

XML Support In Visual Basic 9

Erik Meijer*

Brian Beckman†

XML Programming Using DOM

Programming against XML using the DOM API today is a bitch. The accidental complexity of working with the DOM is so high that many programmers are giving up on using XML altogether, cursing the hype that XML makes dealing with data simple, which no one who has actually written DOM code could claim. The W3C DOM was not designed with ease of programming in mind, but rather evolved as a design by committee from the existing DHTML object model originally created by Netscape.

Consider the following simple purchase order document

```
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    ...
  </billTo>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <price>148.95</price>
    </item>
    ...
  </items>
</purchaseOrder>
```

*Erik.Meijer@microsoft.com

†Brian.Beckman@microsoft.com

The DOM implementation as surfaced in the .NET frameworks as the `System.Xml.XmlDocument` API is extremely imperative, irregular, and complex. Nodes are not first class citizens and have to be created and exist in the context of a given document. To construct this document using the DOM, we must first create a new document as a root for all other nodes we will create. Then we find that creating and inserting child nodes requires two imperative steps. One to create a new child element or attribute, and a second to then insert the just created node into the parent.

```
Dim PO As New XmlDocument
```

```
Dim purchaseOrder As XmlElement = _
  PO.CreateElement("purchaseOrder")
PO.AppendChild(purchaseOrder)
```

```
Dim orderDate XmlAttribute = _
  PO.CreateAttribute("orderDate")
orderDate.Value = "1999-10-20"
purchaseOrder.Attributes.Append(orderDate)
```

```
Dim shipTo As XmlElement = _
  PO.CreateElement("shipTo")
purchaseOrder.AppendChild(shipTo)
```

```
Dim country As XmlAttribute = _
  PO.CreateAttribute("country")
country.Value = "US"
shipTo.Attributes.Append(country)
...
```

This is unfortunate for readability, debuggability, and maintainability, because now there is no helpful relationship between the structure of the code we write and the structure of the XML we need to create.

The access patterns for attributes and elements are gra-

tuitously different, and the handling of namespaces is confusing at best. Finally even pretty-printing an XML document takes several lines of arcane and complex code since the `.ToString()` method is not properly overridden.

XML Programming Using Xlinq

To address the complexity of working with XML, we designed Xlinq, a new modern lightweight XML API that is designed from the ground up with simplicity and ease of programming in mind. Moreover Xlinq integrates smoothly with the language integrated queries of the LINQ framework.

In Xlinq nodes are truly first class citizens that can be passed around freely independent of an enclosing document context. Nested elements are constructed in an expression-oriented fashion (functional construction), but Xlinq also supports imperative updates in case programmers need them. Elements and attributes are accessed uniformly using familiar XPath axis-style methods, while namespace handling is simplified using the notion of universal names throughout the API. Last but not least, `.ToString()` actually works, so it is trivial to pretty print XML documents using a single method call.

The Xlinq object model contains a handful of types. The abstract class `XNode` is the base for element nodes, and provides `Parent` and methods such as `AddBeforeThis`, `AddAfterThis` and `Remove` for updates in the imperative style. For IO, it provides methods for reading `ReadFrom` and writing `WriteTo`.

The abstract class `XContainer` is the base for element nodes that have children. It adds axis methods such as `Content`, `Descendants`, and `Element` and `Elements`, as well as imperative update methods such as `Add`, `AddFirst`, `RemoveContent`, and `ReplaceContent`. The two main types, `XElement` and `XAttribute`, both inherit from `XContainer`, immediately below `XNode`.

The `XElement` class represents proper XML elements and adds further axes such as `Ancestors`, `SelfAndAncestors`, `SelfAndDescendants` and

`Attributes`, and explicit conversions for accessing the content of element nodes.

The `XAttribute` class represents attributes and is stand-alone; it does not derive from `XNode`. It provides the `Parent` axis, as well as imperative updates and explicit conversions to get the value of attributes.

The `XName` class represents fully expanded XML names and provides accessors for the various items of interest such as `ExpandedName`, `LocalName`, and `NamespaceName`, as well as conversions to and from strings to `XName`. Internally, the `XName` class maintains a generational name table that allows for efficient comparison of names.

The Xlinq axes, in combination with the fact that the Visual Basic compiler is happy to insert downcasts on behalf of the programmer, makes the computation of a function `ComputeTotal` that computes the total value of the purchase order a breeze.

```
Function ComputeTotal(PO As XElement) As Double
    For Each Dim Item In PO.Descendants("item")
        Dim Price As Double = _
            Item.Element("price")
        Dim Quantity As Integer = _
            Item.Element("quantity")
        Total += Quantity * Price
    Next
End Function
```

Adding a new child element that contains the computed total is also simple, since Xlinq supports imperative update operations as well as functional construction. In this case we attach the total price as an attribute to the added node:

```
PO.Add(New XElement("Total", _
    New XAttribute("Price", Total(PO))))
```

XML Programming Using VB

On top of the base Xlinq API, Visual Basic adds XML literals with full namespace support, and late bound axis member for attribute, child, and descendant ac-

cess. Programming against XML now actually is easy, as it was originally intended.

With XML literals, we can directly embed XML fragments in a Visual Basic program. Inside XML literals we can leave holes for attributes, attribute names, or attribute values, for element names by using (expression), or for child elements using the ASP.Net style syntax `<%= expression %>`, or `<% statement %>` for blocks. The Visual Basic compiler takes XML literals and translates them into constructor calls of to the underlying XLinQ API. As a result, XML produced by Visual Basic can be freely passed to any other component that accepts XLinQ values, and similarly, Visual Basic code can accept XLinQ XML produced by external components.

One thing that Visual Basic's XML literals make particularly easy is handling of namespaces. We support normal namespace declarations, default namespace declarations, and no namespace declarations, as well as qualified names for elements and attributes. The compiler generates the correct XLinQ calls to ensure that prefixes are preserved when the XML is serialized.

```
Dim BillTo = _
    <a:billTo
      xmlns:a="http://ecommerce.org/schema"
      country="US">
    <a:name>Robert Smith</a:name>
    <a:street>8 Oak Avenue</a:street>
    <a:city>Old Town</a:city>
    <a:state>PA</a:state>
    <a:zip>95819</a:zip>
  </a:billTo>
```

Whereas XML literals make constructing XML easy in Visual Basic, the concept of axis members makes accessing XML easy. The essence of the idea is to delay the binding of identifiers to actual XML attributes and elements until run time. When the compiler cannot find a binding for a variable, it emits code to call a helper function at run time. This tactic will be familiar to many under the rubric "late binding", and, indeed, it is a form of ordinary Visual Basic late binding. But it has the advantage that the names of element tags and attributes can be used directly in Visual Basic code

without quoting. As such, it relieves the programmer of the significant cognitive burden of switching between object space and XML-data space. The programmer can treat the spaces the same: as hierarchies accessed through ".".

For example, we can use the child axis member `BillTo.street` to get all street child elements from the `BillTo` element, or we can use the attribute axis member `BillTo.@country` to get the country attribute of the `BillTo` element. The third axis we support directly is the descendants axis, written literally as three dots in the source code, `PO...item` to get all `item` children of the `PO` document, no matter how deeply in the hierarchy they occur. Axis member access works on both singletons as well as on collections, since in general there can zero, one, or several elements that are returned, or the accessor is applied to a collection to start with. This makes axis members compositional in the sense that we can chain dotting through the results of previous axis as in `PO...item.@partNum` to obtain the collection of all `partNum` attributes in the `PO` document.

Acknowledgements

Many thanks to the members of the XLinQ design team Don Box, Arpan Desai, Anders Hejlsberg, Asad Jawahar, Andrew Kimball, Dave Remy, Michael Rys, Erik Saltwell, and Ion Vasilian, and the Tesla and Visual Basic design teams Avner Aharoni, David Schach, Peter Drayton, Ralf Lammel, Amanda Silver, and Paul Vick for all their hard work to make XLinQ and the Visual Basic 9 language a reality.