XML In Delphi, 3

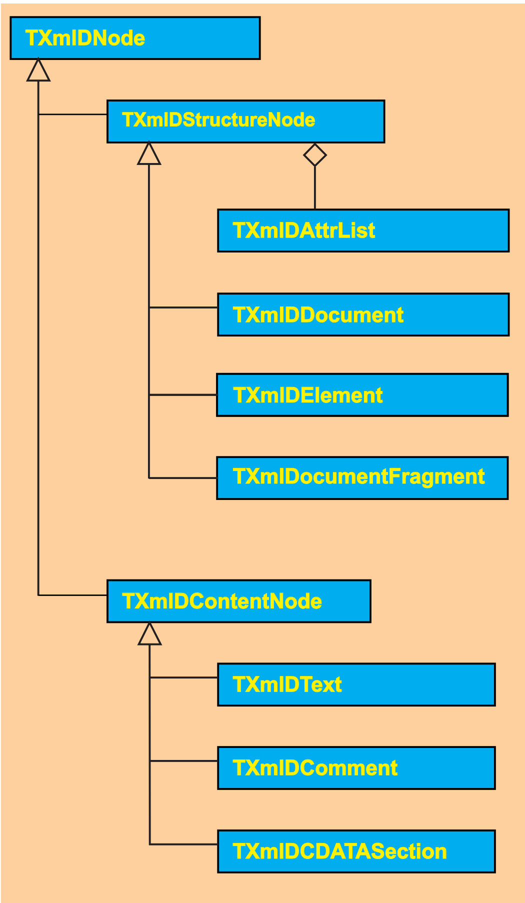
by David Baer

In issues 48 and 50 (August and October of 1999) we began to explore the topic of XML in Delphi by building a class framework with which to load, manipulate and write XML documents. We'll continue here by adding additional features to the framework. We'll add support for one Document Object Model node type absent from the initial implementation, and we'll add a variety of other capabilities to make the life of users of the framework a little bit easier.

Before getting down to the business at hand, let me thank Christian Zietz for reporting a bug in the code from the last instalment. Christian alertly spotted the fact that the MSXML implementation provides a document property, `async`, that has a default value of `True` (recall, MSXML is the Microsoft DLL housing their XML services). This property designates whether or not a parse/load operation must fully complete before returning control to the client process. The code presented in the article did nothing to override this default, and how the demonstration code managed to work (and on more than one machine, on more than one continent!) remains a mystery. The `TXmlDDocument` method `LoadFromFile` has been corrected to set this property to `False`. Also, in preparing the code for this instalment, I spotted another problem relating to setting node parentage under certain circumstances that has been resolved.

Also before commencing, I need to offer one small disclaimer. Due to a miscommunication between *Our Esteemed Editor* and myself (the error being entirely mine), I've learned that my deadline for submission of this piece is much sooner than anticipated. The code shown here, the entirety of which is contained on this issue's disk, works correctly for the usage examples. But it has not been tested to my full satisfaction as of the submission date, and I do plan to do some additional testing. If you plan to copy the code to your machine for any purpose other than a casual perusal, do please check *The Delphi Magazine* website for any corrective updates.

➤ *Figure 1*



A Fragment Of The Imagination

The DOM framework provides a node type which we didn't bother to implement in the earlier instalments. The *document fragment* node type is useful for doing copy, cut and paste operations when manipulating an XML document in place. It provides a convenient node under which to organize sub-structures as they're being built or modified.

The first article in this series discussed the inheritance structure used in the framework. It may be valuable to briefly review that structure, shown in Figure 1. All the concrete node types inherit from a common base class, `TXmlDNode`. Two intermediate classes, `TXmlDStructureNode` and `TXmlDContent` node, are provided to factor out child node management responsibilities for only those concrete class types that actually need them. As you might expect, the `TXmlDDocumentFragment` class derives from `TXmlDStructureNode`.

The document fragment may be used as a parent for elements, text segments, and so forth. You can think of it as a temporary element node that lives outside of any document. In fact, the document fragment itself can never be included (ie, inserted or appended) anywhere in a document. And this is what makes it somewhat useful.

If one needs a temporary parent node, one could certainly create an instance of an element or even a document and just use that. But the document fragment knows a trick that's not in the repertoire of other node types. When a document fragment is involved in an insert or append operation, it discreetly removes itself from the picture. Only its child nodes are actually inserted into the destination.

Actually, the document fragment merging 'intellegence' doesn't live in its own class methods. Instead the `TXmlDNode.InsertBefore` and `TXmlDStructureNode.AppendChild` routines are responsible for providing the special treatment. There's one wrinkle here that we need to address. The nodes in our framework cannot be children of multiple parents. If nothing else, we rely on propagated destruction of children during parent destruction. So the document fragment merging process also removes the child nodes from itself during insertion operations. At the end of this process, the document fragment will be empty.

Listing 1 shows the additions and modifications to the framework declarations made for this instalment. There's nothing particularly tricky in the implementation of `TXmlDDocumentFragment`. Like the other concrete node types, it supplies overridden `Create` and `CloneNode` methods. It also provides an empty override to the abstract `WriteToStream` method. This node can never be part of an actual document and cannot be called upon to output its contents.

As a `TXmlDStructureNode` derivative, `TXmlDDocumentFragment` has access to the `AttrList` property, which can be used to associate attribute values with instances of this node type. But, in this case, that property has no practical value. Attribute values provided will not be copied when the document fragment is inserted into another structure.

Elementary, My Dear Watson

The next new feature provides several easy mechanisms for accessing child elements. Let me just restate an assumption I've been

➤ *Listing 1*

```
type
  TXmlNodeType = (xntDocument, xntElement,
    xntDocumentFragment, xntText, xntComment,
    xntCDATASection);
const
  XmlNodeNames: array[xntDocument..xntCDATASection] of
    String = ('#document', '', '#document-fragment',
    '#text', '#comment', '#cdata-section');
type
  . . .
  TXmlDDocumentFragment = class;
  . . .
  TXmlDElementIterator = class;
  TXmlDElementPattern = class;
  TXmlDNode = class(TPersistent)
    private
      . . .
      FTag:            Integer;
      FLevel:          Integer;
    protected
      . . .
      procedure SetLevel(Lvl: Integer);
      . . .
      procedure SetNodeName(const Value: TXmlName); virtual;
      procedure InsertDocFragBefore(NewNode:
        TXmlDDocumentFragment; ThisNode: TXmlDNode);
      procedure SetParent(ParentNode: TXmlDStructureNode);
      . . .
    public
      . . .
      procedure ZeroAllTags;
      . . .
      property Level: Integer read FLevel;
      property Tag: Integer read FTag write FTag;
  end;
  TXmlDStructureNode = class(TXmlDNode)
    private
      . . .
      FElementCount:   Integer;
    protected
      procedure AppendDocFragChild(NewNode:
        TXmlDDocumentFragment);
      procedure AssignAttrNodesToTreeNodes(ParXmlNode:
        TXmlDNode; ParTreeNode: TTreeNode);
      procedure AssignNodeToTreeNode(XmlNode: TXmlDNode;
        TreeNode: TTreeNode);
      . . .
      function GetElementByName(const Name: String):
        TXmlDElement;
      function GetElements(Index: Integer): TXmlDElement;
      . . .
    public
      . . .
      procedure AssignTo(Dest: TPersistent); override;
      . . .
      function GetFirstChildElement: TXmlDElement;
      . . .
      property ElementCount: Integer read FElementCount;
      property Elements[Index: Integer]: TXmlDElement
        read GetElements;
      property ElementByName[const Name: String]:
        TXmlDElement read GetElementByName; default;
    end;
  TXmlDContentNode = class(TXmlDNode)
    private
      FValue: String;
    protected
      function GetNodeValue: String; override;
      procedure SetNodeValue(const Value: String); override;
  end;
  TXmlDDocument = class(TXmlDStructureNode)
    . . .
    public
    . . .
      function CreateDocumentFragment:
        TXmlDDocumentFragment;
      . . .
  end;
  TXmlDElement = class(TXmlDStructureNode)
    . . .
    protected
      . . .
      function GetAsBoolean: Boolean;
      function GetAsCurrency: Currency;
      function GetAsDate: TDateTime;
      function GetAsDateTime: TDateTime;
      function GetAsInteger: Integer;
      function GetAsString: String;
      function GetAsTime: TDateTime;
      function GetFirstTextNodeValue: String;
      procedure SetAsBoolean(const Value: Boolean);
      procedure SetAsCurrency(const Value: Currency);
      procedure SetAsDate(const Value: TDateTime);
      procedure SetAsDateTime(const Value: TDateTime);
      procedure SetAsInteger(const Value: Integer);
      procedure SetAsString(const Value: String);
      procedure SetAsTime(const Value: TDateTime);
      . . .
      procedure SetFirstTextNodeValue(const Value: String);
      . . .
    public
      . . .
      function GetNextSiblingElement: TXmlDElement;
      property AsBoolean: Boolean read GetAsBoolean
        write SetAsBoolean;
      property AsCurrency: Currency read GetAsCurrency
        write SetAsCurrency;
      property AsDate: TDateTime read GetAsDate
        write SetAsDate;
      property AsDateTime: TDateTime read GetAsDateTime
        write SetAsDateTime;
      property AsInteger: Integer read GetAsInteger
        write SetAsInteger;
      property AsTime: TDateTime read GetAsTime
        write SetAsTime;
      property AsString: String read GetAsString
        write SetAsString;
  end;
  TXmlDDocumentFragment = class(TXmlDStructureNode)
    protected
      procedure WriteToStream(Stream: TStream;
        FormattedForPrint: Boolean); override;
    public
      constructor Create;
      function CloneNode(RecurseChildren: Boolean):
        TXmlDNode; override;
  end;
  TXmlDElementIterator = class(TObject)
  private
    CurrNode: TXmlDStructureNode;
    RootNode: TXmlDStructurenode;
    Position: TList;
    ElementPattern: TXmlDElementPattern;
  protected
    function NextElementInPattern: TXmlDElement;
    function NextElementInStructure: TXmlDElement;
  public
    constructor Create(ContextNode: TXmlDStructureNode =
      nil; const Pattern: String = '');
    destructor Destroy; override;
    function Next: TXmlDElement;
    procedure Reset(ContextNode: TXmlDStructureNode = nil;
      const Pattern: String = '');
  end;
  TElementPatternMatch = (epmNoMatch, epmPathMatch,
    epmEndMatch);
  TXmlDElementPattern = class(TObject)
  private
    RootNode: TXmlDStructureNode;
    PatternPieces: TStringList;
    PatternLevels: Integer;
  protected
    procedure ParsePattern(const Pattern: String);
  public
    constructor Create(ContextNode: TXmlDStructureNode;
      const Pattern: String);
    destructor Destroy; override;
    function PatternMatchType(ELement: TXmlDElement):
      TElementPatternMatch;
  end;
```

using in designing these classes. This framework is designed for use with program-to-program data transmissions. In this context, the document structures will normally be quite straightforward. In particular, element nodes will usually be one of the following: they will parent other element nodes (and nothing else), they will parent a single text segment node, or they will parent nothing (being used to exclusively provide attribute values).

With that in mind, I felt it would be useful to provide a property that would make access to child elements straightforward. As any node type capable of hosting child nodes would benefit from this, the property is introduced in `TXmlDStructureNode`. Actually, there are two properties added for this purpose: `Elements` and `ElementByName`.

`Elements` takes an `integer` index parameter. An additional property, `ElementCount`, provides just what its name implies. The count value is maintained as a byproduct of the various insert and removal methods of the base classes.

`ElementByName` takes a `string` parameter that identifies the element node name of interest. If multiple child nodes bearing the same name exists, it returns only the first. An exception is raised where no element with the specified name is present. I suspect that `ElementByName` would be more frequently used in day-to-day coding than `Elements`, so it is defined as the default property.

You may wonder why these properties, differing only in parameter type, have one singular and one plural name. For the answer, you'd need to track down the Delphi engineer who came up with `Fields` and `FieldByName`. I'm just trying to go with the flow here.

As You Like It

With the two previous properties, we now have an easy way to get references to the elements of interest in our XML documents. What else can we do to make our code more concise and clear? How about a mechanism that could allow us to reference the content of a single text node belonging to an element? What's more, how about providing some convenient type conversions while we're at it?

A means for doing this has plenty of precedent. Read-write properties like `AsString`, `AsDate`, etc are well known to most Delphi practitioners, and they seem to be just what's called for here as well. But we'll adapt them a little from their conventional use.

To begin with, where's the best place to provide them? You could make a case for making them properties of text nodes, in which case their function would be just one of type conversion (string to date, etc). If, instead, we move them up a level and make them properties of the element, they can do just a bit more work (and save class users some coding). Once we have a reference to an element node, we need not acquire a reference to its text node to read or set the text value. We just use the `AsXxxx` properties instead.

The content of a text segment is, of course, always a string. So the `AsString` property performs no conversion. Two others in this property group, `AsCurrency` and `AsInteger`, perform conversions using Delphi's standard service functions: `StrToInt`, *et al.* The resultant string representations of these data types is consistent with the standard ISO representations preferred in XML numeric formatting.

For `Boolean` values, the standard calls for a string value of `0` for `False` and `1` for `True`. So, the `AsBoolean` is a trivial conversion requirement. For date/time values, however, we've got a bit more work to do. Delphi doesn't (to my knowledge anyway) offer canned routines for converting an ISO formatted date or time value. We'll need to 'can' one ourselves, and fortunately this isn't all that difficult.

An example of an ISO compliant date/time string value is `2000-01-31T15:30:00`. The hyphens are not required, and a two-digit year is even acceptable (of course anyone still using just two digits for years should be subjected to highly public ridicule). Like dates, there are several variations on time values. The one shown above specifies only seconds, but finer-grained values are possible. For further information on standard data format types used in conjunction with XML schemas, you may wish to take a look at the W3C proposal for XML Data, which discusses this subject in some detail. You should be able to find it at the location www.w3.org/TR/1998/NOTE-XML-data-0105/.

Several conversion service routines are present in the XML classes unit to enact all of this: `ISOStrToDate`, `ISOStrToTime` and `ISOStrToDateTime` do the more difficult string decoding. They do not accommodate all the possible formats, but offer a serviceable capability which should be adequate for many situations. Converting date/time values to string is quite a bit easier, relying on the old Delphi standby `FormatDateTime` routine to do the grunt work.

Tag, You're It

Before getting to the pick of the litter of these enhancements, let's quickly look at two small but useful extensions. I'm somewhat cynical when it comes to the common `TComponent Tag` property. These can be misused in all kinds of ways. Nevertheless, I was looking at the MSXML-provided DOM services recently, and wished mightily for a simple four-byte node variable to allow me to associate objects with nodes. As you might guess, no such thing exists.

Taking pity upon others who might want the same of the framework presented here, I decided it would do no harm to add a `Tag` property at the node level. For good measure, I've also included a node method `ZeroAllTags` that will cause the `Tag` values of a node and of all its children to be cleared.

On another subject, you may recall that in the last instalment, I provided a means to assign an XML document to a `TTreeView.Items` property. It occurred to me later that there was no reason to limit the assignment to an entire document. It might be appropriate at times to assign a sub-portion of a document for visual display.

To accommodate this, I've moved the `AssignTo` processing out of the `TXmlDDocument` class, and placed it in `TXmlDStructureNode`. Since `TXmlDDocument` derives from `TXmlDStructureNode`, it retains the treeview assignment capability. This capability now also extends to `TXmlDElement` and `TXmlDDocumentFragment` instances.

Walking The Walk

One of the more powerful features in MSXML is the ability to acquire a list of elements using a pattern specification. The pattern provides a template for specifying which nodes are of interest and also provides a means of filtering the result set based on attributes values, for example. On the whole this is an extremely flexible capability. It is actually a feature that is frequently discussed in conjunction with the Extensible Stylesheet Language (XSL). It provides the input selection mechanism to feed XSL transformations. But it has potential uses outside that context as well.

I had contemplated extending the Delphi XML class framework in a similar fashion. Unfortunately, the more I studied the pattern selection capabilities in MSXML, the more I realized that the job was not one to be undertaken lightly. However, a more modest pattern selection service could still be eminently useful, and that's what we'll look at next.

Two new classes were defined to serve this goal: `TXmlDElementIterator` and `TXmlDElementPattern`. Refer to Listing 1 to see their declarations.

`TXmlDElementIterator` actually serves two purposes. It can be used without any pattern to traverse all child element nodes of a `TXmlDStructure` node, or a pattern may be provided to limit the elements returned. In both modes, modifications to the document or element structure will not disrupt the iteration, provided the last-obtained node is not destroyed prior to the next `Next` invocation. Resetting the iterator, possibly switching patterns or start nodes in the process, is done with the `Reset` method.

In the first mode, where no pattern is used, one first creates an instance of an element iterator, passing a `TXmlDStructure` node instance, which specifies the starting point, for the required parameter. Thereafter, calls to the `Next` method will return references to child element nodes at all levels of the subordinate structure. The elements returned in this process will be in the same order as they would be seen in the structure as represented in XML. When no element nodes remain, `Next` returns a `nil` value. The `Next` function in `TXmlDElementIterator` calls one of two internal methods to find the next node. For the no-pattern mode, this navigation is fairly straightforward, as can be seen in the `GetNextElementInStructure` method, shown in Listing 2.

Navigation under the direction of a pattern is a rather more complicated affair, which I'll describe shortly. The patterns supported by this implementation consist of element names for each level, or a * wildcard. The pattern is specified as a single string, in which the levels are separated by a / (slash) character. Within each level, multiple element names may be supplied, these being separated by a | (pipe) character (we'll examine several examples shortly). The first level in the pattern represents the child element level of the start element. The iterator does not return the start node itself. Although the pattern string isn't rigorously parsed, spaces may be included between the names and separators for clarity.

Let's consider a few examples. The following pattern will cause all third level elements under the start node to be returned: * / * / *. To obtain all `Price` elements under `OrderItem` elements, under `Order`, the pattern `Order / OrderItem / Price` will get the job done.

Expanding that last example even further, if we wanted to see both `Price` and `Quantity` elements at the third level, the pattern would look like:

➤ Facing page: Listing 2

```
Order / OrderItem / Price |
  Quantity
```

Although this scheme provides a fair amount of flexibility, you'll notice that we're limited to having elements at one level only returned for any pattern directed iteration.

The pattern parsing and matching is implemented as a separate class, `TXmlDElementPattern`. Although neither of these two classes is very big, iteration under control of a pattern is a somewhat tricky business, so breaking those responsibilities into two classes helps manage the complexity.

But there's an even better reason to do this. If we regard this as an application of the Gang of Four *strategy* pattern, you can see that a more sophisticated pattern 'language' can be installed at a later date by supplying an improved `TXmlDElementPattern` implementation. No changes to `TXmlDElementIterator` should be needed to introduce the upgrade.

In fact, I've included a bit of redundancy in the `TXmlDElementIterator` method `NextElementInPattern` method in anticipation of this possibility. The way things stand right now, we know that we need not look at any nodes a level lower than that in the pattern. But a more sophisticated approach might return elements from differing levels. So we allow a small amount of optimization to be sacrificed to leave that door open.

The pattern iteration technique works as follows. For the first `Next` call, things aren't too complicated. We iterate through the structure in the same manner that we do with no pattern. However, when we reach an element that's not in the path, we proceed back to the parent as if it had no children until we have an end node match or have traversed the entire structure.

Subsequent `Next` calls are a little trickier. We know that the current element matches the pattern, and therefore all parent elements are

```
{ TXmlElementIterator }
constructor TXmlDElementIterator.Create(
  ContextNode: TXmlDStructureNode; const Pattern: String);
begin
  inherited Create;
  Position := TList.Create;
  RootNode := ContextNode;
  CurrNode := ContextNode;
  if Pattern <> '' then
    ElementPattern :=
      TXmlDElementPattern.Create(RootNode, Pattern);
end;
destructor TXmlDElementIterator.Destroy;
begin
  ElementPattern.Free;
  Position.Free;
  inherited Destroy;
end;
function TXmlDElementIterator.Next: TXmlDElement;
begin
  Result := nil;
  if CurrNode = nil then Exit;
  if ElementPattern = nil then
    Result := NextElementInStructure
  else
    Result := NextElementInPattern;
end;
function TXmlDElementIterator.NextElementInPattern:
  TXmlDElement;
  function GetFirstElementInPattern(
    StartNode: TXmlDStructureNode): TXmlDElement;
  var CandidateElement: TXmlDElement;
  begin
    Result := nil;
    if StartNode.ElementCount > 0 then begin
      CandidateElement := StartNode.GetFirstChildElement;
      while ((CandidateElement<>nil) and (Result=nil))
      do begin
        case ElementPattern.PatternMatchType(
          CandidateElement) of
          epmEndMatch  : Result := CandidateElement;
          epmPathMatch : Result :=
            GetFirstElementInPattern(CandidateElement);
        end;
        if Result = nil then
          CandidateElement :=
            CandidateElement.GetNextSiblingElement;
      end;
    end;
  end;
  function GetNextElementInPattern:TXmlDElement;
  var CandidateElement: TXmlDElement;
    function GetNextCandidate(StartNode:
      TXmlDStructureNode): TXmlDElement;
    begin
    Result := TXmlDElement(StartNode).GetNextSiblingElement;
      if Result = nil then begin
        if StartNode.ParentNode <> RootNode then begin
          Result := TXmlDElement(
            StartNode.ParentNode).GetNextSiblingElement;
          if Result = nil then
            Result :=
              GetNextCandidate(StartNode.ParentNode);
        end;
      end;
    end;
  begin
    Result := GetFirstElementInPattern(CurrNode);
    if Result <> nil then Exit;
    CandidateElement := GetNextCandidate(CurrNode);
    while ((CandidateElement <> nil) and (Result = nil))
    do begin
      if CandidateElement <> nil then begin
        case ElementPattern.PatternMatchType(
          CandidateElement) of
          epmEndMatch  : Result := CandidateElement;
          epmPathMatch : Result :=
            GetFirstElementInPattern(CandidateElement);
        end;
        if Result = nil then
          CandidateElement :=
            GetNextCandidate(CandidateElement);
      end;
    end;
  end;
begin
  if CurrNode = RootNode then
    Result := GetFirstElementInPattern(RootNode)
  else
    Result := GetNextElementInPattern;
  CurrNode := Result;
end;
function TXmlDElementIterator.NextElementInStructure:
  TXmlDElement;
begin
  if CurrNode = RootNode then
    Result := RootNode.GetFirstChildElement
  else begin
    if CurrNode.ElementCount > 0 then begin
      Result := CurrNode.GetFirstChildElement;
    end else begin
      Result :=
        TXmlDElement(CurrNode).GetNextSiblingElement;
      if Result = nil then begin
        while (Result = nil) do begin
          CurrNode := CurrNode.ParentNode;
          if CurrNode = RootNode then Break;
          Result :=
            TXmlDElement(CurrNode).GetNextSiblingElement;
        end;
      end;
    end;
  end;
  CurrNode := Result;
end;
procedure TXmlDElementIterator.Reset(ContextNode:
  TXmlDStructureNode; const Pattern: String);
begin
  RootNode := ContextNode;
  CurrNode := ContextNode;
  ElementPattern.Free;
  ElementPattern := nil;
  if Pattern <> '' then
    ElementPattern :=
      TXmlDElementPattern.Create(RootNode, Pattern);
end;
{ TXmlDElementPattern }
constructor TXmlDElementPattern.Create(Contextnode:
  TXmlDStructureNode; const Pattern: String);
begin
  inherited Create;
  RootNode := ContextNode;
  PatternPieces := TStringList.Create;
  ParsePattern(Pattern);
end;
destructor TXmlDElementPattern.Destroy;
begin
  PatternPieces.Free;
  inherited Destroy;
end;
procedure TXmlDElementPattern.ParsePattern(
  const Pattern: String);
var
  I: Integer;
  Lvl: Integer;
  S: String;
  procedure ParsePatternLevel(const Pattern: String);
  var
    I: Integer;
    S: String;
  begin
    S := Pattern;
    while (S <> '') do begin
      I := Pos('|', S);
      if I > 0 then begin
        PatternPieces.AddObject(Trim(Copy(S, 1, (I - 1))),
          Pointer(Lvl));
        S := Copy(S, (I + 1), $7FFF);
      end else begin
        PatternPieces.AddObject(Trim(S), Pointer(Lvl));
        S := '';
      end;
    end;
  end;
begin
  PatternPieces.Clear;
  S := Pattern;
  Lvl := 0;
  while (S <> '') do begin
    Inc(Lvl);
    PatternLevels := Lvl;
    I := Pos('/', S);
    if I = 0 then begin
      ParsePatternLevel(S);
      S := '';
    end else begin
      ParsePatternLevel(Copy(S, 1, (I - 1)));
      S := Copy(S, (I + 1), $7FFF);
    end;
  end;
end;
function TXmlDElementPattern.PatternMatchType(Element:
  TXmlDElement): TElementPatternMatch;
var
  I: Integer;
  Lvl: Integer;
begin
  Result := epmNoMatch;
  Lvl := Element.Level - RootNode.Level;
  if Lvl > PatternLevels then Exit;
  for I := 0 to (PatternPieces.Count - 1) do begin
    if (Integer(PatternPieces.Objects[I]) = Lvl) and
      ((PatternPieces[I] = '*') or
      (PatternPieces[I] = Element.NodeName)) then begin
      Result := epmPathMatch;
      Break;
    end;
  end;
  if (Result = epmPathMatch) and (Lvl = PatternLevels) then
    Result := epmEndMatch;
end;
```

in the path. We next look at child elements. For the present pattern capability, these will never be matches, but that could change with an improved pattern class. Next we look at sibling elements (again, we know parent elements are path matches). If an end node match is found, we're done. If a path match is found, we start examining its child elements. When no siblings are left, we back up a level, and check siblings there. If no path match is found, we go back another level, and so on.

The TXmlDElementPattern class parses the pattern, placing the element names in a TStringList along with the level to which they correspond. The matching processing, as seen in the PatternMatchType method is straightforward. However, we are interested in more than a 'matches/doesn't-match' answer. We need to know we've found a prospective parent-to-matching-node, an end-node match, or a non-match. So this function returns a 'matches-as-end-node / matches-as-path-node/ doesn't-match' result.

Let's Pretend

So, let's put it all together (well, much of it anyway) with a down-to-earth practical example. In this, we'll build a routine to transform an XML order document, like that shown in Listing 3, into a billing document, displayed in the treeview of Figure 2.

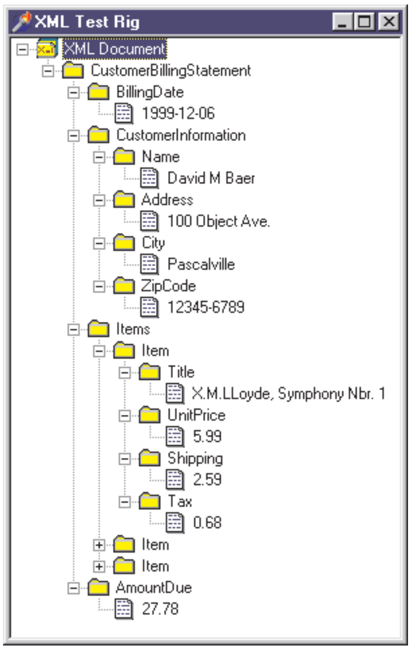
One thing you may immediately notice about the code is that it appears to be awfully 'trusting' that the input data is valid: child elements appear where they're

➤ *Listing 3*

```
<?xml version="1.0"?>
<Order>
  <CustomerNumber>1123A456B</CustomerNumber>
  <OrderReceived>1999-12-02</OrderReceived>
  <CatalogEdition>NOV99-P</CatalogEdition>
  <Items>
    <Item>
      <CatalogNumber>227861</CatalogNumber>
      <Quantity>1</Quantity>
    </Item>
    <Item>
      <CatalogNumber>298662</CatalogNumber>
      <Quantity>1</Quantity>
    </Item>
    <Item>
      <CatalogNumber>214573</CatalogNumber>
      <Quantity>1</Quantity>
    </Item>
  </Items>
</Order>
```

➤ *Listing 4*

```
procedure TfrmTestRig.ProcessOrder;
var
  OrderDoc: TXmlDDocument;
  Order: TXmlDElement;
  Items: TXmlDElement;
  BillDoc: TXmlDDocument;
  Bill: TXmlDElement;
  WorkElement: TXmlDElement;
  CustFrag: TXmlDDocumentFragment;
  ItemsFrag: TXmlDDocumentFragment;
  CustomerNumber: String;
begin
  // get order information
  OrderDoc := TXmlDDocument.Create;
  OrderDoc.LoadFromFile('Order.xml');
  Order := OrderDoc['Order'];
  CustomerNumber := Order['CustomerNumber'].AsString;
  Items := Order['Items'];
  // set up bill document
  OrderTotal := 0.0;
  BillDoc := TXmlDDocument.Create;
  Bill := BillDoc.CreateElement('CustomerBillingStatement');
  BillDoc.AppendChild(Bill);
  // first, set up customer header data
  CustFrag := BillDoc.CreateDocumentFragment;
  PrepareCustomerInfo(BillDoc, CustomerNumber, CustFrag);
  // next process items ordered
  ItemsFrag := BillDoc.CreateDocumentFragment;
  ItemsFrag.AppendChild(BillDoc.CreateElement('Items'));
  PrepareItemsInfo(BillDoc, Items, ItemsFrag);
  // build billing document
  WorkElement := BillDoc.CreateElement('BillingDate');
  WorkElement.AsDate := Date;
  Bill.AppendChild(WorkElement);
  Bill.AppendChild(CustFrag);
  Bill.AppendChild(ItemsFrag);
  WorkElement := BillDoc.CreateElement('AmountDue');
  WorkElement.AsCurrency := OrderTotal;
  Bill.AppendChild(WorkElement);
  TV.Items.Assign(BillDoc);
  TV.FullExpand;
  // wrap it up
  CustFrag.Free;
  ItemsFrag.Free;
  OrderDoc.Free;
  BillDoc.Free;
end;
procedure TfrmTestRig.PrepareCustomerInfo(BillDoc:
  TXmlDDocument; const CustomerNumber: String;
  CustomerInfoFrag: TXmlDDocumentFragment);
var
  CustInfo: TXmlDElement;
begin
  CustInfo := BillDoc.CreateElement('CustomerInformation');
  CustomerInfoFrag.AppendChild(CustInfo);
  CustInfo.AppendChild(BillDoc.CreateElement('Name',
    'David M Baer'));
  CustInfo.AppendChild(BillDoc.CreateElement('Address',
    '100 Object Ave.'));
  CustInfo.AppendChild(BillDoc.CreateElement('City',
    'Pascalville'));
  CustInfo.AppendChild(BillDoc.CreateElement('ZipCode',
    '12345-6789'));
end;
procedure TfrmTestRig.PrepareItemsInfo(BillDoc:
  TXmlDDocument; Items: TXmlDElement; ItemsFrag:
  TXmlDDocumentFragment);
var
  ItemIterator: TXmlDElementIterator;
  Item: TXmlDElement;
  WorkElement: TXmlDElement;
  Quantity: Integer;
  Price: Currency;
  Tax: Currency;
  Shipping: Currency;
  procedure PrepareItem(Item: TXmlDElement);
  var
    OutItem: TXmlDElement;
  begin
    OutItem := BillDoc.CreateElement('Item');
    OutItem.AppendChild(BillDoc.CreateElement('Title',
      'X.M.LLoyde, Symphony Nbr. ' +
      Copy(Item['CatalogNumber'].AsString, 6, 6)));
    Quantity := Item['Quantity'].AsInteger;
    Price := 5.99 * Quantity;
    Shipping := 2.59 * Quantity;
    Tax := (Price + Shipping) * 0.08;
    Tax := Trunc(Tax * 100.0) / 100.0;
    WorkElement := BillDoc.CreateElement('UnitPrice');
    WorkElement.AsCurrency := Price;
    OutItem.AppendChild(WorkElement);
    WorkElement := BillDoc.CreateElement('Shipping');
    WorkElement.AsCurrency := Shipping;
    OutItem.AppendChild(WorkElement);
    WorkElement := BillDoc.CreateElement('Tax');
    WorkElement.AsCurrency := Tax;
    OutItem.AppendChild(WorkElement);
    ItemsFrag.Elements[0].AppendChild(OutItem);
    OrderTotal := OrderTotal + Price + Shipping + Tax;
  end;
begin
  ItemIterator :=
    TXmlDElementIterator.Create(Items, 'Item');
  Item := ItemIterator.Next;
  while Item <> nil do begin
    PrepareItem(Item);
    Item := ItemIterator.Next;
  end;
  ItemIterator.Free;
end;
```



➤ *Figure 2*

supposed to, dates and numbers have valid formats, and so forth. In real life, it would be appropriate that this input would be validated by the parser against a schema declaration. Doing so would, in fact, allow the parser to ensure that the document's content complies with our expectations.

The example does not actually use schema validation, but I've written the code as if it does, because the requisite checking would do little to illuminate the real purpose of this example. This is some of the 'pretend' aspect. The other 'pretend' part is that some service functions that provide, for example, name and address information based on a customer account number, are just dummy routines that supply invariant values. Again, a realistic implementation would not add to what's being demonstrated.

Here's what we need. The input order is an example of a complete order. At least one element of each type shown will be present in all orders. First, we need to supply a `BillingDate` element, followed by a `CustomerInformation` element containing name and address elements. Next, we'll process the `Item` elements in the input document, and produce composite elements with price, shipping and tax charges, as well as showing the item title. Lastly , we'll produce an `AmountDue` element showing the total charges.

Listing 4 contains the code used to do this. You'll notice that we're using document fragments to build up the customer information and line item information prior to insertion into the billing document. You can also see the use of an element iterator for reading through the order items in the input document. I think you'll find this code quite straightforward.

End Game

Well, that's about all. This is likely to be the final instalment of this series and I regret having to say it, since developing these classes has been a truly enjoyable exercise. On the other hand, if you're finding them useful, and have some thoughts for additional enhancements, do send me your ideas!

David Baer is Chief Software Architect at Spear Technologies in San Francisco. As a recovering woodworkaholic, he was rather dismayed to learn that amazon.com is now selling power tools. He can be contacted at dbaer@speartechnologies.com