

# Microsoft Office Programming: A Guide for Experienced Developers

ROD STEPHENS

Apress™

Microsoft Office Programming: A Guide for Experienced Developers  
Copyright © 2003 by Rod Stephens

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-121-6

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Technical Reviewer: John Mueller

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Sofia Marchant

Copy Editor: Scott Carter

Production Editor: Janet Vail

Proofreader: Lori Bring

Compositor: Diana Van Winkle, Van Winkle Design Group

Indexer: Kevin Broccoli

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email [orders@springer-ny.com](mailto:orders@springer-ny.com), or visit <http://www.springer-ny.com>. Outside the United States: fax +49 6221 345229, email [orders@springer.de](mailto:orders@springer.de), or visit <http://www.springer.de>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email [info@apress.com](mailto:info@apress.com), or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section and at <http://www.vb-helper.com/office.htm>.

## CHAPTER 15

# Office 2003

MICROSOFT OFFICE 2003 is a relatively major upgrade, including quite a few important enhancements. Some of the most important changes include:

- Enhancements for programmatically using XML content, XSD schemas, and XSL transformations
- Smart tag enhancements
- Visual Studio .NET Tools for Office, which lets you integrate Visual Studio .NET code with Office
- InfoPath (formerly XDocs), a new XML-based tool for providing fill-in-the-blank business forms
- Smart documents that interact with the task pane

That's quite a lot to absorb all at once. The good news is that you don't need to use all of these features. Although Visual Basic .NET is replacing the traditional Visual Basic language, VBA is still alive and well in Office 2003. It's likely that Visual Basic .NET will move inside the Office applications at some point, but it hasn't happened yet. (In fact, it seems probable that Microsoft will move all of Visual Studio into Office as well so you'll be able to program with C# if you prefer.) You don't have to use these new features to make Office 2003 work more or less the same way Office XP does now.

Several of these enhancements deal directly or indirectly with XML. Earlier chapters in this book avoided XML because it is a fairly complex topic in its own right. This book assumes you understand the basics of programming Visual Basic or VBA, but it doesn't assume that you know XML, XSD, XSL, SOAP, Web Services, and all of the other related topics that go along with XML.

The following section gives a brief introduction to XML so you can understand these Office 2003 enhancements. To learn more, see a book about XML, such as *Visual Basic .NET and XML* by Rod Stephens and Brian Hochgurtel (Wiley, 2002). You can also find XML information online. For example, <http://www.w3schools.com/xml> has a beginner's XML tutorial.

The rest of the chapter describes the major Office 2003 enhancements.

## Introduction to XML

Several Office 2003 enhancements deal with XML. This section provides a quick introduction to XML and some of the related technologies so you can make some sense of these enhancements. To really get the most out of these new features, you will probably need to learn a lot more about XML, but at least this section can get you started. It should at least help you understand what the enhancements do so you can decide whether you care enough to learn more.

XML is a very simple data storage language. It uses a system of user-defined *tags* to delimit data. An *element* begins with an open tag consisting of a word surrounded by pointy brackets. An element ends with a corresponding close tag that has the same name as the open tag, except a slash comes before the name.




---

**NOTE** *The starting and ending tags must have exactly the same name and matching capitalization.*

---

Elements can be nested to any depth and may define quite complex data structures. To be properly formatted, an XML document must have a single root-level element that contains all other elements. For example, the following XML text defines a Book record with elements named Author, Title, and Publisher. The Author element contains two sub-elements LastName and FirstName.

```
<Book>
  <Author>
    <LastName>Stephens<LastName>
    <FirstName>Rod<FirstName>
  </Author>
  <Title>Office Smackdown</Title>
  <Publisher>Apress</Publisher>
</Book>
```

That's about all there is to XML, at least in concept. XML defines some special tags that are used by an XML *parser*. The most important of these is an XML declaration tag that goes at the very top of the XML file. This tag defines the XML version (1.0 is the only version currently available), may include an “encoding” field that identifies the file's character set, and may include a “standalone” field that indicates whether the file is self-contained or needs external definitions. The following code shows a complete XML document defining a series of Title elements.

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<Books>
  <Title>Visual Basic Graphics Programming</Title>
  <Title>Ready-to-Run Visual Basic Algorithms</Title>
  <Title>Visual Basic Code Library</Title>
</Books>
```

In the never-ending quest to save keystrokes while adding new features, developers have taken XML from this humble starting point and extended it to a level of bewildering complexity. For instance, tags can have *attributes* that define conditions on an element. The following statement defines a phone number with a “kind” attribute set to the value “work.”

```
<Phone kind="work">234-456-7890</Phone>
```

Sometimes an element can define its data without actually containing any text, often by including the data in an attribute. In that case, the empty element can use a shorthand version that includes a trailing slash inside the opening tag and has no closing tag. For example, the following two lines both define the same image file.

```
<Image src="http://www.vb-helper.com/vbhelper_213_33.gif"></Image>
<Image src="http://www.vb-helper.com/vbhelper_213_33.gif" />
```

The last topic mentioned here is that of namespaces. If lots of different businesses tried to define their data in XML files, many would come up with similar elements for common items. They would probably end up using the same names for elements such as Customer, Employee, InventoryItem, Bribe, Attorney, and so forth. Different companies would probably have different definitions of these objects, however, so a program trying to work with documents from more than one company would have trouble with data mismatches.

You can use namespaces to resolve these kinds of conflicts. You declare a namespace by adding an `xmlns` statement to the document’s root node. The following code identifies the `vbhelper` namespace with the URL `www.vb-helper.com/smackdown`.

```
<?xml version="1.0" encoding="utf-8" ?>
<Books xmlns:vbhelper="www.vb-helper.com/smackdown">
  <vbhelper:Title>Visual Basic Graphics Programming</vbhelper:Title>
  <vbhelper:Title>Ready-to-Run Visual Basic Algorithms</vbhelper:Title>
  <vbhelper:Title>Visual Basic Code Library</vbhelper:Title>
</Books>
```

Note that the URL doesn't need to exist; it just needs to be unique. As long as different developers don't use the same URL, their element names won't collide.




---

**TIP** You can also define a default namespace for every element in the document by omitting the namespace's name in the root element. For instance, the following code makes `www.vb-helper.com/smackdown` the default namespace for the document. Whatever program parses the XML code should add that namespace to every element in the document.

```
<?xml version="1.0" encoding="utf-8" ?>
<Books xmlns="www.vb-helper.com/smackdown">
  <Title>Visual Basic Graphics Programming</Title>
  <Title>Ready-to-Run Visual Basic Algorithms</Title>
  <Title>Visual Basic Code Library</Title>
</Books>
```

---

There's not a whole lot more to XML. The real complexity comes from auxiliary technologies added to XML. For example, DTD (Document Type Definition), XDR (XML Data Reduced), and XSD (XML Schema Definition) are all XML schema definition languages. Their files define the types of data that an XML file is allowed to contain. For example, an XSD file can declare that an XML file's Address element must contain Street, City, State, and Zip elements exactly once each and in that order.

By attaching a schema to an XML file, you can provide some data validation. For example, if a customer sends you an order in an XML file that has been validated against a schema, you know that it contains whatever fields it must: Name, Address, Phone, CreditCard, and so forth. Unfortunately, schema languages are relatively complex, so they're not covered in any detail here.

XSL (eXtensible Stylesheet Language) is another XML accessory technology. XSL is a data transformation language. You can use it to tell a *transformation engine* how it should convert XML data into some form of output. For example, you might use different XSL files to transform a single XML document into versions in plain text, HTML, or VoiceXML. You could even transform the XML file into a new XML file with a different structure or containing only parts of the whole dataset.

XSL consists of three parts: XSLT (XSL Transformation Language), FO (XSL Formatting Objects), and XPath (XML Path). XSLT is the language of XSL. It lets you specify how you want different elements transformed.

FO includes the objects that make up the output document. These are the element, attribute, and other entities in the document. Generally, you don't need to think about these objects or even really know they exist.

XPath is a language (yes, another one!) that specifies nodes inside an XML document. An XPath statement looks a bit like a directory specification. For example, the statement `/Books/Book/Author/LastName` means the `LastName` element inside an `Author` element inside a `Book` element inside the `Books` root element.

XPath uses two kinds of wild cards to skip intermediate values. The `*` symbol tells XPath to match any single intermediate element. For example, the statement `/Books/*/Author` matches any `Author` element that is a grandchild of the `Books` element. If the XML file contains `Book`, `Article`, and `WebPage` elements, this statement would find the `Author` elements of all three (assuming `Author` is the next level down in the hierarchy).

The `//` sequence tells XPath to skip any number of intermediate nodes. The statement `/Books//Author` matches any `Author` element that is a descendant of the `Books` data root node.

XSL is much more complex than this simple discussion indicates, but there's no room to go into any kind of depth here. For more information, see a book on XML or XSL. You can also find an introductory tutorial at <http://www.w3schools.com/xsl>.

## Word Tools for Manipulating XML

In Office XP, Excel 2002 can load and save properly formatted XML files. When you save a workbook in an XML file, Excel includes document properties, worksheet styles, and worksheet options in addition to the data itself. Excel 2002 can load a file saved in this format but it cannot load an arbitrary XML file.

Word 2003 and Excel 2003 can both load arbitrary XML files intelligently. The following code shows a very simple XML file.

```
<Books>
  <Book>Visual Basic Graphics Programming</Book>
  <Book>Ready-to-Run Visual Basic Algorithms</Book>
  <Book>Visual Basic Code Library</Book>
</Books>
```

Figure 15-1 shows Word 2003 displaying this file. The file's start and end tags are outlined so they are easy to find. The pointer is hovering over a `</Book>` end tag, so Word is displaying a popup that says End of Book. The first `Book` element is selected in the XML Structure panel on the right, so that element is highlighted in the main document.

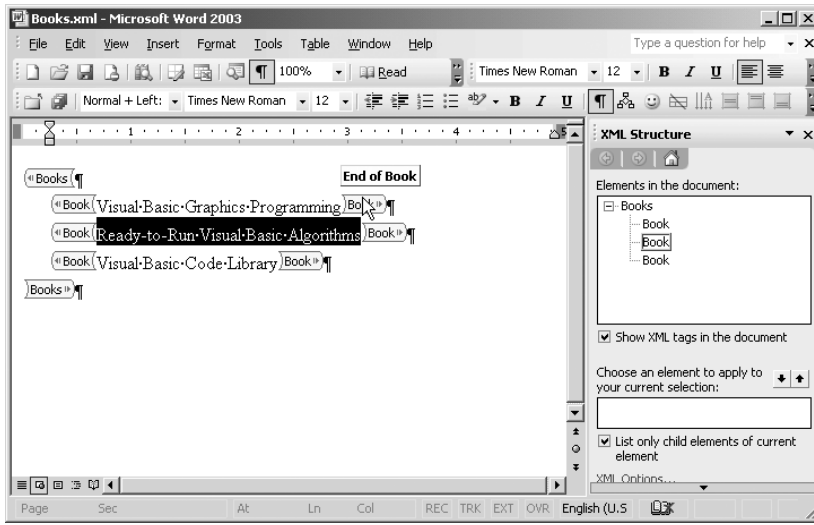


Figure 15-1. Word 2003 can load arbitrary XML files.



**NOTE** To get the full benefit of Word's XML tools, you must attach a schema to the XML document.

To transform an XML file, load the file, select File ► Save As, check the “Apply transform” box, click the Transform button, and select the XSL file to apply.

A Word document can also hold XML data inside a normal text document. Simply invoke the Insert menu's File command and select the file.

All this is new but it's not VBA programming. The following sections describe new objects and methods that let you examine and manipulate a document's XML content using VBA code.

## The XMLNode Object

The new XMLNode object represents a node in an XML hierarchy. You can use these objects to navigate over and manipulate the XML data's structure. Table 15-1 lists some of the XMLNode's most important properties and methods.



Table 15-1. Useful XmlNode Properties and Methods

PROPERTIES/METHODS	PURPOSE
Attributes	A collection holding XmlNode objects representing the node's attributes. The attribute node's BaseName property gives the attribute's name, and its NodeValue property gives its value.
BaseName	The name of the node without the namespace.
ChildNodes	A collection holding XmlNode objects representing the node's child nodes.
Delete	Removes the node from its XML hierarchy. Contrary to what you might expect, this leaves the node's contents behind. If the node contained other nodes, those nodes become children of the parent node. If the node contains text, the text becomes part of the parent's text. See also the RemoveChild method.
FirstChild, LastChild	The first and last child nodes of this node. You can use these together with NextSibling and PreviousSibling to loop through the node's children.
HasChildNodes	True if the node has child nodes.
NextSibling, PreviousSibling	Return the node's next and previous sibling (brother or sister) in the parent node's child collection.
NodeType	Returns wdXmlNodeAttribute or wdXmlNodeElement to indicate whether the node is an element or another node's attribute value.
NodeValue	A string containing an attribute node's value.
ParentNode	The XmlNode that is this node's parent.
RemoveChild	Removes a specified child from the node. The specified child node must be a child of this node. This method removes the child's contents, including any text and child nodes it contains. See also the Delete method.
SelectNodes	Returns a collection of XmlNode objects satisfying an XPath query. See the following section for more information.
SelectSingleNode	Returns a single XmlNode object satisfying an XPath query. See the following section for more information.
Text	The node's text. For example, in the XML code <Flavor>Chocolate</Flavor> the Flavor node has Text value Chocolate. Note that this text includes the text of all nodes contained in this one.
Validate	Validates the node against any attached XML schemas.
XML	Returns the node's XML code. This method's parameter indicates whether the method should include the Word XML markup.

The following sections describe some of the ways you can use `XmlNode` objects to examine and manipulate XML structure.

### *Using `SelectNodes` and `SelectSingleNode`*

The `XmlNode` object's `SelectNodes` and `SelectSingleNode` methods search the XML hierarchy below the node and return any nodes that match a given XPath expression. The section “Introduction to XML” earlier in this chapter briefly described XPath and mentioned that it allows the wildcards `*` and `//`. To see how these work, suppose a document contains the following XML data.

```
<Desserts>
  <IceCream>
    <Flavor>Chocolate</Flavor>
    <Flavor>Vanilla</Flavor>
    <Flavor>Strawberry</Flavor>
  </IceCream>
  <Cookie>
    <Flavor>Chocolate Chip</Flavor>
    <Flavor>Pecan Sandy</Flavor>
  </Cookie>
</Desserts>
```

The following statement uses the first node's (the `Desserts` node's) `SelectSingleNode` method to display the text contained in the first `Flavor` node. It uses the `//` wildcard so it would search as deeply as necessary to find a `Flavor` node.

```
Debug.Print ActiveDocument.XMLNodes(1).SelectSingleNode("//Flavor").Text
```

The following code displays all of the `Flavor` node text values whether they are contained in the `IceCream` node or the `Cookie` node. The XPath statement uses the `*` wildcard to skip one level in the hierarchy (both the `IceCream` and `Cookie` nodes) between the `Desserts` node and its `Flavor` grandchildren. Note that the XPath statement begins with the name of the current node, `Desserts`.

```
For Each node In ActiveDocument.XMLNodes(1).SelectNodes("/Desserts/*/Flavor")
  Debug.Print node.Text
Next node
```

After these methods return one or more `XMLNodes`, you can use the nodes' properties and methods to manipulate them.

## Examining the XML Hierarchy

You can use XMLNode objects to navigate over the XML hierarchy to examine its data.

For example, the following code displays the XML structure of a Word document. Subroutine ShowDocumentXMLHierarchy searches the active document's XMLNodes collection looking for nodes with no parents. Those are the root nodes for the document's various blocks of XML data embedded in the Word document. For each of those root nodes, the code calls subroutine ShowNodeXMLHierarchy.

The ShowNodeXMLHierarchy subroutine builds a string to describe the node it received as a parameter. It begins with the node's name. If the node has attributes, the code adds them to the string. It then displays the string in the Debug window.

Next subroutine ShowNodeXMLHierarchy loops through the node's ChildNodes collection calling itself recursively to display information on each child node.



**FILE** Ch15\Books.doc

---

```
' Display the document's XML hierarchy.
Sub ShowDocumentXMLHierarchy()
Dim node As XMLNode

    ' Search the nodes for those with no parents.
    For Each node In ActiveDocument.XMLNodes
        If node.ParentNode Is Nothing Then
            ShowNodeXMLHierarchy node
        End If
    Next node
End Sub

' Display this node's XML hierarchy.
Sub ShowNodeXMLHierarchy(ByVal node As XMLNode, _
    Optional ByVal indent As Integer = 0)
Dim txt As String
Dim attr_node As XMLNode
Dim i As Integer

    ' Display this node.
    txt = Space$(indent) & node.BaseName
    If node.Attributes.Count > 0 Then
```

```

        txt = txt & " ("
        For Each attr_node In node.Attributes
            txt = txt & attr_node.BaseName & "=" & attr_node.NodeValue & ", "
        Next attr_node
        txt = Left$(txt, Len(txt) - 2) & ")"
    End If
    Debug.Print txt

    ' Display the hierarchy below this node.
    ' (A bug seems to make For Each not work.)
    For i = 1 To node.ChildNodes.Count
        ShowNodeXMLHierarchy node.ChildNodes(i), indent + 4
    Next i
End Sub

```

The following text shows the output produced for the file Ch14\Books.doc. This file contains two chunks of XML data with root nodes named Books and Desserts.

```

Books
  Book (Price=$49.99, Pages=395)
    Title
  Book (Price=$49.99, Pages=684)
    Title
  Book (Price=$49.99, Pages=712)
    Title
Desserts
  IceCream
    Flavor
    Flavor
    Flavor
  Cookie
    Type
    Type

```

### *Building XML Data*

You can also use XMLNode objects to build XML data. You can use the Add method provided by the XMLNodes collection in the Document, Range, and Selection objects to create new nodes. Then you can use the Add method provided by the ChildNodes collection in those nodes to add child nodes.

An example described shortly shows how to do this. Before it can work, however, the document must have the proper XML namespace installed. The schema file *Books.xsd* (shown in the following code) defines the namespace for the document. This file sets its target and default namespaces to `http://www.vb-helper.com/bks`. It then defines a *Books* element. That element is a sequence containing a series of *Book* elements. Each *Book* element is a sequence containing *Title*, *URL*, *Image*, *Price*, and *Page* elements. Each of those elements is a string that can occur zero or one times.




---

**FILE** *Ch15\Books.xsd*


---

```
<?xml version="1.0"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.vb-helper.com/bks"
  xmlns="http://www.vb-helper.com/bks"
  elementFormDefault="qualified"
>
  <xsd:element name="Books">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Book">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Title" type="xsd:string"
                minOccurs="0"
                maxOccurs="1"/>
              <xsd:element name="URL" type="xsd:string"
                minOccurs="0"
                maxOccurs="1"/>
              <xsd:element name="Image" type="xsd:string"
                minOccurs="0"
                maxOccurs="1"/>
              <xsd:element name="Price" type="xsd:string"
                minOccurs="0"
                maxOccurs="1"/>
              <xsd:element name="Pages" type="xsd:string"
                minOccurs="0"
                maxOccurs="1"/>
            
```

```

        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>

```

The following code builds an XML hierarchy from scratch. It begins by trying to get the necessary namespace object from the Application object's XMLNamespaces collection. If the namespace doesn't exist, the program adds it. After it has a reference to the XMLNamespace object, the code attaches it to the Word document.

Now the code makes a Range object and collapses it so it indicates the end of the document. It calls the Range's XMLNodes.Add method to create a new Books node. The Namespace parameter specifies the value `http://www.vb-helper.com/bks` defined in the schema file. Note that the schema defines the Books element.

Next the code calls subroutine MakeBookNode three times to define more nodes in small XML subtrees. Subroutine MakeBookNode uses the Books node's ChildNodes.Add method to create a new Book node. The subroutine then adds Title and URL elements to the Book element. As it does so, the code sets the text values of these new elements. Notice that the schema defines all of these elements.




---

**FILE** *Ch15\BuildXml.doc*

---

```

' Build some XML data from scratch.
Sub BuildXML()
    Dim ns As XMLNamespace
    Dim rng As Range
    Dim books_node As XMLNode

    ' Get the XMLNamespace.
    On Error Resume Next
    Set ns = Application.XMLNamespaces( _
        "http://www.vb-helper.com/bks")
    On Error GoTo 0

    ' Add the namespace if necessary.
    If ns Is Nothing Then
        Set ns = Application.XMLNamespaces.Add( _
            "Books.xsd", _
            "http://www.vb-helper.com/bks")
    End If

```

```

' Attach the schema to the document.
ns.AttachToDocument ActiveDocument

' Make a Range at the end of the document.
Set rng = ActiveDocument.Range()
rng.Collapse wdCollapseEnd

' Create a Books node.
Set books_node = rng.XMLNodes.Add( _
    Name:="Books", Namespace:=ns.URI)

' Make some Book nodes.
MakeBookNode books_node, _
    "VB .NET and XML", _
    "http://www.vb-helper.com/xml.htm"
MakeBookNode books_node, _
    "Visual Basic Graphics Programming", _
    "http://www.vb-helper.com/vbgp.htm"
MakeBookNode books_node, _
    "Ready-to-Run Visual Basic Algorithms", _
    "http://www.vb-helper.com/vba.htm"
End Sub

' Add a Book node as a child of this node.
Sub MakeBookNode(ByVal parent_node As XMLNode, ByVal book_title As String, _
    ByVal book_url As String)
Dim book_node As XMLNode

' Make the Book node.
Set book_node = parent_node.ChildNodes.Add( _
    Name:="Book", Namespace:="http://www.vb-helper.com/bks")

' Add the Title node child.
book_node.ChildNodes.Add( _
    Name:="Title", Namespace:="http://www.vb-helper.com/bks").Text = _
    book_title

' Add the URL node child.
book_node.ChildNodes.Add( _
    Name:="URL", Namespace:="http://www.vb-helper.com/bks").Text = book_url
End Sub

```




---

**CAUTION** While I was working with this code, I had a lot of trouble getting Word to refresh the schema file. It kept using old versions of the file, so the code didn't work. The solution I found was to interactively use the Schema Library (Tools ► Templates and Add-Ins ► Schema Library button) to delete the schema, detach the schema on the Templates and Add-Ins dialog's XML Schema tab, close the dialog, reopen it, and read the schema. You should be able to reload the schema as in `ActiveDocument.XMLSchemaReferences(1).Reload`, but I had trouble getting it to work properly in the beta.

---

Although this code works, it's somewhat cumbersome. If you just want to create a block of XML data, you can use the Range object's `InsertXML` method to dump the data's XML definition right into the document.

The following code builds a string representing some XML data and inserts it into the active Word document.




---

**FILE** *Ch15\InsertXml.doc*

---

```
' Build some XML data from scratch.
Sub MakeXML()
Dim rng As Range

    Set rng = ActiveDocument.Range()
    rng.Collapse wdCollapseEnd

    Dim txt As String
    txt = "<?xml version='1.0'?">"
    txt = txt & "<Desserts>"
    txt = txt & "    <IceCream>"
    txt = txt & "        <Flavor>Chocolate</Flavor>"
    txt = txt & "        <Flavor>Vanilla</Flavor>"
    txt = txt & "        <Flavor>Strawberry</Flavor>"
    txt = txt & "    </IceCream>"
    txt = txt & "    <Cookie>"
    txt = txt & "        <Type>Chocolate Chip</Type>"
    txt = txt & "        <Type>Pecan Sandy</Type>"
    txt = txt & "    </Cookie>"
    txt = txt & "</Desserts>"
    rng.InsertXML txt
End Sub
```



This method is more convenient when you want to create a whole block of data, but the `BuildXML` subroutine described earlier demonstrates techniques you could use to add nodes in the middle of an existing piece of XML code.

## The Document Object

The `Document` object provides several new tools for working with the document's XML content. Its `XMLNodes` collection contains `XMLNode` objects that describe the content. The following section describes the `XMLNode` object. See the previous sections for examples that manipulate `XMLNode` objects.

The `Document` object's `XMLSaveDataOnly` property indicates whether Word saves the document's XML data only or whether it saves the full Word XML markup. Saving the full markup takes a bit more space in the document file.

The `XMLUseXSLTWhenSaving` property indicates whether Word should transform the document using XSL when you save it as XML. This only applies when you save the file as XML, not when you save it as a normal Word document.

If `XMLUseXSLTWhenSaving` is `True`, then Word applies the XSL template indicated by the `Document` object's `XMLSaveThroughXSLT` property.

When you select the XML document type in the `Save As` dialog, the dialog displays a "Save data only" check box. Check this to save only the data. Leave the box unchecked to save the entire document in Word XML format.



**NOTE** *Saving through XSLT may produce some strange results if the document is not an XML file.*

---

## The XML Method

The `Range`, `Selection`, and `SmartTag` objects now have an `XML` method that returns an object's XML contents. The method's only parameter is a Boolean that indicates whether the routine should include the Word XML markup. Set this parameter to `False` to get just the XML data.

Note that XML data ignores carriage returns and other white space between tags, so many programs strip these off. The XML data is valid but may be hard to read. The following XML code shows the results of the `XML` method for one Word document. The output is broken here to fit on the page, but the actual result is all on two lines (the XML declaration gets its own line).

```
<?xml version="1.0" standalone="no"?>
<Books><Book Price="$49.99" Pages="395"><Title>Ready-to-Run Visual Basic Algorithms</Title></Book><Book Price="$49.99" Pages="684"><Title>Custom Controls Library</Title></Book><Book Price="$49.99" Pages="712"><Title>Visual Basic Graphics Programming</Title></Book></Books>
```

## Excel Tools for Manipulating XML

Excel uses XML mappings to transform XML data into worksheet cells. Typically, you use an XSD schema to define the mapping. Then, when you load an XML file that uses the namespace defined by the schema, Excel uses that mapping to display the data.

For example, consider the following schema. Note that the elements in this schema use the namespace `http://www.vb-helper.com/bks`. Later, when VBA code wants to load XML data using this schema's mapping, it must define data using the same namespace.



**FILE** *Ch15\Books2.xsd*

---

```
<?xml version="1.0"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.vb-helper.com/bks"
  xmlns="http://www.vb-helper.com/bks"
  elementFormDefault="qualified"
>
  <xsd:element name="Books">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Book">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Title" type="xsd:string"
                minOccurs="0"
                maxOccurs="1"/>
              <xsd:element name="URL" type="xsd:string"
                minOccurs="0"
```

```

        maxOccurs="1"/>
<xsd:element name="Image" type="xsd:string"
    minOccurs="0"
    maxOccurs="1"/>
<xsd:element name="Price" type="xsd:string"
    minOccurs="0"
    maxOccurs="1"/>
<xsd:element name="Pages" type="xsd:string"
    minOccurs="0"
    maxOccurs="1"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>

```

The following VBA code makes an `XmlMap` object to represent a mapping for this schema. The code begins by using the `ActiveWorkbook` object's `XmlMaps` collection to create a new `XmlMap` object and changes its name to `Books_Map`. The code saves the object's `RootElementNamespace.Prefix` property. This is a namespace abbreviation that Excel created for the namespace.

The routine then uses the `ActiveWorksheet` object's `ListObjects` collection to make a new `ListObject` to represent a new List. A List is a new feature in Excel 2003 that defines the fields in the schema that should be mapped onto the worksheet.

When the `ListObjects` collection creates a new `ListObject`, it initially places one item in the `ListObject`'s `ListColumns` collection. The `ListColumns` entries define the columns that should be selected from the schema. After it creates the `ListObject`, the code changes the name of this initial `ListColumn` object to `Title`. It then calls the object's `SetValue` method to tell the `ListColumn` what element it represents in the schema. The first parameter to this method is the `XmlMap` it should use.

The `SetValue` method's second parameter is an XPath expression describing the data that this `ListColumn` should represent. The program defines the expression using `@` symbols as namespace placeholders and then replaces the symbols with the `XmlMap` object's namespace prefix. After this first call to `SetValue`, the first `ListColumn` object represents the `/Books/Book/Title` element in the schema.

Now the code makes a new `ListColumn` object, sets its name to `URL`, and assigns it to the `/Books/Book/URL` schema element.

Finally, the code makes one more `ListColumn` object, sets its name to `Price`, and assigns it to the `/Books/Book/Price` schema element.




---

**FILE** Ch15\Books2.xls
 

---

```
' Map an XSD schema.
Sub MapSchema()
Dim books_map As XmlMap
Dim books_list As ListObject
Dim books_column As ListColumn
Dim pfx As String
Dim xpath_spec As String

' Load the schema.
Set books_map = ActiveWorkbook.XmlMaps.Add("Books2.xsd")
books_map.Name = "Books_Map"

' Get the XmlMap's namespace prefix.
pfx = books_map.RootElementNamespace.Prefix

' Create a new list.
Set books_list = ActiveSheet.ListObjects.Add
books_list.Name = "Books_List"

' Map the Title element in the initially
' created ListColumn.
books_list.ListColumns(1).Name = "Title"
xpath_spec = Replace("/@:Books/@:Book/@:Title", "@", pfx)
books_list.ListColumns(1).XPath.SetValue _
    books_map, xpath_spec

' Map the URL element in a new ListColumn.
Set books_column = books_list.ListColumns.Add
books_column.Name = "URL"
xpath_spec = Replace("/@:Books/@:Book/@:URL", "@", pfx)
books_column.XPath.SetValue _
    books_map, xpath_spec

' Map the Price element in a new ListColumn.
Set books_column = books_list.ListColumns.Add
books_column.Name = "Price"
xpath_spec = Replace("/@:Books/@:Book/@:Price", "@", pfx)
books_column.XPath.SetValue _
    books_map, xpath_spec
End Sub
```

After it has defined an `XmlMap` object for a schema, the code can use that object to load an XML file. The following code shows the XML data for this example. Notice that the `Books` element declares itself as using the `http://www.vb-helper.com/bks` namespace. It's the fact that this namespace matches the one used by the elements defined in the schema that tells Excel to use the schema's mapping when loading this data.




---

**FILE** *Ch15\Books2.xml*


---

```
<?xml version="1.0"?>
<Books xmlns="http://www.vb-helper.com/bks">
  <Book>
    <Title>Visual Basic .NET and XML</Title>
    <URL>http://www.vb-helper.com/xml.htm</URL>
    <Image>http://www.vb-helper.com/xml.jpg</Image>
    <Price>$39.99</Price>
  </Book>
  <Book>
    <Title>Ready-to-Run Visual Basic Algorithms</Title>
    <URL>http://www.vb-helper.com/vba.htm</URL>
    <Image>http://www.vb-helper.com/vba.jpg</Image>
    <Price>$49.99</Price>
  </Book>
  <Book>
    <Title>Custom Controls Library</Title>
    <URL>http://www.vb-helper.com/ccl.htm</URL>
    <Image>http://www.vb-helper.com/ccl.jpg</Image>
    <Price>$49.99</Price>
  </Book>
  <Book>
    <Title>Visual Basic Graphics Programming</Title>
    <URL>http://www.vb-helper.com/vbgp.htm</URL>
    <Image>http://www.vb-helper.com/vbgp.jpg</Image>
    <Price>$49.99</Price>
  </Book>
</Books>
```

The following code loads the XML file. It retrieves a reference to the Books\_Map XmlMap object and calls that object's Import method passing it the name of the XML file to load. Excel loads the data by using the defined mapping and returns a success or failure code.




---

**FILE** Ch15\Books2.xls

---

```
' Load XML data using the Books_Map map.
Sub LoadXmlData()
Dim books_map As XmlMap

    ' Get the XmlMap.
    Set books_map = ActiveWorkbook.XmlMaps("Books_Map")

    ' Import the data.
    Select Case books_map.Import("Books2.xml")
        Case xlXmlImportSuccess
            MsgBox "OK"
        Case xlXmlImportElementsTruncated
            MsgBox "Some data was truncated"
        Case xlXmlImportValidationFailed
            MsgBox "The XML file doesn't match the schema"
    End Select
End Sub
```




---

**NOTE** *In my tests, the Import method always returned xlXmlImportValidationFailed even though the XML file looked like it matched the schema. Loading the file interactively using this mapping worked perfectly. This may be a bug in the beta, and it may be fixed before Office 2003's final release.*

---

Figure 15-2 shows the results. The cells A1:C5 contain the imported data. The XML Source pane on the right shows the XML mapping. The Title, URL, and Price elements are bold because ListColumn objects define mappings for them.

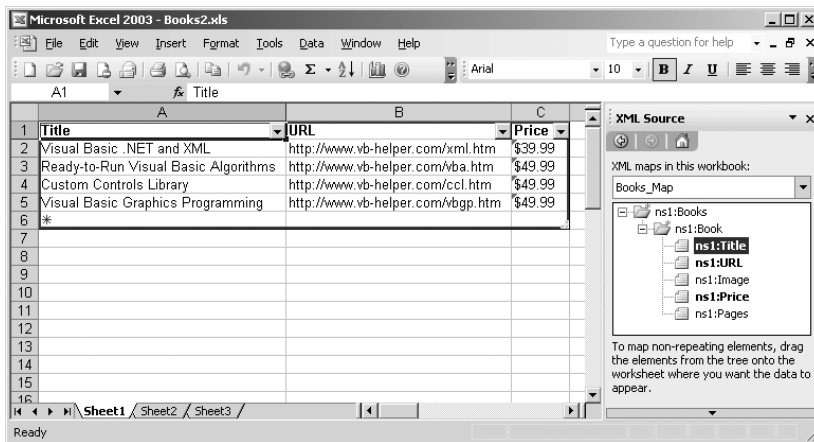


Figure 15-2. Excel 2003 can display XML data loaded by an XmlMap object.

## Excel List Tools

The previous section explains how to use a List in Excel to display XML data. You can also use Lists to display non-XML data.

The following code makes an empty List and then fills it with data. It begins by making the variable `ws` refer to the active worksheet. The code could use `ActiveSheet` instead of a variable; however, VBA doesn't provide IntelliSense for the `ActiveSheet` object, but it does for a variable of type `Worksheet`.

Next the program uses the `Worksheet` object's `ListObjects.Add` method to make a new List. It specifies that the new List should come from the range consisting of the single cell A1.



**NOTE** Two Lists cannot overlap. If you try to create a List with a range already being used by another List, VBA raises an error.

The code then uses a `With` statement to work with the new `ListObject`. It could work directly with the result of the `ListObjects.Add` method as in `With ws.ListObjects.Add(...)`, but again VBA wouldn't provide IntelliSense.

Next the program sets the List's `Name` property. Note that the name must be unique. If you try to give a List the same name as an existing List, Excel raises an error.

The subroutine uses the `ListObject`'s `Range` method to add values to the List. It finishes by specifying some formatting for the List, fitting the List's columns to their data, making the header row bold, and making the body rows nonbold.




---

**FILE** *Ch15\Lists.xls*


---

```

' Make a List from scratch.
Sub MakeListFromScratch()
Dim ws As Worksheet
Dim books_list As ListObject

    ' Make the list starting at cell A1.
    Set ws = ActiveSheet
    Set books_list = ws.ListObjects.Add( _
        SourceType:=xlSrcRange, _
        Source:=ws.Range("A1"), _
        XlListObjectHasHeaders:=xlYes)

    ' Work with the list.
    With books_list
        ' Name the list.
        .Name = "Book_List"

        ' Make some data.
        .Range(1, 1).Value = "Title"
        .Range(1, 2).Value = "URL"
        .Range(1, 3).Value = "Price"

        .Range(2, 1).Value = "Visual Basic .NET and XML"
        .Range(2, 2).Value = "http://www.vb-helper.com/xml.htm"
        .Range(2, 3).Value = "$39.99"

        .Range(3, 1).Value = "Ready-to-Run Visual Basic Algorithms"
        .Range(3, 2).Value = "http://www.vb-helper.com/vba.htm"
        .Range(3, 3).Value = "$49.99"

        .Range(4, 1).Value = "Visual Basic Graphics Programming"
        .Range(4, 2).Value = "http://www.vb-helper.com/vbgp.htm"
        .Range(4, 3).Value = "$49.99"

        ' Format the results.
        .HeaderRowRange.Font.Bold = True      ' Bold headers.
        .DataBodyRange.Font.Bold = False      ' Non-bold data.
        .Range().Columns.AutoFit              ' Fit to the data.
    End With
End Sub

```



You can also make a List by using data in an existing range rather than creating the List first and then adding data to it. The following code creates a Range variable and uses its `FormulaArray` property to fill it with data. It then creates a new List object much as the previous example did, but this time it builds the new List on the data-filled range. That automatically fills the List with the range's data. The code finishes by setting the List's name and performing some formatting as before.




---

**FILE** *Ch15\Lists.xls*


---

```
' Make a List from an existing Range.
Sub MakeListFromRange()
Dim ws As Worksheet
Dim rng As Range
Dim books_list As ListObject

    ' Place data in the Range A7:C10.
    Set ws = ActiveSheet
    Set rng = ws.Range("A7:C10")
    rng.FormulaArray = Array( _
        Array("Title", "URL", "Price"), _
        Array("Visual Basic .NET and XML", _
            "http://www.vb-helper.com/xml.htm", "$39.99"), _
        Array("Ready-to-Run Visual Basic Algorithms", _
            "http://www.vb-helper.com/vba.htm", "$49.99"), _
        Array("Visual Basic Graphics Programming", _
            "http://www.vb-helper.com/vbgp.htm", "$49.99") _
    )

    ' Make the list on the pre-populated range E1:G4.
    Set books_list = ws.ListObjects.Add( _
        SourceType:=xlSrcRange, _
        Source:=rng, _
        XlListObjectHasHeaders:=xlYes)

    ' Format the list.
    With books_list
        .Name = "Book_List2"
        .HeaderRowRange.Font.Bold = True      ' Bold headers.
        .DataBodyRange.Font.Bold = False      ' Non-bold data.
        .Range().Columns.AutoFit              ' Fit to the data.
    End With
End Sub
```

The `ListObject` class provides a couple methods for working with a List. The `Delete` method removes the List from the workbook along with the data it contains. The `Publish` method publishes the List to a SharePoint server so other users can access it.

The `Unlist` method converts the List into a normal Excel range. The range contains the List's data but doesn't provide the List's special features. The following code converts the List named `Book_List2` into a normal range.



**FILE** *Ch15\Lists.xls*

---

```
' Convert the list into a regular range.
Sub ListToRange()
Dim ws As Worksheet
Dim books_list As ListObject

    Set ws = ActiveSheet
    Set books_list = ws.ListObjects("Book_List2")
    books_list.Unlist
End Sub
```

The `ListObject`'s `ShowTotals` property determines whether the List automatically adds a total row at the bottom. The following code creates a List using a three-column data-filled range. It adds a fourth column to the List, calculating its values by multiplying the second and third columns. It then formats the List, making the second and fourth columns display as monetary values and the third display as a whole number.



**FILE** *Ch15\Lists.xls*

---

```
' Make an expense List from an existing Range.
Sub MakeExpenseList()
Dim ws As Worksheet
Dim rng As Range
Dim invoice_list As ListObject
Dim i As Integer
```

```

' Place data in the Range A7:C10.
Set ws = ActiveSheet
Set rng = ws.Range("E1:G6")
rng.FormulaArray = Array( _
    Array("Item", "Unit Cost", "Quantity"), _
    Array("Floppy Discs, 50", "$25.95", "2"), _
    Array("CD RW discs, 50", "$43.50", "1"), _
    Array("Mouse Pad", "$4.95", "1"), _
    Array("Laser Printer Labels", "$1.95", "6"), _
    Array("Notebook Computer", "$1,234.00", "1") _
)

' Make the list on the pre-populated range E1:G4.
Set invoice_list = ws.ListObjects.Add( _
    SourceType:=xlSrcRange, _
    Source:=rng, _
    XlListObjectHasHeaders:=xlYes)

' Format the list.
With invoice_list
    ' Add the Total Price column.
    .HeaderRowRange(1, 4).Value = "Total"
    For i = 1 To .DataBodyRange.Rows.Count
        .DataBodyRange(i, 4).Value = _
            .DataBodyRange(i, 2).Value * _
            .DataBodyRange(i, 3).Value
    Next i

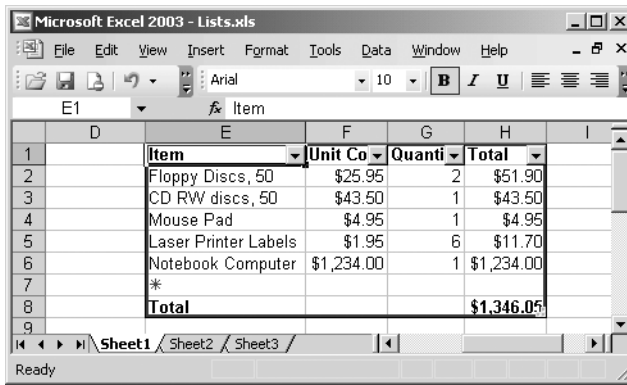
    .Name = "Invoice_List"
    .HeaderRowRange.Font.Bold = True      ' Bold headers.
    .DataBodyRange.Font.Bold = False      ' Non-bold data.

    .DataBodyRange.Columns(2).NumberFormat = "$#,##0.00"
    .DataBodyRange.Columns(3).NumberFormat = "0"
    .DataBodyRange.Columns(4).NumberFormat = "$#,##0.00"

    .ShowTotals = True                    ' Display totals.
    .Range().Columns.AutoFit              ' Fit to the data.
End With
End Sub

```

Figure 15-3 shows the resulting List. In this Figure the List is selected so it is displaying its “insert row” (with the asterisk in the leftmost column). Below that is the total row.



	D	E	F	G	H	I
		Item	Unit Co	Quanti	Total	
1						
2		Floppy Discs, 50	\$25.95	2	\$51.90	
3		CD RW discs, 50	\$43.50	1	\$43.50	
4		Mouse Pad	\$4.95	1	\$4.95	
5		Laser Printer Labels	\$1.95	6	\$11.70	
6		Notebook Computer	\$1,234.00	1	\$1,234.00	
7		*				
8		Total			\$1,346.05	
9						

Figure 15-3. This List has ShowTotals set to True.

For information about other ListObject properties and methods, consult the online help.

## Smart Tag Enhancements

Chapter 14 explained how to build a smart tag DLL by using Visual Basic .NET. All those techniques still work with Office 2003, but Office 2003 has also introduced a few new tools for building smart tag DLLs.

The ISmartTagRecognizer2 and ISmartTagAction2 interfaces define new methods you can define for the smart tag classes. Leave the ISmartTagRecognizer and ISmartTagAction interfaces you have already defined in the classes and just add the new ones.



**NOTE** If you haven't already read Chapter 14, you may want to at least skim its material on smart tags. The example described here assumes it already implements the ISmartTagRecognizer and ISmartTagAction interfaces. If you want to follow along and add the new code yourself, but you don't want to implement these interfaces, you can start with the BookSmartTag example from Chapter 14.

The `ISmartTagRecognizer2` interface defines four new methods. The `SmartTagInitialize` method is called before any other recognizer method. It's `ApplicationName` parameter lets you learn earlier which host application is using the smart tag.

The `PropertyPage` method should return `True` if you are supplying a property page for the smart tag. If `PropertyPage` returns `True`, the `DisplayPropertyPage` method should display the property page.

The `Recognize2` method is where the class tries to recognize text. One new feature here is the `TokenList` parameter. The `TokenList` contains a list of the text's words broken into tokens at spaces, carriage returns, punctuation, and other characters that normally separate words. Instead of examining the entire text sent by the host application, the code can look at the words in this list if that is easier for it.

The following code fragment shows the `FlavorSmartTag` recognizer class's `Recognize2` method. This example simply looks for the string "flavor."



**FILE** *Ch15\FlavorsSmartTag\SmartTagRecognizer.vb*

---

```
Public Sub Recognize2(ByVal examine_text As String, _
    ByVal DataType As SmartTagLib.IF_TYPE, ByVal LocaleID As Integer, _
    ByVal RecognizerSite2 As SmartTagLib.ISmartTagRecognizerSite2, _
    ByVal ApplicationName As String, _
    ByVal TokenList As SmartTagLib.ISmartTagTokenList) _
    Implements SmartTagLib.ISmartTagRecognizer2.Recognize2
    Dim i As Integer
    For i = 1 To TokenList.Count
        If TokenList.Item(i).Text.ToLower = "flavor" Then
            RecognizerSite2.CommitSmartTag2( _
                "http://www.vb-helper.com#FlavorsSmartTag", _
                TokenList.Item(i).Start, _
                TokenList.Item(i).Length, _
                RecognizerSite2.GetNewPropertyBag())
        End If
    Next i
End Sub
```

The `ISmartTagAction2` interface defines five new methods. The `SmartTagInitialize` method is called before any other action method. Its `ApplicationName` parameter lets you learn earlier which host application is using the smart tag.

The `VerbCaptionFromID2` method returns the smart tag menu's caption for a verb given its `VerbID`. This would be the same as the existing `VerbCaptionFromID` method except this method allows you to build cascading menus. Simply insert three slashes (`///`) in the caption where you want similar captions to match. For example, the following three captions define a menu labeled "Replace with..." that cascades to three submenu items labeled "Chocolate," "Vanilla," and "Strawberry."

```
Replace with...///Chocolate
Replace with...///Vanilla
Replace with...///Strawberry
```

The following code shows the `FlavorSmartTag` example's `VerbCaptionFromID2` method. It provides captions for two top-level menu items labeled "Replace with..." and "Go To Web Site." The first menu has three submenus labeled "Chocolate," "Vanilla," and "Strawberry." Figure 15-4 shows the result.




---

**FILE** *Ch15\FlavorsSmartTag\SmartTagAction.vb*

---

```
Public ReadOnly Property VerbCaptionFromID2(ByVal VerbID As Integer, _
    ByVal ApplicationName As String, ByVal LocaleID As Integer, _
    ByVal Properties As SmartTagLib.ISmartTagProperties, _
    ByVal recognized_text As String, ByVal Xml As String, _
    ByVal Target As Object) As String _
    Implements SmartTagLib.ISmartTagAction2.VerbCaptionFromID2
    Get
        Select Case VerbID
            Case 1
                Return "Replace with...///Chocolate"
            Case 2
                Return "Replace with...///Vanilla"
            Case 3
                Return "Replace with...///Strawberry"
            Case 4
                Return "Go To Web Site"
        End Select
    End Get
End Property
```

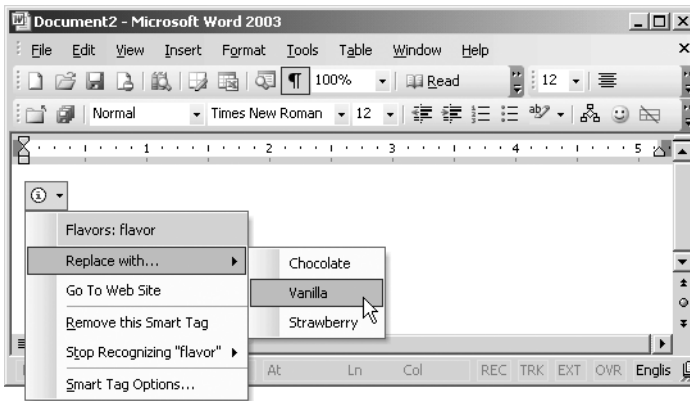


Figure 15-4. Smarts tags in Office 2003 can display cascading menus.

The `IsCaptionDynamic` method should return `True` if the caption is dynamic. If `IsCaptionDynamic` is `True`, the smart tag's `SmartTagInitialize` method executes every time the menu displays.

The `ShowSmartTagIndicator` method should return `True` if you want the smart tag indicator (for example, the purple dotted underline in Word) to be visible. The following code shows the `FlavorSmartTag` example's `ShowSmartTagIndicator` method.




---

**FILE** `Ch15\FlavorsSmartTag\SmartTagAction.vb`

---

```
Public ReadOnly Property ShowSmartTagIndicator(ByVal VerbID As Integer, _
    ByVal ApplicationName As String, ByVal LocaleID As Integer) As Boolean _
    Implements SmartTagLib.ISmartTagAction2.ShowSmartTagIndicator
    Get
        Return True
    End Get
End Property
```

Finally, the `InvokeVerb2` method works much as the `InvokeVerb` method does except it provides an extra `LocaleID` parameter. The following code fragment shows the `FlavorSmartTag` example's `InvokeVerb2` method. It examines its `VerbId` parameter to see which action it should perform. If the action is a replacement, the method calls subroutine `PerformReplacement` to make the replacement. Subroutine `PerformReplacement` replaces the recognized string with the word that the user selected. It does a little work to try to give the replacement word the same case as the word it is replacing.

If the user selected the Go To Web Site action, InvokeVerb2 creates an Internet Explorer application server, points it at the appropriate Web page, and makes it visible.




---

**FILE** *Ch15\FlavorsSmartTag\SmartTagAction.vb*

---

```
' Perform the appropriate action.
Public Sub InvokeVerb2(ByVal VerbID As Integer, _
    ByVal ApplicationName As String, ByVal Target As Object, _
    ByVal Properties As SmartTagLib.ISmartTagProperties, _
    ByVal recognized_text As String, ByVal Xml As String, _
    ByVal LocaleID As Integer) _
    Implements SmartTagLib.ISmartTagAction2.InvokeVerb2
    If VerbID <= 3 Then
        ' Make a replacement.
        PerformReplacement(VerbID, ApplicationName, Target, recognized_text)
    Else
        ' Go to the Web site.
        Dim browser As Object
        browser = CreateObject("InternetExplorer.Application")
        browser.Navigate2("http://www.vb-helper.com/office.htm")
        browser.Visible = True
    End If
End Sub

' Replace the recognized text.
Public Sub PerformReplacement(ByVal VerbID As Integer, _
    ByVal ApplicationName As String, ByVal Target As Object, _
    ByVal recognized_text As String)
    Dim new_text As String

    ' Figure out what to replace the text with.
    Select Case VerbID
        Case 1
            new_text = "chocolate"
        Case 2
            new_text = "vanilla"
        Case 3
            new_text = "strawberry"
    End Select
```



```

' Set the proper case.
If recognized_text = recognized_text.ToLower() Then
    ' Lower case.
    new_text = new_text.ToLower()
ElseIf recognized_text = recognized_text.ToUpper() Then
    ' Upper case.
    new_text = new_text.ToUpper()
Else
    ' Mixed case.
    new_text = StrConv(new_text, VbStrConv.ProperCase)
End If

' Replace the text for different hosts.
If ApplicationName.StartsWith("Word.Application") Then
    Target.Text = new_text
ElseIf ApplicationName.StartsWith("Excel.Application") Then
    Target.Value = new_text
ElseIf ApplicationName.StartsWith("PowerPoint.Application") Then
    Target.Text = new_text
End If
End Sub

```

## Visual Studio .NET Tools for Office

Visual Studio .NET Tools for Office is an SDK that helps integrate Visual Studio .NET code with Office applications. In a nutshell, it lets you use Visual Studio .NET code in more or less the same way you use VBA code.




---

**NOTE** *When this was written, the Visual Studio .NET Tools for Office beta was located at <http://www.microsoft.com/downloads/details.aspx?FamilyID=9e0b1b7c-4ab5-40d2-b4d9-5817ab0bc1e5>. The beta SDK's documentation and samples were at <http://www.microsoft.com/downloads/details.aspx?familyid=2fa2f8fe-a435-4cb8-9c74-0b25a2fa5ac9>. You will probably find this package extremely useful. Of course, in the final release, the help and samples may be included with the SDK.*

---

When it is installed, this SDK adds some new items to the Visual Studio .NET environment's New Project dialog. When you select New►Project, the dialog shown in Figure 15-5 appears. Open the Microsoft Office 2003 Projects folder to see the new choices. Open that folder's Visual Basic Projects subfolder to see Visual Basic .NET selections.

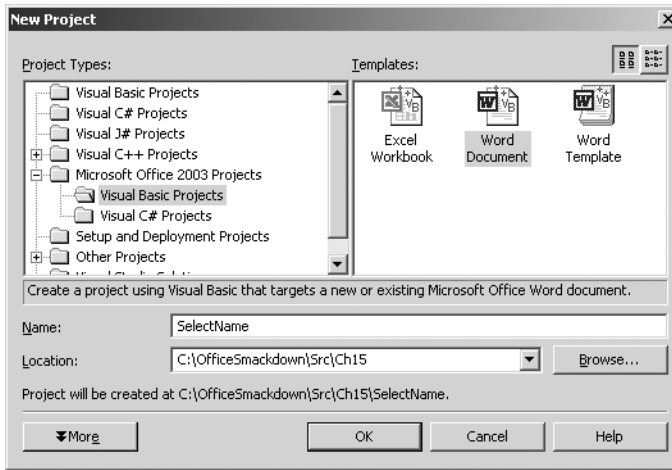


Figure 15-5. Visual Studio .NET Tools for Office adds new options to the IDE's New Project dialog.

The following sections step through an example that shows how Visual Basic .NET code can manipulate a Word document.

## Start a Project

To build a simple example, select New►Project, select the Word Document project, give it a meaningful name, and click OK. This makes Visual Studio display the dialog shown in Figure 15-6.

If you select the “Create new document” option, Visual Studio will make a new Word document for you. By default, the document has the same name as the project and is placed in the same directory.

If you select “Use existing document,” you must enter the Word document's name and location manually. You can use the ellipsis (...) button to browse for it.

Click on the Security Settings link on the left to display the dialog's second tab, shown in Figure 15-7. Leave the box checked if you want Visual Studio to set your local permissions to allow the compiled project to run.

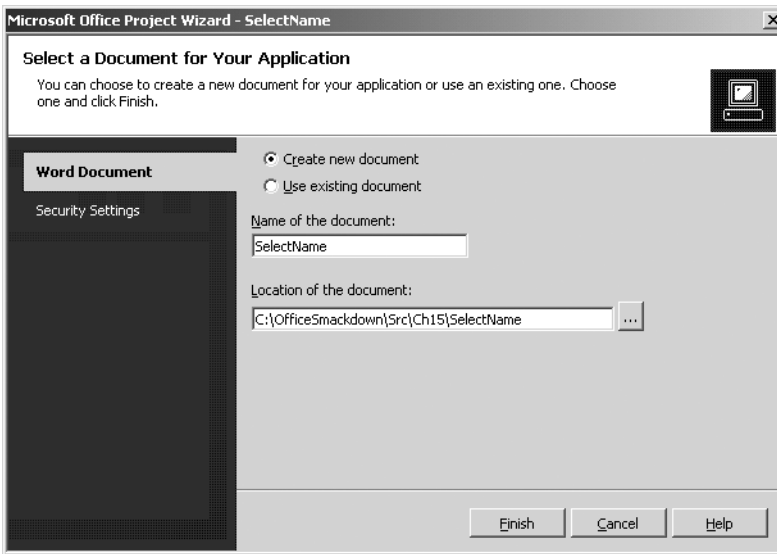


Figure 15-6. Select a Word document for the project to manipulate.

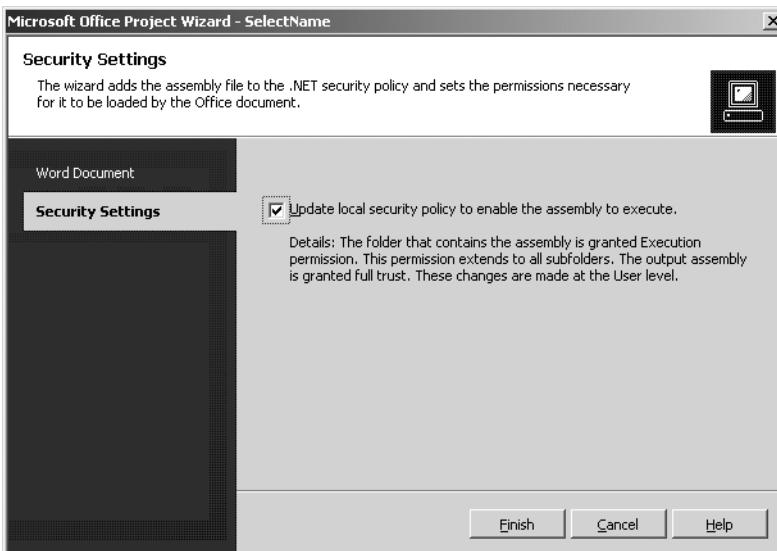


Figure 15-7. Check the box to allow the final compiled assembly to run.



**NOTE** Normally, you'll want to leave this box checked. One reason you might not want Visual Studio to change the security options automatically is if you want to do this yourself, which requires more experience but gives you more options.

When you click Finish, Visual Studio builds the new project (it may take a while, so be patient). In this case, where the project's name is `SelectName`, it builds a `SelectName_bin` directory in addition to the usual files. This directory will contain the compiled DLL file. If you left the box checked in the Security Settings page shown in Figure 15-7, Visual Studio grants trusted status to all DLLs in this directory.

The following code shows the project initially created by Visual Studio, but reformatted slightly so it can fit on the page. A few key points deserve special attention.

The code uses the `WithEvents` keyword to declare the variables `ThisDocument` and `ThisApplication`. That allows the program to catch events for those variables.

Subroutine `_Startup` (note the leading underscore) takes two parameters of type `Object` named `application` and `document`. These are references to the `Word` object model's `Application` and `Document` objects. The subroutine saves these objects into the variables `ThisApplication` and `ThisDocument` so you can catch their events if you like. Because these variables are declared with the detailed `Word.Application` and `Word.Document` data types, they also allow Visual Studio to provide `IntelliSense`.

The code includes two overloaded versions of the `FindControl` function. The first simply invokes the second to find a control on `ThisDocument`. The second version loops through a `Document` object's `InlineShapes` collection looking for the control. If it doesn't find the control, the code looks through the `Document`'s `Shapes` collection.

Finally, the code defines empty `ThisDocument_Open` and `ThisDocument_Close` event handlers for you to fill in if you like.

```
Imports System.Windows.Forms
Imports Word = Microsoft.Office.Interop.Word

' Office integration attribute. Identifies the startup class for the document. _
' Do not modify.
<Assembly: System.ComponentModel.DescriptionAttribute( _
    "OfficeStartupClass, Version=1.0, Class=SelectName.OfficeCodeBehind")>

Public Class OfficeCodeBehind

    Friend WithEvents ThisDocument As Word.Document
    Friend WithEvents ThisApplication As Word.Application

    #Region "Generated initialization code"

        ' Default constructor.
        Public Sub New()
            End Sub
```

```

' Required procedure. Do not modify.
Public Sub _Startup(ByVal application As Object, ByVal document As Object)
    ThisApplication = CType(application, Word.Application)
    ThisDocument = CType(document, Word.Document)

    If (ThisDocument.FormsDesign = True) Then
        ThisDocument.ToggleFormsDesign()
        ThisDocument_Open()
    End If
End Sub

' Required procedure. Do not modify.
Public Sub _Shutdown()
    ThisApplication = Nothing
    ThisDocument = Nothing
End Sub

' Returns the control with the specified name in ThisDocument.
Overloads Function FindControl(ByVal name As String) As Object
    Return FindControl(name, ThisDocument)
End Function

' Returns the control with the specified name in the specified document.
Overloads Function FindControl(ByVal name As String, _
    ByVal document As Word.Document) As Object
    Try
        Dim inlineShape As Word.InlineShape
        For Each inlineShape In document.InlineShapes
            If (inlineShape.Type = _
                Word.WdInlineShapeType.wdInlineShapeOLEControlObject) Then
                Dim oleControl As Object = inlineShape.OLEFormat.Object
                Dim oleControlType As Type = oleControl.GetType()
                Dim oleControlName As String = _
                    CType(oleControlType.InvokeMember("Name", _
                        Reflection.BindingFlags.GetProperty, Nothing, _
                            oleControl, Nothing), String)
                If (String.Compare(oleControlName, name, True, _
                    System.Globalization.CultureInfo.InvariantCulture) = 0) Then
                    Return oleControl
                End If
            End If
        Next
    End Try
End Function

```

```

Dim shape As Word.Shape
For Each shape In document.Shapes
    If (shape.Type = _
        Microsoft.Office.Core.MsoShapeType.msoOLEControlObject) Then
        Dim oleControl As Object = shape.OLEFormat.Object
        Dim oleControlType As Type = oleControl.GetType()
        Dim oleControlName As String = _
            CType(oleControlType.InvokeMember("Name", _
                Reflection.BindingFlags.GetProperty, Nothing, _
                oleControl, Nothing), String)
        If (String.Compare(oleControlName, name, True, _
            System.Globalization.CultureInfo.InvariantCulture) = 0) Then
            Return oleControl
        End If
    End If
Next

Catch Ex As Exception
    ' Returns Nothing if the control is not found.
End Try
Return Nothing
End Function
#End Region

' Called when the document is opened.
Private Sub ThisDocument_Open() Handles ThisDocument.Open

End Sub

' Called when the document is closed.
Private Sub ThisDocument_Close() Handles ThisDocument.Close

End Sub
End Class

```

## *Set Additional Security*

When it creates the project, Visual Studio modifies your local security settings to allow the compiled DLL to run (if you left the box checked on the Security Settings tab in Figure 15-7). That doesn't give the Word document permission to run the DLL's code.



**NOTE** This seems like a rather annoying feature, so it may be changed in the final release of Visual Studio .NET Tools for Office. If so, you need only read this section for more detailed information on the security settings.

To allow Office documents to run your assemblies, open the Control Panel, open Administrative Tools, and run Microsoft .NET Framework 1.1 Configuration. This opens the application shown in Figure 15-8.

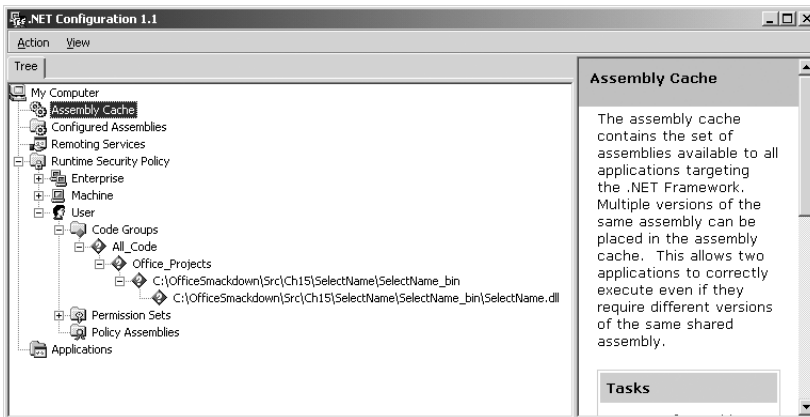


Figure 15-8. Use the .NET Configuration tool to manage assembly security.



**NOTE** Depending on which versions of Visual Studio .NET you have had installed, you may have more than one version of the .NET Configuration tool installed. Be sure you select the right version for your current installation.

Right-click on the Assembly Cache entry and select Add. Select the file msossec.dll and click open. By default, this file should be in C:\Program Files\Microsoft Office\Office11\Addins\msossec.dll.

Now in the .NET Configuration tool, open the path User/Code Groups/All\_Code/Office\_Projects so you see a display similar to the one in Figure 15-8. Inside that folder, you should see an entry for the project you just created. This entry ends in the name of the directory where Visual Studio will place the DLL. In this example, the directory is named SelectName\_bin. This entry indicates that all DLLs inside this directory should be allowed to execute.

Inside that entry is another named after the compiled DLL. In this example, that's `SelectName_bin/SelectName.dll`. This entry grants full trust to that specific DLL.

To allow Office documents to run the code, right-click the outer entry and select **New**. Enter a name and description for the new code group in the dialog as shown in Figure 15-9. Then click **Next**.

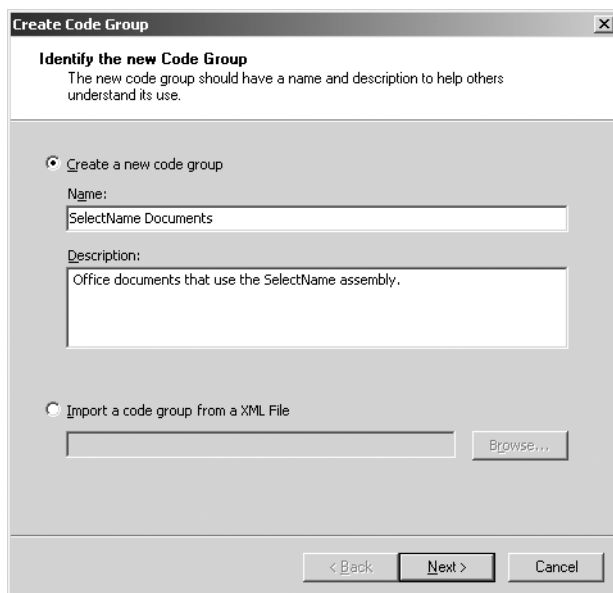


Figure 15-9. Specify the new code group's name and description.

On the next tab, shown in Figure 15-10, scroll the combo box down to the “(custom)” entry. Click **Imports** and select the file `MSOSEC.XML`. By default, this file should be in `C:\Program Files\Microsoft Office\Office11\Addins\MSOSEC.XML`. Click **Next**.

On the next tab, shown in Figure 15-11, select the “Use existing permission set” option, select the **FullTrust** permission, and click **Next**.

On the next tab, click **Finish**. Now the DLL has permission to run and your Office document has permission to run it. Close the .NET Configuration tool and write some code.

Note that changes to the settings do not necessarily take effect immediately. You must close all instances of Word 2003 before they take effect.



**TIP** For more information on using the .NET Configuration tool, see the help provided with the beta SDK's documentation and samples download mentioned earlier in this chapter.



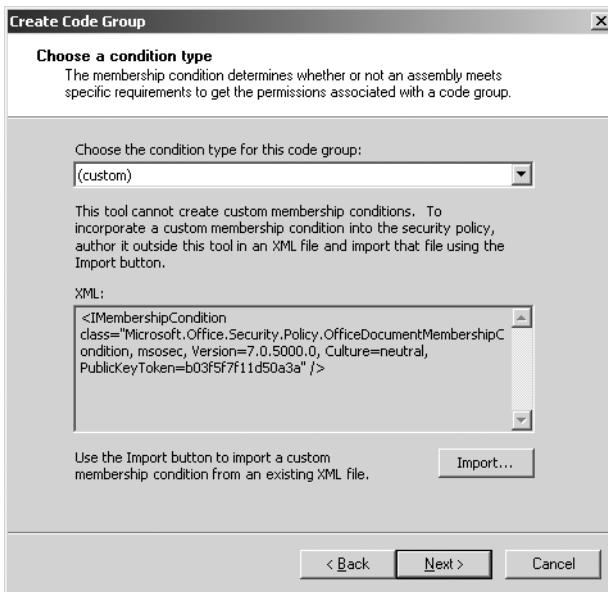


Figure 15-10. Select (custom), click Import, and select the file MSOSEC.XML.

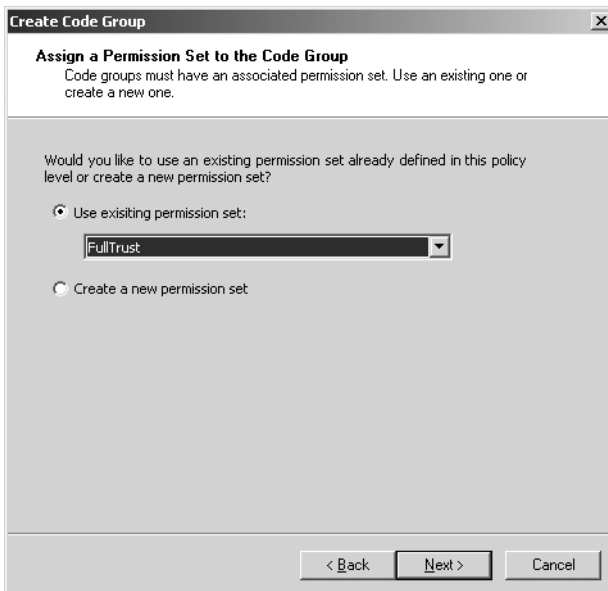


Figure 15-11. Select "Use existing permission set" and FullTrust.

## Prepare the Word Document

After all this setup, writing code to manipulate the `ThisApplication` and `ThisDocument` objects is relatively straightforward. Earlier chapters in this book show how to use the Word object model to modify the document.

This example displays a combo box named `cboNames` that lists several names inside the Word document. When you select a name, the DLL inserts the selection in the bookmark named `bmName`. Before the code can work with the combo box and bookmark, however, you need to add them to the document.

Use Word 2003 to open the document that Visual Studio created for you. Ignore any error messages for now. Select **View** ► **Toolbars** and enable the **Control Toolbox**.

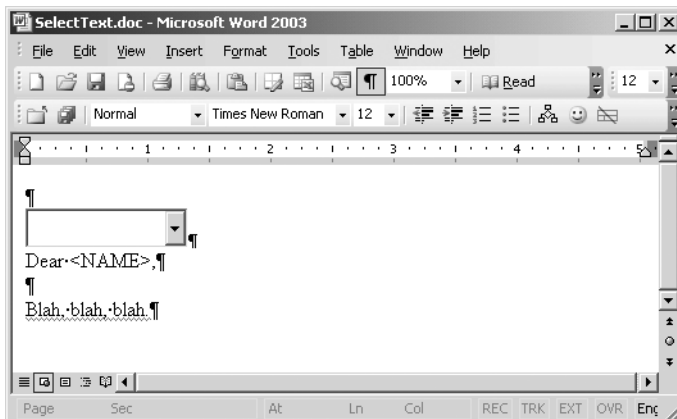
Click the **ComboBox** tool to add a combo box to the document. Right-click the control and select **Properties**. Then in the **Properties** window, change the control's name to `cboName`.

Below the control, add the following text to the Word document.

Dear <NAME>,

Blah, blah, blah.

Highlight the text `<NAME>`, select **Insert** ► **Bookmark**, enter the name `bmName`, and click **Add**. The result should look like Figure 15-12.



*Figure 15-12. This document holds the combo box and bookmark used by the `SelectName` DLL.*

## Customize the Code

Now you're ready to write some code. Start by adding the following statement at the top of the file.

```
Imports MSForms = Microsoft.Vbe.Interop.Forms
```

Unfortunately, the project doesn't contain a reference to this namespace by default, so you must add it. Select Project ► Add Reference and click Browse. Select the file FM20.DLL. On my system, it's at C:\WINNT\SYSTEM32\FM20.DLL.

Click OK. For some strange reason, Visual Studio seems to try to add this reference twice, so it displays an error message. Ignore the error, remove the reference you just added from the Add Reference dialog, and close the dialog. Now you should see MSForms listed in the Project Explorer's References section.

Below the declarations of the ThisApplication and ThisDocument variables, add the following code to define a combo box variable.

```
Private WithEvents cboName As MSForms.ComboBox
```

Edit the ThisDocument\_Open event handler as shown in the following code. This code finds the Word document's cboName control and saves a reference to it in the variable cboName. It then clears that control's list of items and adds some new ones.

```
' Called when the document is opened.
Private Sub ThisDocument_Open() Handles ThisDocument.Open
    ' Find and initialize the ComboBox.
    cboName = FindControl("cboName")
    cboName.Clear()
    Try
        cboName.AddItem(ThisDocument.BuiltInDocumentProperties("Author").Value)
    Catch ex As Exception
    End Try
    cboName.AddItem("George Bush")
    cboName.AddItem("Annie Lennox")
    cboName.AddItem("Sergio Aragonés")
End Sub
```

Finally, add the following code to the class. Because the cboName variable is declared WithEvents, this event handler executes when the user clicks the combo box. The code finds the document's bookmark named bmName and replaces its

text with the user's selection. The code then recreates the bookmark using its new text so the user can change the name again later.

```
' Insert the selected name.
Private Sub cboName_Click() Handles cboName.Click
    ' Find the bmName bookmark.
    Dim bm As Word.Bookmark
    Try
        bm = ThisDocument.Bookmarks("bmName")
    Catch ex As Exception
        MsgBox("Cannot find bookmark bmName")
    Exit Sub
End Try

    ' Replace the bookmark's text with the selection.
    Dim rng As Word.Range = bm.Range
    rng.Text = cboName.Text

    ' Make the new text the bookmark.
    ThisDocument.Bookmarks.Add("bmName", rng)
End Sub
```

## Test the Code

At this point, everything's ready to run in the debugger. When you told Visual Studio to build a Word Document project, it automatically configured the project to use Word as its external program for debugging.

To verify that the program will use Word, go to the Project Explorer, right-click the project name, and select Properties. Open the Configuration Properties item and select Debugging. The “Start external program” option should be selected, and the corresponding text box should contain the path to the Word 2003 executable.




---

**NOTE** *This may seem familiar because it's the same procedure you follow manually when testing a smart tag in Office XP. See the section “Start a New Project” in Chapter 14.*

---

To debug the project, set break points in the code and select Debug ► Start or press F5. At that point, Visual Studio automatically starts Word. When you open

the file `SelectName.doc`, the `ThisDocument_Open` event handler fires and initializes the combo box. If you select an entry, the `cboName_Click` event handler runs and places your selection in the `bmName` bookmark.

After you debug the project, you can open the file `SelectName.doc` directly from Word 2003 without going through Visual Studio.

## *Ponder the Results*

Having built this example project using Visual Basic .NET, it's worth taking a few minutes to think about what it does. This program responds to a Word document's events and uses Word objects to modify the document.

But that's exactly what VBA code does with a lot less effort. Writing VBA code that responds to document events and uses the Word object model to modify a document is relatively straightforward, and you can do it within the IDE provided by Word. It doesn't require a separate development environment (Visual Studio) and a rather cryptic security configuration tool (Microsoft .NET Framework 1.1 Configuration).

On the other hand, Visual Studio .NET Tools for Office has some advantages. It allows you to package code in a compiled DLL, which should generally give you better performance than interpreted VBA code. The system administrator can manage the DLL's security rather than relying on possibly inexperienced Word users to protect themselves against macro viruses. This technique also gives you access to all of Visual Studio .NET's tools, powerful namespaces, and framework. For example, if you want to process XML files or use functions in .NET's cryptographic namespaces, it may be easier to use Visual Studio .NET than trying to perform these chores in VBA.

## **InfoPath**

InfoPath, formerly code named XDocs, is a new addition to Office that lets you build fill-in-the-blank style forms. This is nothing you cannot do using a VBA form or even a Word document that contains controls. The big difference is that InfoPath focuses strongly on XML and related technologies such as XSD, XSL, and scripting (JScript and VBScript).

For example, Figure 15-13 shows a very simple name and address entry form. Using InfoPath, I built this form in just a few minutes. The only customizations I made were to change the fields' names from their default names (`field1`, `field2`) to something meaningful (`txtFirstName`, `txtLastName`), and to make each field required.

After you design a form, you can use InfoPath to run it and fill in the blanks. If you save the results, InfoPath stores the results in a relatively simple XML document. The following code shows the results produced by the form in Figure 15-13. The data begins with the `myFields` element. That element contains the data in its `txtFirstName`, `txtLastName`, and other child elements.

Figure 15-13. InfoPath lets you build fill-in-the-blank style forms.



**FILE** *Ch15\InfoPath\AddressOut.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<?mso-infoPathSolution solutionVersion="1.0.0.5"
  productVersion="11.0.5329" PIVersion="1.0.0.0"
  href="file:///C:\OfficeSmackdown\Src\Ch15\InfoPath\Address.xsn" ?>
<?mso-application progid="InfoPath.Document"?>
<my:myFields xmlns:my=
  "http://schemas.microsoft.com/office/infopath/2003/myXSD/2003-08-08T15:13:41"
  xml:lang="en-us">
  <my:txtFirstName>Rod</my:txtFirstName>
  <my:txtLastName>Stephens</my:txtLastName>
  <my:txtStreet>1234 Programmer Way</my:txtStreet>
  <my:txtCity>Bugsville</my:txtCity>
  <my:cboState>OR</my:cboState>
  <my:txtZip>12345</my:txtZip>
</my:myFields>
```

This output is remarkably easy to understand. You could easily build an application to parse this data and add it to a database or Excel worksheet, use it to generate Word documents, and so forth.




---

**TIP** You can use libraries of XML tools to easily read, write, and modify XML files. Visual Studio .NET has XML tools built in. For more information, see a book about XML programming such as my book *Visual Basic .NET and XML* by Rod Stephens and Brian Hochgurtel (Wiley, 2002).

---

If you select the File menu's Extract Form Files command, InfoPath saves a series of files that define the form's structure. These files include:

- A template XML file that shows the XML file structure containing no data
- A sample data XML file showing the file's structure with default values you assigned to each field filled in
- A schema file that restricts the values allowed in the form's fields
- An XML "manifest" that describes the files that make up the form's solution
- XSL stylesheet files used to generate the form's different views
- Any VBScript or JScript code you attached to the form

InfoPath lets you use JScript or VBScript to provide code for a form much as you can add VBA to an Office document. Unfortunately, for those of us who use Visual Basic, InfoPath uses JScript by default. You can change the scripting language it uses, but not if the form has any script code. I have yet to figure out how to remove any existing script files from a form once you add them, so if you want to use VBScript you should change the scripting language right away before you write any script code.

To use VBScript, select Tools ► Form Options, open the Advanced tab, and select VBScript in the "Form script language" combo box.

To add an event handler to a control, right-click the control and select the Properties option at the bottom of the context menu. On the Data tab, click the Data Validation button. In the Events dropdown, select the event you want to add. For example, a text box supports the events `OnBeforeChange`, `OnValidate`, and

OnAfterChange. After you select an event handler, click the Edit button to open MSE (Microsoft Script Editor).

Initially, InfoPath generates a script file filled with dire warnings and cautions saying InfoPath created the file and that you should not change function names or their parameters. This is good advice, so leave the declarations alone and add the code you need to the event handler's body.

The following example shows how the form Address2 verifies that the value in txtZip looks like a ZIP code. It creates a RegExp object representing a regular expression. It sets the object's Pattern property to "[0-9]{5}" so it matches only strings that consist of exactly five digits. It calls the object's Execute method passing it the new value the txtZip field is about to have, and it examines the results. If the new value doesn't match this expression, the code displays a message and indicates failure by setting the event object's ReturnStatus property to False.




---

**FILE** Ch15\InfoPath\Address2.xsn

---

```
Sub msoxd_my_txtZip_OnBeforeChange(eventObj)
    ' Note: eventObj is an object of type DataDOMEvent.
    ' Use a RegExp object to verify that the new value
    ' looks like a ZIP code.
    Dim reg_exp, matches
    Set reg_exp = New RegExp
    reg_exp.Pattern = "[0-9]{5}"
    Set matches = reg_exp.Execute(eventObj.NewValue)
    If matches.Count < 1 Then
        eventObj.ReturnMessage = "Invalid Zip code format"
        eventObj.ReturnStatus = False
    End If
End Sub
```

InfoPath is a lightweight application intended to let you quickly and easily generate forms that produce XML output that you can integrate easily into other applications. It is strongly based on XML and related technologies such as XSD, XSL, and the Web scripting languages VBScript and JScript. If you want to build a simple form to produce XML results, you may find InfoPath helpful.

On the other hand, VBScript is more cumbersome than VBA, Visual Basic, or other programming languages. Because it provides only one variable type, Variant, the MSE cannot provide IntelliSense. MSE also seems somewhat stark after using the VBA development environment provided by other Office applications.



If you want IntelliSense and the more familiar VBA or Visual Basic programming environment and you don't need XML output, you may want to stick with VBA and Visual Basic.

## Web Services Support

A Web Service is basically a server that responds to remote function calls. A client application sends the service a request, and the service returns a response. The service can perform such tasks as looking up data in its local database, providing access to special purpose hardware, calculating the result of a function, providing price quotes, or storing bug reports in a database. The request and response sent between a client and a Web service are written in XML and packaged using SOAP (Simple Object Access Protocol).

You can use the Office XP Web Services Toolkit 2.0 to add Web Services to your Office XP applications. Office 2003 will provide integrated support for Web Services as an add-on. After it is installed, you can open the IDE's Tools menu, click Web Service References, and provide information identifying the Web Service. Then you can write code to interact with the service. For more information on using this add-on, see the online help. For information about Web Services, see a book on Web Services or XML.



---

**NOTE** *Currently you can download Office XP Web Services Toolkit 2.0 at <http://www.microsoft.com/downloads/details.aspx?FamilyId=4922060F-002A-4F5B-AF74-978F2CD6C798>.*

---

## Smart Documents

A smart document is a document that works with Word 2003 or Excel 2003. The document contains embedded XML tags that give the application some idea of what the user needs to do in different parts of the document. The document can control the Word or Excel task panes to provide context-sensitive help, links to relevant information, or controls such as combo boxes and text boxes to help the user enter appropriate information.

Programmatically, a smart document is similar to a smart tag. You build a DLL containing a class that implements the `ISmartDocument` interface. The methods provided by this interface let the application ask the class about its properties and tell the class when to perform its specialized actions.

Building a smart document is a bit more complicated than building a smart tag, however. The `ISmartDocument` interface requires you to implement more methods, although some can be empty depending on what you want the document to do.

Installing a smart document is also a bit more complicated than installing a smart tag. To install a smart document, you build an expansion pack. The expansion pack includes a manifest written in XML that describes the locations and purposes of the files that make up the expansion pack. It includes such items as the solution's type (smart document), the smart document's name, the DLL file's name, and CLSID values identifying the solution and class in the Registry.

A smart document provides access to the task pane and is sensitive to the user's position in the document. You can implement most of its other capabilities more easily using controls embedded in the document and VBA code.

## Summary

Word's and Excel's new XML tools let your code easily create and manipulate XML data embedded in Word documents and Excel worksheets. The new classes and methods are fairly straightforward, so your main challenge in using them may be to become familiar with the structure and capabilities of XML files.

Office 2003 includes smart tag enhancements that make building smart tags a little easier. The `Recognize2` method provides a `TokenList` parameter that breaks text into tokens for easier processing, and the `VerbCaptionFromID2` method allows cascading smart tag menus, a nice feature.

Visual Studio .NET Tools for Office helps you integrate code written in Visual Studio .NET into an Office application. It seems likely that Office will support Visual Studio .NET in addition to or in place of VBA in some future release. When that happens, integration with externally compiled Visual Studio .NET code should become easier. Until then, the Visual Studio .NET Tools for Office SDK makes this kind of integration possible.

One of the new members of the Office suite, `InfoPath`, makes building fill-in-the-blank style forms relatively painless. It's a bit of an oddball in the Office family because it uses JScript or VBScript as a scripting language rather than VBA. It still may be useful if you want to use forms to generate XML data.

The last topic discussed in this chapter, smart documents, shows promise. A smart document lets an externally compiled DLL provide context-sensitive support for Word or Excel users. Building a smart document is rather difficult (harder than building a smart tag, for example), but this should become easier in future Office releases.

Until then, you can always stick with embedded controls, toolbar and menu items, and VBA code. These aren't as reactive to the user's position within a document, but they are intuitive, easy to build, and easy to use.