apostrophes) a quoting character that's the same one used to enclose the value will make the parser assume it's found the end of the value string.

So why is this our problem? If an application is going to be using these sorts of characters, why not make it responsible for supplying the character entity substitutions itself? The answer is that the application could, in fact, do so. But the next time the document is parsed, the character entities will be replaced by the characters being represented. When using an XML stream for persistent storage or for some kind of work flow application, we'll have dodged the bullet only the first time the document is parsed. If we write out what was presented to us by the parser, then the next party to have the document parsed will be out of luck.

So, let's see what we can do to remedy this potential pitfall. One solution would be to just blindly plow through the text node and attribute values and force a substitution. The only problem there is that we might then disrupt some delicately planned and executed strategy wherein the application *is* taking on this responsibility. We would not like to encounter an entity reference (which would start with a real ampersand) and ruin it by turning that ampersand into a character entity. So what can we do?

The approach I finally decided upon makes the assumption that most applications won't be using entity references. If there's the occasional problem character in the stream, we can let the class framework deal with it and not bother the client with details. Furthermore, we'll only concern ourselves, by default, with three characters.

First of all we'll ensure that quote characters, which are used exclusively for enclosing attribute values on output, have substitutions made. But we'll only do this for attribute values. Secondly, we'll deal with less-than characters, but only in text node values. Finally, we'll substitute ampersands in either attribute values or general text.

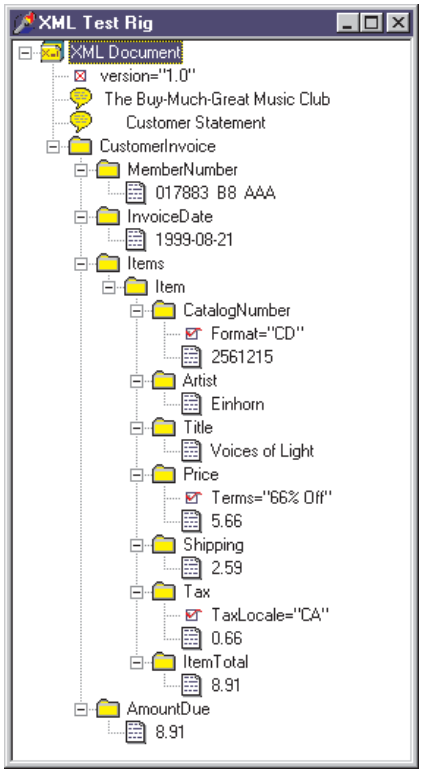
But we need to offer a way for a client to bend those rules. To that end, the framework provides a type declaration for `CharEntity` (refer back to Listing 1), and two properties: `AttrCharEntities` and `TextCharEntities`. Their values default in accordance with what was described in the preceding paragraph.

Then we'll do one more thing. We'll provide two events on `TXmlDDocument` which may be assigned handlers to cater to special situations. `OnOutputAttrValue` and `OnOutputText` can be used to override any default handling by the class output facilities. The handlers, both of type `TSubstituteCharEntities`, supply a `var` parameter for the value, and another Boolean var parameter available to indicate the value should be left as is. `Sender` is either the `TXmlDText` node reference or a `TXmlDAttrList` reference as appropriate. You may notice I also added an `OwnerNode` property to `TXmlDAttrList` for use in situations like this.

Out, DOM Spot!

Before wrapping up, I need to mention two observations about behavior I've discovered in the late-stage testing of this code. Both have to do with `LoadFromStream` processing. `LoadFromStream` calls the `IXMLDOMDocument.loadXml` method to which a string parameter is passed (as opposed to the `IXMLDOMDocument.load` method called by `LoadFromFile`). It turns out that `loadXml` will perform no validation processing, even when requested and when the document contains an external DTD reference. I have no way of determining whether this is an msxml bug or is intentional. Not knowing which is the case, I've left the `ValidateOnParse` parameter in place, even though it has no effect at the moment.

A second surprise at least has a logical rationale. If one passes XML having `encoding="UTF-8"` (in the XML prolog) to `LoadFromStream`, the msxml parser objects. UTF-8 specifies that the document uses 8-byte characters. But COM demands WideStrings for parameters. Delphi causes a conversion to `WideString` when setting up



➤ *Figure 1*

the `loadXML` call, and the parser reasonably objects. The problem can be circumvented by either not using the encoding clause in the prolog or by using `LoadFromFile`, for which no problems are reported by the parser.

So, are we finally finished here? Actually, there are a few more nice features I'd like to incorporate into this class framework, and perhaps we'll do some of them in a future article.

But Delphi 5 is expected to arrive on my doorstep any day now, and it will be a whole new day. I think that for the moment the best strategy is to assume a posture that's... ahem... extensible.

David Baer is Chief Software Architect at Spear Technologies in San Francisco. His least favorite Wagner opera is Parsifal, and he'd be quite annoyed to have to write a validating XML parser. Coincidence? He can be reached at dbaer@speartechnologies.com