VISUAL STUDIO OBA TOOLS

Simplify OBA Development With Interop API Extensions

Andrew Whitechapel, Phillip Hoff, and Vladimir Morozov

CODE DOWNLOAD AVAILABLE FROM THE MSDN CODE GALLERY

Browse the Code Online

THIS ARTICLE DISCUSSES:

- VSTO Power Tools and Office interop APIs
- Using the Outlook, Word, and Excel extensions
- Extending the extensions
- Using the extensions in Visual Basic

THIS ARTICLE USES THE FOLLOWING TECHNOLOGIES:

Office, VSTO Power Tools, LINQ

Contents

Developing applications in C# against the Microsoft Office client application object models can be awkward because these object models use features—such as optional parameters and parameterized properties—that are handled internally in Visual Basic for Applications (VBA) and Visual Basic, but not in C#. Furthermore, Office makes extensive use of loosely typed parameters and return values, which are anathema to a strongly typed system like the Microsoft .NET Framework.

The Microsoft Visual Studio Tools for the Office System (VSTO) Power Tools include a set of Office interop API extension libraries that provide C# developers with support for these features. Developers using Visual Basic will also find the libraries useful, specifically for querying Microsoft Office Outlook items in a type-safe manner. Using these extensions can help to make your code more robust, as well as increase developer productivity.

To introduce you to these Visual Studio Power Tools API extensions, we'll walk through the development of an application that uses the Power Tools to automate Office applications. We'll also look at the .NET Framework 3.0 features being employed by the extensions, as well as extending the extensions for more customized use. We'll even take a look at using the extensions with your Visual Basic code.

Installing the Extensions

The VSTO Power Tools are documented on MSDN and can be obtained from the VSTO System Power Tools v1.0.0.0 download page. The Office Interop API extensions are all in the download file named VSTO_PTExtLibs.exe. When you execute this self-extracting file, it will install the documentation for the extensions under %PROGRAMFILES%\Microsoft VSTO Power Tools 1.0\Office Interop Extensions. The extension assemblies themselves are installed in the Global Assembly Cache (GAC).

The first release of these extensions includes support for Microsoft Office Excel, InfoPath, Project, Outlook, PowerPoint, Visio, and Word in both Office 2003 and the 2007 Office system. The extensions all have a consistent naming convention based on the following pattern:

Microsoft.Office.Interop.<application>.Extensions.dll. For example, the extensions for Excel are defined within the namespace Microsoft.Office.Interop.Excel.Extensions and built into an assembly named Microsoft.Office.Interop.Excel.Extensions.dll.
Note that the extensions are essentially very thin wrappers to the Office primary interop assemblies (PIAs), and you therefore need to have the appropriate Office PIAs installed in order for the extensions to work. The PIAs are installed by default with the 2007 Office system, and optionally with Office 2003. Alternatively, you can use the PIA redistributables. Although the extensions are produced by the same team that produces VSTO, you can use the extensions in any solution that uses Office regardless of whether you are building a VSTO solution.

Using the Extensions

To explore the extensions, we'll build a solution that invokes functionality in three Office applications: Outlook, Excel, and Word. Our application is not a VSTO solution; rather it is a simple Windows Presentation Foundation (WPF) application that automates the Office applications in turn. First, it fetches e-mail data from Outlook and performs some simple calculations on that data; then it feeds the results into Excel to produce a chart; and finally it pastes the chart into a Word document and saves it.
These three tasks have been separated out so that they can be controlled individually by the user via buttons in the main window (see **Figure 1**). This is obviously an artificial mechanism, but it serves an important teaching purpose: to separate each operation so that it can be examined in isolation. Also, any of the tasks can be performed using either the raw PIA objects or the extensions library, in any permutation.

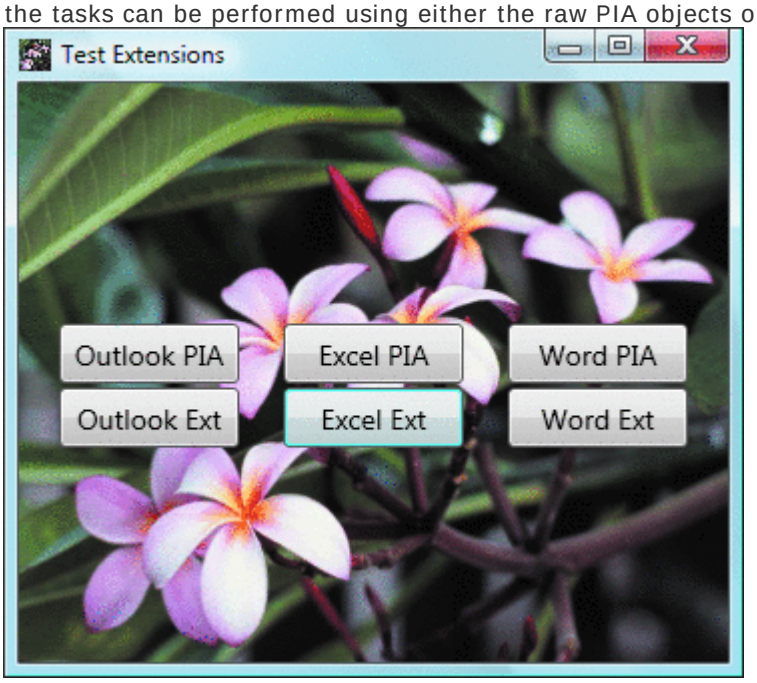

Figure 1 **Sample Application Main Window**

The final output from the application is a Word document in HTML format that contains an Excel chart generated from Outlook data. The Outlook dataset of interest is the size of a selected set of e-mail items, as well as the date and time they were sent and received. The chart plots the size of each item against the elapsed time between sending and receiving it. This document is shown in **Figure 2**.

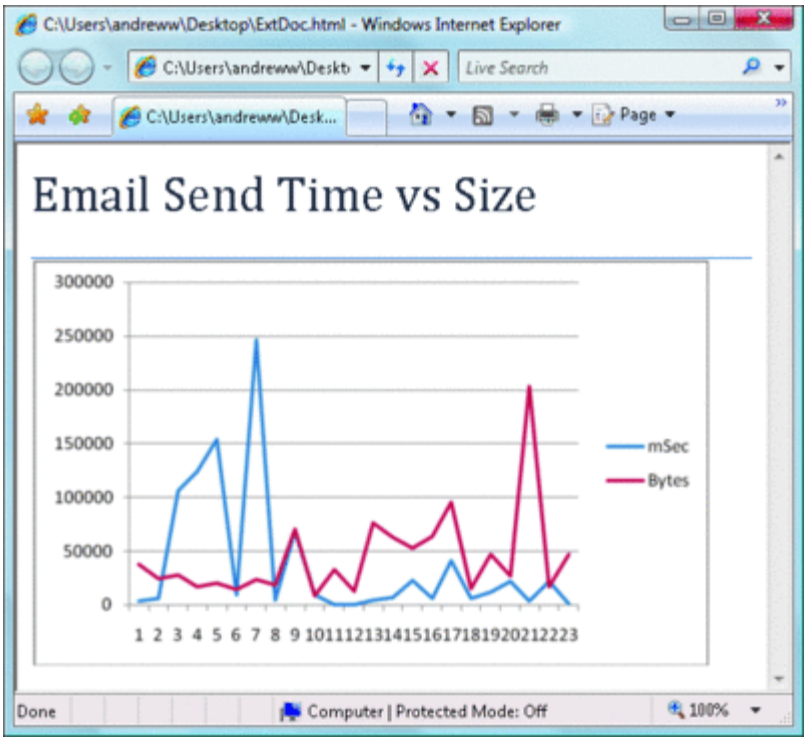

Figure 2 **HTML Document Produced by the Application** (Click the image for a larger view)

As it happens, the sequence of operations in this solution starts with the most complex use of the extensions and ends with the simplest use of the extensions. For this reason, we'll examine the major steps in reverse order, before putting everything together in the correct order. **Figure 3** summarizes the solution requirements and the features of the extensions used to meet each requirement.

Figure 3 Solution Features Mapped to Extension Features

Solution Feature	Office Application	Problem Addressed	Extension Feature
Fetch selected data from e-mail Inbox.	Outlook	Build a filtered query without using an error-prone DASL query string.	LINQ-to-DASL.
Use e-mail data to create a chart.	Excel	Specify Excel cell ranges without using the parameterized get_Range property. Find all negative cell values and set them to zero, in a performant manner.	Parameterized properties.Collections exposed via parameterized properties. LINQ-to-Objects.
Paste the chart into a document and save it.	Word	Invoke Office functionality using strongly typed parameters. Invoke Office functionality without having to pass optional parameters. Use Word features without having to pass parameters explicitly by reference.	Strongly typed method overloading. Strongly typed optional parameters. Nullable types. Object initializers.

Word Extensions

In our example solution, the operations we perform using Word are simple: insert some text and set its style, then move to the end of the document and paste in an Excel chart from the Clipboard. One of the awkward features of the Office object models, which is particularly prevalent in Word, is the use of reference parameters. For example, the Document.Range method in Word takes two parameters: the start position and the end position of the range you want to fetch. In C#, both these parameters must be passed as explicit variables by reference.

The following code shows how to get the range that corresponds to the start of the document, using the raw PIA approach:

Copy Code
```
object startPosition = 0;
object endPosition = 0;
Word.Range r = (Word.Range)doc.Range(ref startPosition,
  ref endPosition);
```
One of the simplest features provided by the extensions is an overload of the Range method that takes two simple parameters which do not need to be passed by reference. Using the extensions, the call to get the start of the document is far simpler. In this example, the extensions avoid the need to pass the parameters by reference—and they also enforce the requirement to pass strongly typed parameters. The start and end position will always be integers, but the Word object model only specifies that they are loosely typed objects. The extensions impose strong typing on the method signature, which therefore brings the benefits of design-time IntelliSense and compile-time type-checking. It also reduces the code from three lines to one:
Copy Code
```
Word.Range r = doc.Range(0, 0);
```
Another minor inconvenience in the Office object models is the use of parameterized properties. It is common in Office to expose properties that take parameters. Bear in mind that all properties exposed from COM servers are really methods. Methods, of course, can take multiple parameters.
In C#, although methods can take parameters, properties cannot. For this reason, the Office PIAs expose parameterized properties in C# as a pair of get_ and set_ methods. For example, the PIA code to set the style property requires you to pass an object parameter (and by reference again). The following raw PIA code continues on from the previous snippet: here we insert some text after the current Range and set its style:
Copy Code
```
r.InsertAfter("Email Send Time vs Size");
r.InsertParagraphAfter();
r = (Word.Range)doc.Paragraphs[1].Range;
object style = Word.WdBuiltinStyle.wdStyleTitle;
r.set_Style(ref style);
```
To do the same thing using the extensions, the code is far more intuitive. As before, it uses strongly typed parameters and a simple method call, without the need for an interim variable and without the set_ or get_ prefix:
Copy Code
```
r.InsertAfter("Email Send Time vs Size");
r.InsertParagraphAfter();
r = (Word.Range)doc.Paragraphs[1].Range;
r.Style(Word.WdBuiltinStyle.wdStyleTitle);
```
Many methods in the Office object models take optional parameters. In Visual Basic, you do not need to supply any parameters that are optional, because the Visual Basic runtime provides a generic parameter on your behalf with a value that tells the Office object to use the default value for the missing parameter. C# does not perform this operation for you. This is another area where the extensions take care of the underlying operation and provide C# developers with an experience that is very similar to the experience of using Visual Basic.
For example, to move to the end of the document and paste something in from the Clipboard, code using the raw PIAs must pass any missing parameters explicitly as System.Type.Missing:
Copy Code
```
object gotoItem = Word.WdGoToItem.wdGoToLine;
object gotoDirection = Word.WdGoToDirection.wdGoToLast;
object missing = Type.Missing;
r = r.GoTo(ref gotoItem, ref gotoDirection,
  ref missing, ref missing);
r.Paste();
```
Using the extensions, on the other hand, we can use the object-initializers feature introduced in C# 3.0 to provide strongly typed named parameters for the values we want to specify and to completely omit any parameters for which we want to use default values. The use of named parameters is another feature that brings a Visual Basic experience to C# developers:

Copy Code
```
r = r.GoTo(
  new DocumentGoToArgs {
    What = Word.WdGoToItem.wdGoToLine,
    Which = Word.WdGoToDirection.wdGoToLast
  });
r.Paste();
```
The usefulness of the optional parameter feature in the extensions becomes more obvious with a method like Document.SaveAs in Word. Using PIA code, you need to supply all 16 parameters. In our example, we want to save the document as an HTML file, which requires us to specify the file name and the enum value for HTML file format in Word, both as loosely typed objects. We don't care about the remaining 14 parameters, but we have to pass them anyway, and we have to pass all parameters by reference:
Copy Code
```
object saveFormat = Word.WdSaveFormat.wdFormatHTML;
doc.SaveAs(ref fileName, ref saveFormat,
  ref missing, ref missing, ref missing, ref missing,
  ref missing, ref missing, ref missing, ref missing,
  ref missing, ref missing, ref missing, ref missing,
  ref missing, ref missing);
```
The extension method again uses strong typing and allows us to completely omit optional parameters:
Copy Code
```
doc.SaveAs(fileName, Word.WdSaveFormat.wdFormatHTML);
```

Excel Extensions

In the example solution, we use Excel to put a chart into the Clipboard so that we can subsequently paste it into the Word document. First, we get hold of the first worksheet in the workbook. It is common in the Office object models to return specific objects typed as generic objects. For example, the Sheets collection may contain both Worksheet objects and Chart objects, so when you specify a particular item in the collection, the Excel PIA returns it to you as a generic object, which you then need to cast to the specific Sheet type you are expecting:
Copy Code
```
Excel.Worksheet sheet = (Excel.Worksheet)book.Sheets[1];
```
A general theme in the extensions is the use of strong typing throughout, and this extends to the type of object returned from heterogeneous collections, without forcing the developer to make a speculative cast on the returned object. Apart from simplifying the developer's code, this also moves the type-checking from runtime to compile time, which improves the robustness of the application and reduces the cost of the test. With the extensions, you can index into the collection using either an integer or the name of the sheet:
Copy Code
```
//Excel.Worksheet sheet =
//  book.Sheets.Item<Excel.Worksheet>("Sheet1");
Excel.Worksheet sheet =
  book.Sheets.Item<Excel.Worksheet>(1);
```
In the example, we want to get specific cell ranges and insert some data. We will have two columns of data: one for the elapsed time in milliseconds between sending and receiving an e-mail message and one for the size of the e-mail in bytes. After inserting the column headings, we then insert the e-mail data into a grid of cells.
Using the PIA approach, we again have to use parameterized properties for Range and Offset. Note that the emailData variable is an array of strings, which we will populate in the complete solution from e-mail data in Outlook:
Copy Code
```
sheet.get_Range("A1", missing).Value2 = "mSec";
sheet.get_Range("B1", missing).Value2 = "Bytes";
Excel.Range firstCell = sheet.get_Range("A2", missing);
Excel.Range lastCell;
```

```
for (int r = 0; r < this.emailData.Length; r++) {
  String[] cellValues = this.emailData[r].Split(',');
  for (int c = 0; c < cellValues.Length; c++) {
    lastCell = (Excel.Range)firstCell.get_Offset(r, c);
    lastCell.Value2 = cellValues[c];
  }
}
```

Using the extensions, we can use the strongly typed Range method. Note that the extensions library covers most of the Office object model, but not all of it. So, in this example, we can use the extension for the Range property, but we have to use the PIA Offset property—unless we extend the extensions (more on that later):

Copy Code

```
sheet.Range("A1").Value2 = "mSec";
sheet.Range("B1").Value2 = "Bytes";
firstCell = sheet.Range("A2");
```

Next, we want to find any negative values in the sheet and set them to zero. We can do this in a simple way, using the PIA:

Copy Code

```
Excel.Range chartData = sheet.get_Range("A2", lastCell);
foreach (Excel.Range cell in chartData) {
  if (((double)cell.Value2) < 0) {
    cell.Value2 = 0;
  }
}
```

Alternatively, we can do this using a SQL-like query, using the extensions. This uses the LINQ-to-Objects functionality in the extensions library. Note that LINQ is optimized for performance and uses a delayed execution model where the query is not actually executed until you begin to enumerate the resultset. In some scenarios, this can dramatically improve the performance of your application. That said, it is sometimes more efficient to force execution of the query by calling ToList or ToArray and then operate on the result—for instance if you intend to iterate over the resultset several times. In this example, the search condition is very simple (cell value less than zero), but you can see how a SQL-like query would become more useful as the complexity of the search conditions increases:

Copy Code

```
Excel.Range chartData = sheet.Range("A2", lastCell);
var cells = (from c in chartData.Items()
             where ((double)c.Value2) < 0
             select c).ToArray();
foreach (Excel.Range cell in cells) {
  cell.Value2 = 0;
}
```

Next, we create a simple line chart from the mSec and Bytes columns and copy it to the Clipboard. With the PIA approach, we have to use loose typing, explicit optional parameters, and parameterized properties:

Copy Code

```
Excel.Shape chart = sheet.Shapes.AddChart(
  Excel.XlChartType.xlLine,
  missing, missing, missing, missing);
chart.Select(missing);
excel.ActiveChart.SetSourceData(
  sheet.get_Range("A1", lastCell), missing);
excel.ActiveChart.ChartArea.Copy();
```

Out of the box, the extensions only help us here with the parameterized Range property—they don't provide overloads for Shapes.AddChart or Chart.SetSourceData. This is another case where we might want to extend the extensions.

Outlook Extensions

You can search, filter, and query Outlook items using the DAV Searching and Locating (DASL) query language. The language is similar to SQL and is quite powerful. However, it has one serious drawback: it

requires you to compose a string and pass this as a filter to the Outlook Restrict or Find methods, and because the query is a string, there can be no design-time or compile-time support. This means that you cannot be sure that the string really represents the query you intended until you execute it at run time. You cannot even be sure that the string is a correctly formed query. DASL queries use property URNs, which are non-intuitive, typically include obscure hex values, are not well-documented, and are easily mistyped. If your query uses string patterns, date, or time conditions, you must observe wildcard and formatting rules, which are even more easily mistyped. Clearly, these query strings can rapidly become complex and difficult both to construct and to parse, slowing down productivity, increasing test cost, and making code maintenance difficult.

In our example, we set up a DASL filter, shown in **Figure 4**, to find all Inbox items that are e-mail messages (that is, their MessageClass is based on IPM.Note), where the subject starts with "RE:" and were received in the last 30 days. We use Windows Desktop Search (WDS) if it is available, because this provides faster searching. WDS is installed by default on Windows Vista, but is optional on Windows XP and Windows Server 2003. Note that DASL queries always perform date-time comparisons in Coordinated Universal Time (officially abbreviated as UTC), so if you use a date literal in a comparison string, you must use its UTC value for the comparison.

Figure 4 DASL Filter to Find E-Mail Messages

Copy Code

```
string filter;
if (outlook.Session.DefaultStore.IsInstantSearchEnabled) {
  filter =
    @"@SQL=((""http://schemas.microsoft.com/mapi/proptag/0x001a001e"""
    + "CI_STARTSWITH 'IPM.Note')"
    + @" AND (""urn:schemas:httpmail:subject"" CI_STARTSWITH 'RE:'))"
    + @" AND (""urn:schemas:httpmail:datereceived"" >= '"
    + (DateTime.Now.ToUniversalTime()
    - new TimeSpan(30, 0, 0, 0)).ToString("g") + @"'))";
}
else {
  filter =
    @"@SQL=((""http://schemas.microsoft.com/mapi/proptag/0x001a001e"""
    + "LIKE 'IPM.Note%')"
    + @" AND (""urn:schemas:httpmail:subject"" LIKE 'RE:%'))"
    + @" AND (""urn:schemas:httpmail:datereceived"" >= '"
    + (DateTime.Now.ToUniversalTime()
    - new TimeSpan(30, 0, 0, 0)).ToString("g") + @"'))";
}

Outlook.Items folderItems = folder.Items;
Outlook.Items filteredItems = folderItems.Restrict(filter);
StringBuilder builder = new StringBuilder();

foreach (Outlook.MailItem item in filteredItems) {
  builder.AppendLine(String.Format("{0},{1},{2}",
    item.SentOn, item.ReceivedTime, item.Size));
}

CalculateElapsedTime(builder.ToString());
```

Once we have retrieved all the items of interest, we will compose a string for each item that consists of the time taken to send the e-mail and the size of the e-mail (using a custom method, CalculateElapsedTime, which simply performs string and DateTime operations, and does not use the Office object models—this method is not listed here, but you can find it in the accompanying source-code download).

Using the extensions, we can dispense with the string query and instead perform the same search using a LINQ query, which will bring all the benefits of design-time IntelliSense and compile-time type and syntax-checking—very different from the DASL string query experience. Note one limitation: the extensions do not

expose the Outlook item's Size property, so we use the Length of the Body property instead (this is another case where extending the extensions would make sense):

Copy Code

```
var filteredItems = (
  from item in folder.Items.AsQueryable<Mail>()
  where item.MessageClass.StartsWith("IPM.Note")
  && item.Subject.StartsWith("RE:")
  && item.DateReceived >=
  DateTime.Now.ToUniversalTime() - new TimeSpan(30, 0, 0, 0)
  select item).ToList();
StringBuilder builder = new StringBuilder();

foreach (Mail item in filteredItems) {
  builder.AppendLine(String.Format("{0},{1},{2}",
    item.Date, item.DateReceived, item.Body.Length));
}

CalculateElapsedTime(builder.ToString());
```

Extension Internals

Now that we've seen how much simpler and more error-free the developer's coding experience is with the extensions, let's look at how this simplicity and robustness are achieved under the covers. Consider first the Document.SaveAs method in Word. This takes 16 optional parameters, and in most cases you only need to specify a very small number of these parameters (or, in some cases, none).

To write an extension method, you simply write a static method that takes the object you want to extend as its first (or only) parameter. So, to provide a strongly typed SaveAs method for the Document object, you can write a method called SaveAs that takes a Document object as its first parameter. You can also provide overloads for your extension methods so that each overload is used to simplify the developer's coding experience, but the real functionality is ultimately invoked on the underlying PIA Document object itself. This is how the extensions are implemented, as shown in the code listings for the SaveAs method in **Figure 5**. At design-time in Visual Studio, the extension methods appear to the developer to be methods of the object that they extend, and are available through IntelliSense and autocomplete. IntelliSense will flag extension methods with "(extension)," as shown in **Figure 6**.

Figure 5 Overloaded SaveAs Methods

Copy Code

```
public static void SaveAs(this Document doc) {
  SaveAs(doc, (string) null);
}

public static void SaveAs(this Document doc, string fileName) {
  SaveAs(doc, fileName, null);
}

public static void SaveAs(this Document doc, string fileName,
   WdSaveFormat? fileFormat) {
  SaveAs(doc, new DocumentSaveAsArgs {
    FileName = fileName, FileFormat = fileFormat });
}

public static void SaveAs(this Document doc, DocumentSaveAsArgs args) {
  doc.SaveAs(
    ref args.FileNameInternal,
    ref args.FileFormatInternal,
    ref args.LockCommentsInternal,
    ref args.PasswordInternal,
    ref args.AddToRecentFilesInternal,
```

```
    ref args.WritePasswordInternal,
    ref args.ReadOnlyRecommendedInternal,
    ref args.EmbedTrueTypeFontsInternal,
    ref args.SaveNativePictureFormatInternal,
    ref args.SaveFormsDataInternal,
    ref args.SaveAsAOCELetterInternal,
    ref args.EncodingInternal,
    ref args.InsertLineBreaksInternal,
    ref args.AllowSubstitutionsInternal,
    ref args.LineEndingInternal,
    ref args.AddBiDiMarksInternal);
}
```

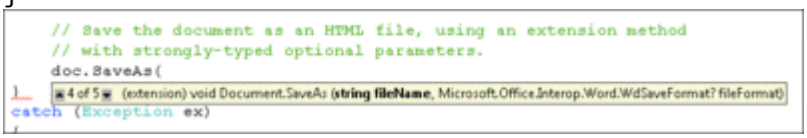

Figure 6 **IntelliSense for an Extension Method in Visual Studio** (Click the image for a larger view)

You can also take advantage of the nullable types feature of C#, where a strongly typed parameter can be passed as null (note that this is not the same as passing a loosely typed object with the System.Type.Missing value). In the third overload SaveAs method shown in **Figure 5**, the WdSaveFormat parameter is specified as nullable (declared using the ? syntax), which allows us to invoke this method and pass null for this parameter. The real work of saving the document is ultimately done in the overload that takes a DocumentSaveAsArgs object. As we have seen, this uses object initializers, to allow the developer to provide values for the interesting parameters by name and to omit any that can be defaulted.

The DocumentSaveAsArgs class is derived from the SaveAsArgs extensions class. The argument values themselves are stored within internal fields of the class, such as FileNameInternal, and these are used directly by the extension methods. The developer uses the public properties, such as FileName, rather than the internal fields. This design arises because of the requirement in most methods in the Word object model to pass parameters by reference. The argument classes in the extensions library for the other Office applications do not use this extra layer:

Copy Code

```
public abstract class SaveAsArgs : ExtensionArgs {
  internal object FileNameInternal = Type.Missing;

  public string FileName {
    get {
      return ToReference<string>(FileNameInternal);
    }
    set {
      FileNameInternal = ToObject(value);
    }
  }
  // code omitted for brevity.
}
```

SaveAsArgs is, in turn, derived from ExtensionArgs, which includes code to convert between real types and their boxed equivalents and includes logic for handling System.Type.Missing:

Copy Code

```
protected T ToReference<T>(object obj) where T : class {
  return (obj != null && obj != Type.Missing) ? (T) obj : null;
}
```

Parameterized properties are also handled by overloaded extension methods in the extensions library—as before, these use the underlying PIA objects for the ultimate functionality. For example, the Excel.Range extensions provide three overloads that use the get_Range property method under the covers:

Copy Code

```
public static Range Range(this Worksheet worksheet, string name) {
  return worksheet.get_Range(name, Type.Missing);
```

```
}

public static Range Range(this Worksheet worksheet,
  string name1, string name2) {
  return worksheet.get_Range(name1, name2);
}

public static Range Range(this Worksheet worksheet, string range1, Range range2) {
  return worksheet.get_Range(range1, range2);
}
```

To support LINQ-to-DASL queries in Outlook, the extensions library provides an abstract base class to represent the data source for the query: OutlookItemSource. This class implements the IQueryable<T> and IQueryProvider interfaces used by LINQ. The extensions also provide derived classes ItemsSource and ApplicationSource for performing queries on the Items collection of a single folder and queries across multiple folders, respectively:

Copy Code

```
public abstract class OutlookItemSource<T> : IQueryProvider,    IQueryable<T> {
  public IQueryable<TElement> CreateQuery<TElement>    (Expression expression) {
    MethodCallExpression call = expression as MethodCallExpression;
    switch (call.Method.Name) {
      case "Where": return new
        OutlookItemSourceWhereHandler<TElement>(
          this, expression, _builder, _search);
      case "Select": return new
        OutlookItemSourceSelectHandler<TElement>(
          this, expression);
    }
  }
  // code omitted for brevity.
}
```

The OutlookItemSource class uses other internal extensions classes to build a DASL query string from the LINQ query. For example, the ParseMethodCallExpression method converts LINQ expressions such as Contains to DASL query sub-strings such as (LIKE '%'):

Copy Code

```
private static void ParseMethodCallExpression(MethodCallExpression call) {
  if (call.Method.DeclaringType == typeof(string)) {
    string format = null;
    switch (call.Method.Name) {
      case "Contains": format = @"({0} LIKE '%{1}%')"; break;
      case "StartsWith": format = @"({0} LIKE '{1}%')"; break;
      case "EndsWith": format = @"({0} LIKE '%{1}')"; break;
    }
    // code omitted for brevity.
  }
}
```

The OutlookItemProperty attribute provides a mapping between DASL properties and properties on the Outlook item interfaces. This attribute is used on the public properties exposed from the OutlookItem class, and this is how the extensions allow the developer to avoid the use of the obscure URN/hex tag strings to specify target properties:

Copy Code

```
public class OutlookItem : IOutlookItemWrapper {
  [OutlookItemProperty(
    "http://schemas.microsoft.com/mapi/proptag/0x001a001e")]
  public virtual string MessageClass
    { get { return GetProperty("MessageClass") as string; } }
  // code omitted for brevity.
```

```
}
```

Extending the Extensions

Knowing how the extensions are implemented internally, you can easily extend them. Some objects and methods in the Office object model are not covered by the extensions, and in some cases you might even want to provide user-defined extensions.

From our example application, the most obvious cases are the Excel methods Range.get_Offset, Shapes.AddChart, Shape.Select and Chart.SetSourceData. For all of these, it would be trivial to provide extension methods. Recall that these simply need to be static methods that take as their first parameter an object of the type which you want to extend as shown in **Figure 7**.

Figure 7 Extending the Excel Methods

Copy Code

```
public static class RangeExtensions {
  public static Excel.Range Offset(
    this Excel.Range cells, int rowOffset, int columnOffset) {
    return cells.get_Offset(rowOffset, columnOffset) as Excel.Range;
  }
}

public static class ShapesExtensions {
  public static Excel.Shape AddChart(
    this Excel.Shapes shapes, Excel.XlChartType chartType) {
    return shapes.AddChart(chartType,
      Type.Missing, Type.Missing, Type.Missing, Type.Missing);
  }
}

public static class ShapeExtensions {
  public static void Select(this Excel.Shape shape) {
    shape.Select(Type.Missing);
  }
}

public static class ChartExtensions {
  public static void SetSourceData(this Excel.Chart chart,
    Excel.Range range) {
    chart.SetSourceData(range, Type.Missing);
  }
}
```

Note that, taking the Shapes.AddChart method as an example, this extension method works by virtue of having a different number of arguments from the PIA AddChart method (or any of the extensions overloads). If you tried to provide an extension method that had the same arguments, but was simply strongly typed, the compiler would never call it (because it favors native methods, regardless of the strength of the argument types). We can do this with Word because of its use of by-reference arguments; by providing by-value extensions of Word methods, the compiler is always able to differentiate between the two.

The extensions library for Excel actually includes a number of methods that are unreachable by the compiler in this way. It might seem strange to provide methods that the compiler will not use, but they will still show up in IntelliSense and will therefore help the user identify the real types of the arguments. Also, the developer could use these methods with normal static method syntax if desired. With these custom extensions, we can simplify the application code:

Copy Code

```
//lastCell = (Excel.Range)firstCell.get_Offset(r, c);
lastCell = (Excel.Range)firstCell.Offset(r, c);

//chart = sheet.Shapes.AddChart(
//    Excel.XlChartType.xlLine, missing, missing, missing, missing);
```

```
chart = sheet.Shapes.AddChart(Excel.XlChartType.xlLine);
//chart.Select(missing);
chart.Select();
//excel.ActiveChart.SetSourceData(sheet.Range("A1", lastCell), missing);
excel.ActiveChart.SetSourceData(sheet.Range("A1", lastCell));
```

Apart from simply allowing the developer to provide additional extension methods, the extensions library itself supports a degree of extensibility. For example, the LINQ-to-DASL features support pluggable logging (see **Figure 8**). The ItemsSource class provides a Log property that you can set to any object derived from System.IO.TextWriter. In the example, we can write a simple class that outputs to the debug window. This will then be used internally when the extensions code has completed building the DASL query string from the LINQ query—and this string will be output to the debug window.

Figure 8 Logging with LINQ-to-DASL

Copy Code

```
// Provide a logging class for our LINQ-to-DASL queries.
internal class DebuggerWriter : TextWriter {
  public override Encoding Encoding {
    get { throw new NotImplementedException(); }
  }

  public override void WriteLine(string value) {
    Debug.WriteLine(value);
  }
}

// Subclass the extensions Mail class so that we can expose a Size
// property that maps to the real Outlook size property.
internal class MailEx : Mail {
  [OutlookItemProperty("http://schemas.microsoft.com/mapi/proptag/0x0E080003")]
  public int Size { get { return Item.Size; } }
}

// Create the ItemsSource manually (this is
// what Items.AsQueryable() does implicitly).
var source = new ItemsSource<MailEx>(folder.Items);

// Set the Log property of the ItemsSource to a TextWriter.
// It will be given the DASL query string immediately before
// the query is executed in Outlook.
source.Log = new DebuggerWriter();

//var filteredItems = (from item in folder.Items.AsQueryable<Mail>()
var filteredItems = (from item in source
                     where item.MessageClass.StartsWith("IPM.Note")
                     && item.Subject.StartsWith("RE:")
                     && item.DateReceived >=
                     DateTime.Now.ToUniversalTime() -                        new
TimeSpan(30, 0, 0, 0)
                     select item).ToList();

//foreach (Mail item in filteredItems)
foreach (MailEx item in filteredItems) {
  builder.AppendLine(String.Format("{0},{1},{2}",
    //item.Date, item.DateReceived, item.Body.Length));
    item.Date, item.DateReceived, item.Size));
}
```

At the same time, we can subclass the extensions Mail class to expose a public property for the underlying Outlook Size property, using the OutlookItemProperty attribute to establish the mapping. Recall that we previously had to use Body.Length as a workaround because Size is not exposed by the extensions by default.

With these additional extension enhancements in place, our example application is complete. We use LINQ-to-DASL to fetch filtered Outlook mail item data, feed that data into Excel to produce a chart, and copy that chart into a Word document, which we finally save as an HTML file.

Note that Outlook 2007 introduced the Table object, which is a performance-optimized mechanism for enumerating folder items. It allows you to specify properties as table columns, and to filter and sort items as rows in the table. From Outlook 2007 onwards, you should use the Table mechanism in preference to the traditional method of working with collections of items.

To get a table, you use the same filter string that you would use with the Restrict or Find methods, but you use this with the GetTable method instead. From the code shown in **Figure 9**, it should be clear that there would be little advantage in using the existing version of the LINQ-to-DASL extensions with an Outlook Table, because you still need to use a query string to get a Table, and Table filtering is done by specifying Table Columns. The LINQ-to-DASL approach in the extensions library can be used for either Outlook 2003 or Outlook 2007, but if you are targeting Outlook 2007 only, the Table approach is likely to offer better performance, especially for large data sets. Example code for this approach is in **Figure 9**.

Figure 9 Using Table in Outlook 2007

Copy Code

```
//Outlook.Items folderItems = folder.Items;
//Outlook.Items filteredItems = folderItems.Restrict(filter);
//foreach (Outlook.MailItem item in filteredItems) {
//  builder.AppendLine(String.Format("{0},{1},{2}",
//    item.SentOn, item.ReceivedTime, item.Size));
//}

// Get a table of mail items, remove the default column set,
// and add the specific columns we're interested in instead.
table = folder.GetTable(filter, Outlook.OlTableContents.olUserItems);
table.Columns.RemoveAll();
table.Columns.Add("SentOn");
table.Columns.Add("ReceivedTime");
table.Columns.Add("Size");

// Iterate the table rows.
while (!table.EndOfTable) {
  Outlook.Row nextRow = table.GetNextRow();
  builder.AppendLine(String.Format("{0},{1},{2}",
    nextRow["SentOn"], nextRow["ReceivedTime"], nextRow["Size"]));
}
```

VSTO Resources

To learn even more about using VSTO to build your own Office-based applications, see these articles from the archives of *MSDN Magazine*:

VSTO 3.0: Developing Office Business Apps with Visual Studio 2008 by Steve Fox

VSTO: Build Office-Based Solutions Using WPF, WCF, and LINQ by Andrew Whitechapel

Office Apps: Extend Your VBA Code with VSTO by Paul Stubbs and Kathleen McGrath

Using the Extensions in Visual Basic

The main benefit of the extensions is to provide C# developers with a coding experience that is similar to Visual Basic. That said, some of the extensions—in particular, the LINQ-to-DASL extensions—also bring benefits to developers working in Visual Basic (or indeed any other managed language). For example, it is

beneficial in any language to use a strongly typed LINQ query instead of a string-based DASL query. **Figure 10** shows a Visual Basic version of the Outlook query operation.

Figure 10 Outlook Query in Visual Basic

Copy Code

```
Dim source As New ItemsSource(Of MailEx)(folder.Items)
Dim builder As New StringBuilder()

Dim filteredItems = ( _
  From item In source _
  Where item.MessageClass.StartsWith("IPM.Note") _
  AndAlso item.Subject.StartsWith("RE:") _
  AndAlso item.DateReceived >= _
  (DateTime.Now.ToUniversalTime() - New TimeSpan(30, 0, 0, 0)) _
  Select item).ToList()

For Each filteredItem In filteredItems
  builder.AppendLine(String.Format("{0},{1},{2}", _
  filteredItem.Date, filteredItem.DateReceived, filteredItem.Size))
Next
MessageBox.Show(builder.ToString())
```

Note one of the differences in the way C# and Visual Basic generate expression trees. C# uses the base type—OutlookItem, in our example—which defines a mapped property for MessageClass. On the other hand, Visual Basic uses the most derived type—Mail—which defines an override for MessageClass that is not attributed with a property mapping. For this reason, we have to define another override of MessageClass in our MailEx class (see **Figure 11**) that is attributed with a property mapping (this is another possible place for improvement in future releases).

Figure 11 MailEx for Visual Basic

Copy Code

```
Friend Class MailEx
  Inherits Mail

  <OutlookItemProperty(
"http://schemas.microsoft.com/mapi/proptag/0x0E080003")> _
  Public ReadOnly Property Size() As Integer
    Get
      Return MyBase.Item.Size
    End Get
  End Property

  <OutlookItemProperty(
"http://schemas.microsoft.com/mapi/proptag/0x001a001e")> _
  Public Overrides ReadOnly Property MessageClass() As String
    Get
      Return MyBase.Item.MessageClass
    End Get
  End Property
End Class
```

Wrapping Up

In this article, we showed how C# developers can use the Office interop API extensions to streamline Office development. The extensions provide a thin, strongly typed layer over the loosely typed Office object models, which makes your code less error-prone and more robust, and can significantly reduce the cost of testing and maintenance. We also showed how to extend the extensions for custom scenarios and how to use them in Visual Basic.

Andrew Whitechapel is a Program Manager in the Visual Studio Business Applications team, responsible for architecting new Office development features in Visual Studio.

Phillip Hoff is a Software Development Engineer in the Visual Studio Business Applications team and is one of the original authors of the Office Interop API Extensions library. He is currently working on SharePoint tool integration with Visual Studio.

Vladimir Morozov is a developer in the Visual Studio Business Applications team. He worked together with Phillip Hoff on envisioning and creation of the Office Interop API Extensions library. He is currently working on SharePoint tooling for Visual Studio.