VBA to VB.Net XLL add-ins with Excel-Dna

Patrick O'Beirne Mail3 at sysmod.com
Web: http://www.sysmod.com
Blog: http://sysmod.wordpress.com
LinkedIn: http://www.linkedin.com/in/patrickobeirne

When the time comes to expand your skill set beyond VBA, the Microsoft path is to DotNet, either VB.Net or C#. This describes the easy route to VB using either or both of two open source libraries, Excel-Dna and NetOffice. Once in DotNet, you can then acquire C# skills for which there is more demand in the marketplace.

Excel-DNA (Excel Dot Net Assembly)

http://excel-dna.net/
Quote from the home page of this project by Govert van Drimmelen:
Excel-Dna is an independent project to integrate .NET into Excel. The primary target is the Excel user who currently writes VBA code for functions and macros, and would like to start using .NET. Also, C/C++ based .xll add-in developers who want to use the .NET framework to develop their add-ins.

The Excel-Dna Runtime is free for all use, and distributed under a permissive open-source license that also allows commercial use.

Excel-Dna is developed using .NET, and users have to install the freely available .NET Framework runtime. The integration is by an Excel Add-In (.xll) that exposes .NET code to Excel. The user code can be in text-based (.dna) script files (C#, Visual Basic or F#), or compiled .NET libraries (.dll). Excel versions '97 through 2010 can be targeted with a single add-in.

The latest Excel-Dna version is available on the CodePlex site.
http://ExcelDna.codeplex.com also has links to tutorials
http://exceldna.codeplex.com/wikipage?title=Reference is a quick reference
http://groups.google.com/group/exceldna the discussion list for primary support

All applications use the ExcelDna.xll addin. The stages of learning described below successively add files:

1) Using only a text file (.dna) which includes the source code text.
2) Add an external DLL which you create from VB.net source code and compile using either the vbc.exe compiler or an IDE (Integrated Development Environment).
3) Ease the transition from the VBA Excel object model to VB.Net objects using either the MS Primary Interop Assemblies (PIA) or third party libraries such as NetOffice.
4) An add-in using the Excel Ribbon

If you do not already have a favourite text file editor, I recommend Notepad++
http://notepad-plus-plus.org/

Example: Create a user-defined function in Visual Basic

Stage 1: using only a .DNA text file

Getting Started with Excel-Dna – extracted from the web page:
http://ExcelDna.codeplex.com/wikipage?title=Getting%20Started

Do this first:

- Install the Microsoft .NET Framework Version 4.0 Redistributable Package.
- Install the most recent release of ExcelDna, unzip in a convenient directory.

Make a copy of ExcelDna.xll in a convenient directory, calling the copy TestDna.xll. Create a new text file, called TestDna.dna (the same prefix as the .xll file), with contents:

```
<DnaLibrary>
<![CDATA[
    Public Module MyFunctions
        Function AddThem(x, y)
            AddThem = x + y
        End Function
    End Module
]]>
</DnaLibrary>
```

Load TestDna.xll in Excel (either File->Open or Tools->Add-Ins and Browse...). You should be prompted whether to Enable Macros; click Enable. There should be an entry for AddThem in the function wizard, under the category TestDna.

Enter `=AddThem(4,2)` into a cell - you should get 6.
Enter `=AddThem("a","b")` into a cell - you should get ab.

Troubleshooting

- If you are not prompted to Enable Macros and nothing else happens, your security level is probably on High. Set it to Medium.
- If you get a message indicating the .Net runtime could not be loaded, you might not have the .NET Framework installed. Install it.
- If a window appears with the title 'ExcelDna Error Display' then there were some errors trying to compile the code in the .dna file. Check that you have put the right code into the .dna file. Eg,"error BC30001: Statement is not valid in a namespace" could mean you omitted the Public Module / End Module.
- If Excel prompts for Enabling Macros, and then the function does not appear to be available, you might not have the right filename for the .dna file. The name should be the same as the .xll file and it should be in the same directory. Or, you may have omitted to declare the module as Public.
- Otherwise, post on the discussion list http://groups.google.com/group/exceldna

You can call a UDF in an XLL from VBA by using Application.Run.

```
X = Application.Run("MyFunc", param1, param2)
```

I have not yet seen any significant overhead in doing this.
You could also use the Evaluate function or its square bracket equivalent, but these require the parameters to be passed as literals. For example:

```
Function MyFunc2(s As String, d As Double) As Double
```

You could call it from VBA as

```
X = Application.Run("MyFunc2", "test", 3)
X = [MyFunc2("test",3)]
X = Evaluate("MyFunc2(""test"",3)")
```

This gives a simple way to migrate a UDF from VBA to Excel-Dna for a quick and easy performance improvement. How much of an improvement depends on how efficient the code is. Avoid many thousands of calls to Excel worksheet functions, such as ExcelDnautil.Application.WorksheetFunction.MIN(). Look for a native VB.Net equivalent, or it may be faster to rewrite some functions in inline logic. Write timing tests to verify whether there is any speed improvement.

Stage 2: Compiling a .DLL

2.1 Compiling without an IDE

You can skip to the IDE example, as I'll only show once this bit of a throwback to the old command line days, but it does reduce things to essentials. Create a text file 'TestDll.vb' containing this code:

```
' Simple test of ExcelDna
Public Module MyFunctions
	Function AddThem(x, y)
		AddThem = x + y
	End Function
End Module
```

Change directory to where the Visual Basic compiler is, and use the vbc.exe compiler to compile the .vb file to a DLL (Dynamic Link Library) file to be included in the ExcelDna project.

```
CD C:\Windows\Microsoft.NET\Framework\v4.0.30319
vbc F:\DOCS\SCC3\ExcelDna\TestDll\TestDll.vb /target:library
```

This creates a file `F:\DOCS\SCC3\ExcelDna\TestDll\TestDll.dll`

You can now refer to this external library from a .Dna file as follows:

```
<DnaLibrary Language="VB" Name="MyFunctions" RuntimeVersion="v4.0">
<ExternalLibrary Path="TestDll.dll" />
</DnaLibrary>
```

Copy the file ExcelDna.xll from the ExcelDna distribution folder to this project's folder and rename it TestDll.xll. You should now have four files:

TestDll.vb
TestDll.dll
TestDll.dna
TestDll.xll

Now, double-click on TestDll.xll to load it. TestDll.xll will on loading read the TestDll.dna file and load the code from the TestDll.dll specified in the .dna file. Enable macros, and test the function.

Troubleshooting

Error: Could not load file or assembly '…TestDll.dll' or one of its dependencies. This assembly is built by a runtime newer than the currently loaded runtime and cannot be loaded. `RuntimeVersion="v4.0"` was not specified so by default .Net runtime version 2 was loaded.

Do not put spaces around the equals signs in the <DnaLibrary…> line.

If you enclosed the function in a Public Class rather than a Public Module, and the function is not visible in Excel, add the keyword Shared to its declaration.

2.2 Using an IDE

If you have the budget, go for the Pro editions of the Visual Studio IDE with all the professional plugins. I shall use Microsoft Visual Basic 2010 Express which is free. An alternative is SharpDevelop, also free, which offers integration with third party plugins such as refactoring tools.
http://www.icsharpcode.net/OpenSource/SD/Default.aspx

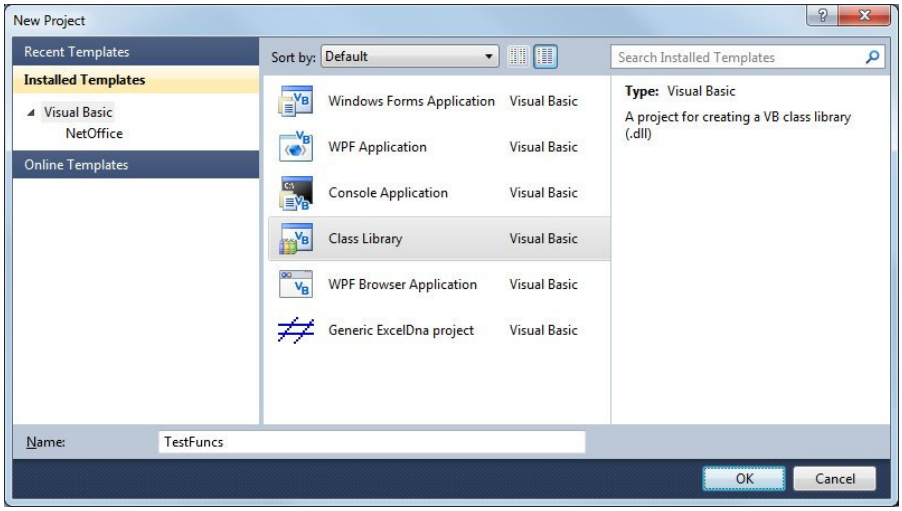
Ross McLean uses the SharpDevelop IDE in this example
http://www.blog.methodsinexcel.co.uk/2010/09/22/writing-an-Excel-Dna-function-using-an-ide/

Download and install Visual Basic 2010 Express free from
http://www.microsoft.com/visualstudio/en-us/products/2010-editions/express
I recommend you read the Getting Started projects.
By the time you read this, Visual Studio 2012 may be available.

Start MS VS 2010 Visual Basic. You are going to create a new project. When I have re-used existing directories for VB.Net projects I ended up with a mess of duplicate files, so I'll start from scratch and copy in content as required.

In Tools > Options > Projects and Solutions > check “Show Output window when Build starts”

New Project > Class library, enter TestFuncs in the Name box.



In the Solution Explorer pane, right-click Class1.vb and delete it.
Project > Add New Item, Module, name it TestFuncs.vb.

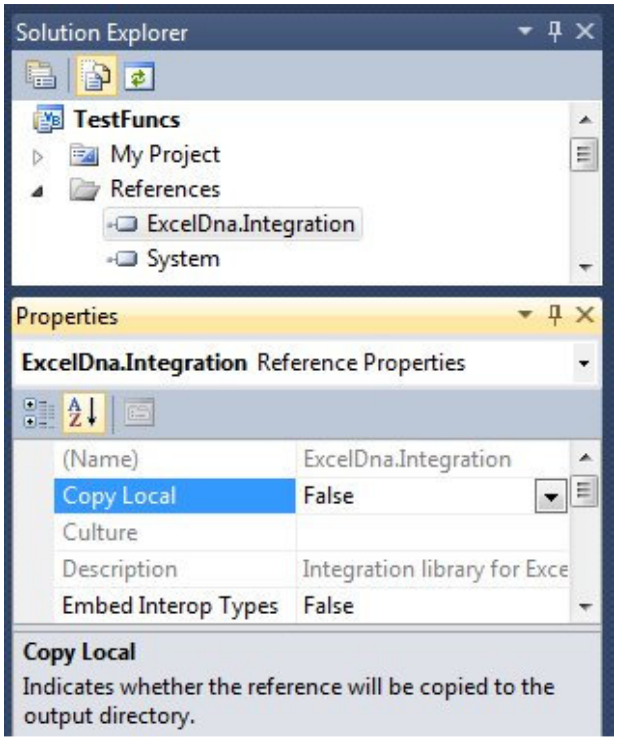
Enter the following code. It is very similar to what you would have in VBA except that in VBA the function declaration would be

```
Public Function SumNValues(Values As Variant) As Double
```

```
'Sum all numeric values in a 2-D array, excluding numbers formatted as dates
Public Module MyFunctions
    Public Function SumNValues(ByVal Values(,) As Object) As Double
        Dim value As Object
        SumNValues = 0
        For Each value In Values
            Select Case VarType(value)
                Case vbDouble, vbCurrency, vbDecimal  ' exclude vbDate
                    SumNValues = SumNValues + value
            End Select
        Next
    End Function
End Module
```

Next, you want the compiler to have a reference to the ExcelDna Integration library so it can resolve references to it. But you don't want to include the library as a file in the project, because it is also embedded in the file ExcelDna.xll which you will include as a file with the name changed to the project name.

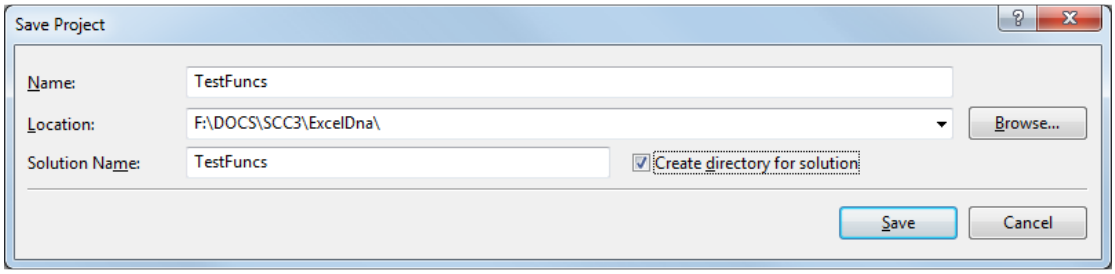
Project > Add Reference > Browse tab, to ExcelDna.Integration.dll eg …ExcelDna-0.30\Distribution\ExcelDna.Integration.dll
In subsequent projects, you can use Recent tab on the Add Reference dialog to revisit that location.
Project > Show All Files and expand the References branch,
Select ExcelDna.Integration, make property *Copy Local=False*



Project > Add New Item > Text File > and name it TestFuncs.Dna
Set in its File Properties the property *Copy to Output Directory* to *Copy if newer*.
Enter the following content:

```
<DnaLibrary Language="VB" RuntimeVersion="v4.0">
<ExternalLibrary Path="TestFuncs.dll" />
</DnaLibrary>
```

File > Save All, leave the name as TestFuncs, create a directory for the solution. You can leave the output directory as the default (…documents\visual studio 2010\Projects) or change it to the same path as the previous example F:\DOCS\SCC3\ExcelDna\. If the folder name already exists, it will simply create another directory with a 1 added to the name, eg TestFuncs1. That could be a source of confusion.



The folder has two files with extensions .sln, .suo, a subfolder *TestFuncs* with .vbproj and .vbproj.user, the .dna and .vb files you have just created, and subfolders *bin*, *My Project*, and *obj*. The *bin* folder has subfolders *Debug* and *Release*. You don't need to copy any files into Debug or Release, the *Copy file* properties you set above in the IDE will determine that.

Outside the IDE, copy the file Excel-Dna-0.29\Distribution\ExcelDna.xll to the subfolder containing TestFuncs.dna and rename it TestFuncs.xll. To add it to the project do:
Project > Add Existing Item > TestFuncs.xll and set in its File Properties the property *Copy to Output Directory* to *Copy if newer*.

All you need to do now is click Build and check the Output window for any errors:

```
------ Build started: Project: TestFuncs, Configuration: Debug Any CPU ------
  TestFuncs -> F:\DOCS\SCC3\ExcelDna\TestFuncs\TestFuncs\bin\Debug\TestFuncs.dll
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

Finally, double-click TestFuncs.xll in the output folder (Debug or Release) and test the function in Excel. Here is some test data:

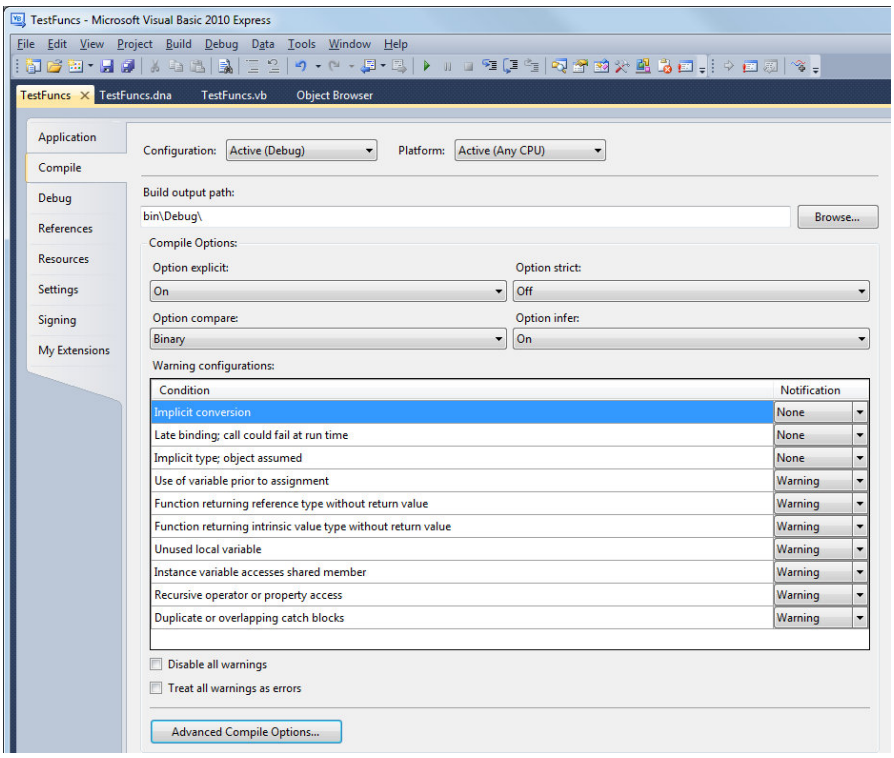
A1: '1	apostrophe prefix
A2: 1	number
A3: =1	formula evaluating to 1
A4: 100%	1 formatted as percent
A5: $1.00	1 formatted as currency (your locale determines the symbol)
A6: 01/01/1900	1 formatted as date

`=SumNValues(A1:A6)` should return 4 the same as the VBA version (ie it should exclude A1 and A6), but it returns 5. Why? Let's do some debugging to find out.

Compiler Options

Visual Studio automatically chooses the Debug configuration when you choose Start from the Debug menu and the Release configurations when you use the Build menu. You can tailor this from the menu Build > Configuration Manager. See http://msdn.microsoft.com/en-us/library/wx0123s5.aspx

Project > TestFuncs Properties > Compiler shows the options in effect. If you want a really nit-picking approach to possible problems, choose Option Strict On in the dialog:



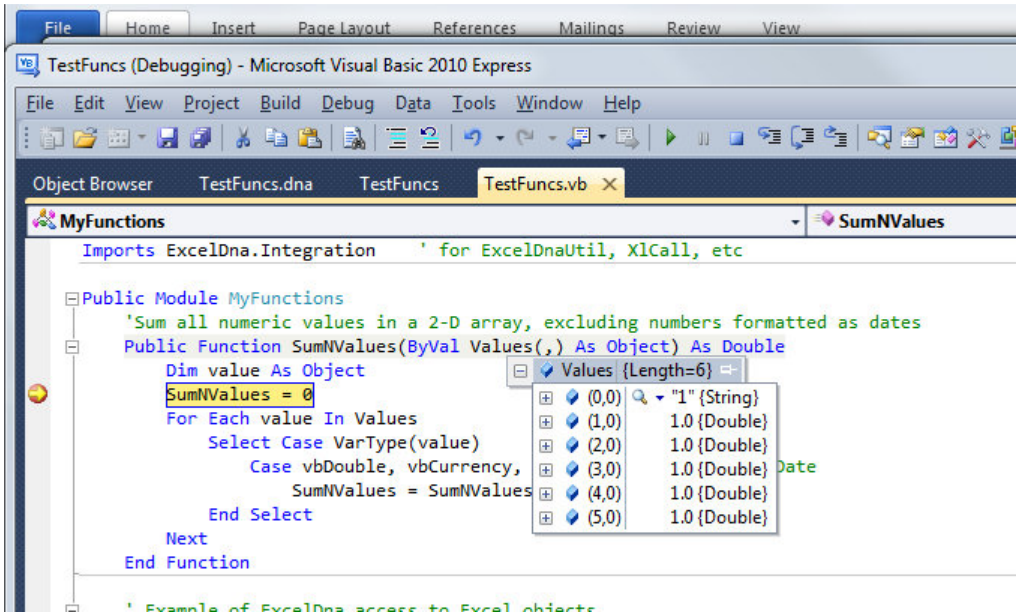
2.3 Debugging

Debugging is easy in VBA, it takes a little more effort in VB.Net but still doable.

VB Express does not show the option in its user interface (UI) to debug an addin executing inside Excel but you can do it yourself by editing the MyApp.vbproj.user file as follows, substituting MyApp with the XLL name.

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' ==
'Debug|AnyCPU' ">
    <StartAction>Program</StartAction>
    <StartProgram>C:\Program Files\Microsoft
Office\Office14\EXCEL.EXE</StartProgram>
    <StartArguments>MyApp.xll</StartArguments>
  </PropertyGroup>
  <PropertyGroup>
    <ProjectView>ShowAllFiles</ProjectView>
  </PropertyGroup>
</Project>
```

Close the project and edit TestFuncs.vbproj.user as above, changing MyApp to TestFuncs. Then re-open the project and choose Debug or press F5. Excel will launch, you enable macros as normal. In the IDE, set a breakpoint on line 7:
`SumNValues = 0`
F9 sets a breakpoint same as VBA. Enter the function in a cell
`=SumNValues(A1:A6)`
The code stops with the familiar yellow line in the IDE, you can inspect the variables, use the immediate window, and do pretty well anything you can do in VBA except edit the code on the run.
Hover the mouse over the Values(,) parameter in line 5 and expand the tree to see what its items are:



As you can see the last item, from A6, is passed as a Double rather than as a Date as is the case in VBA. This is another catch to be aware when passing values to VB.Net.

To know what the original cell type is, we need to pass a reference to the range A1:A6 to the function, not the values of the range. It's time to learn how to receive Excel references.

2.4 Getting to the Excel object model

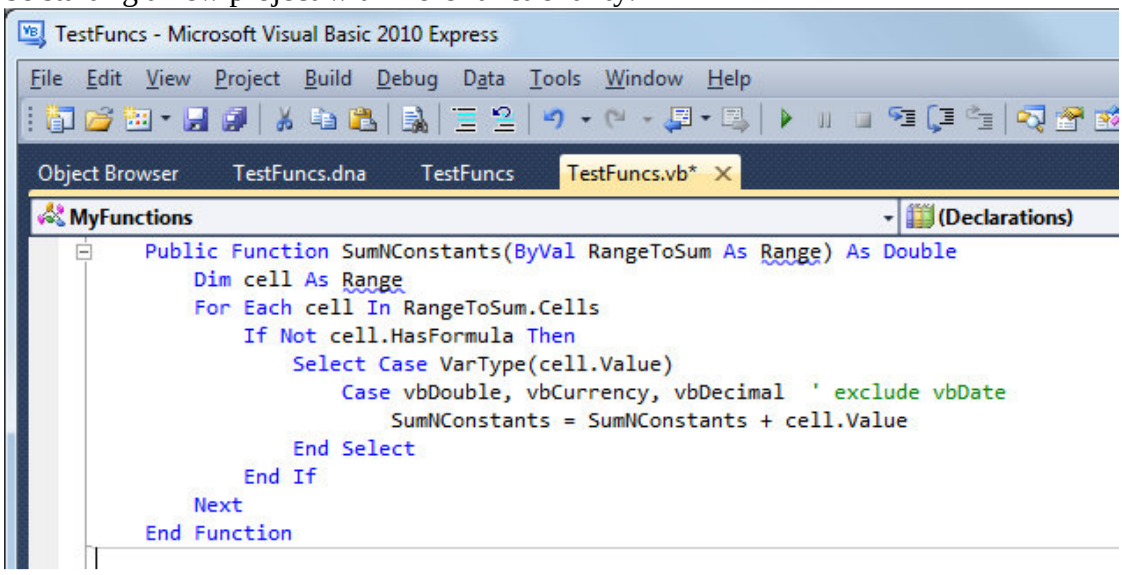
The alternatives for interacting with Excel are explained by Govert van Drimmelen in a post to http://groups.google.com/group/Excel-Dna reproduced in the background reading later.

Example: Excel object model via COM, late bound

Here is a VBA function that returns the sum of numeric values in a range excluding formula values and date constants. It uses properties such as .HasFormula and .Value.

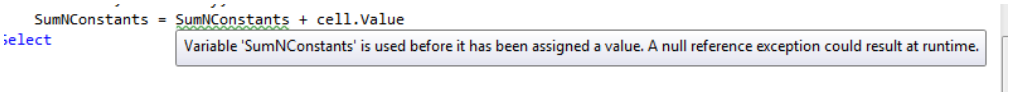
```
Public Function SumNConstants(RangeToSum As Range) As Double
    Dim cell As Range
    For Each cell In RangeToSum.Cells
        If Not cell.HasFormula Then
            Select Case VarType(cell.Value)
                Case vbDouble, vbCurrency, vbDecimal  ' exclude vbDate
                    SumNConstants = SumNConstants + cell.Value
            End Select
        End If
    Next
End Function
```

For simplicity, let's add this function to the TestFuncs you already have. Later, you'll be starting a new project with more functionality.

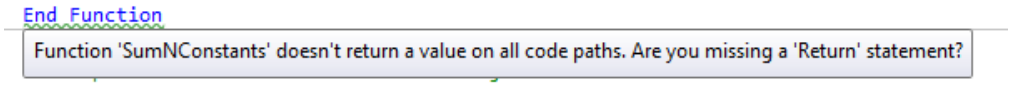


The type Range is flagged with green squiggles because it belongs to System.Data.Range which is a completely different VB.Net class which is not referred to in this project. Without using any Excel type library you need to use the variant Object type for *Range* and *cell*.

When you fix that, you see a warning on the SumNConstants calculation:



And on the End Function line



These can be cleaned up by initialising SumNConstants to zero at the start of the function.

However, we are not done yet. The line

```
        For Each cell In RangeToSum.Cells
```

would throw a runtime error because RangeToSum is an Object in VB.Net with no connection to Excel. You would see #VALUE! in the cell and see in the IDE Output Window:

```
A first chance exception of type 'System.MissingMemberException' occurred in
Microsoft.VisualBasic.dll
```

There are several things to add:

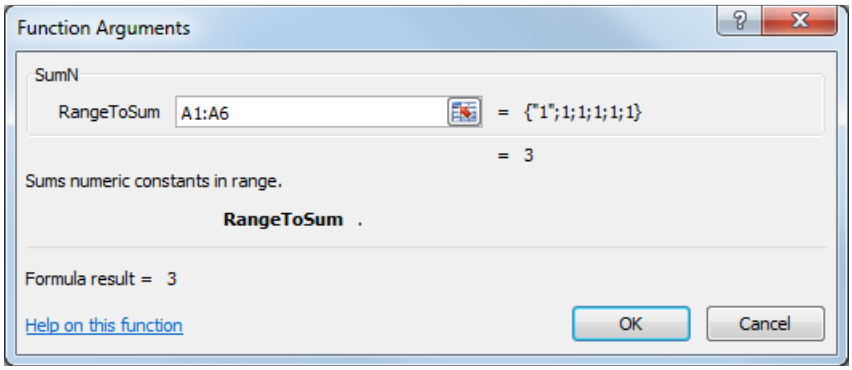
1) At the top of TestFuncs.vb add a line `Imports ExcelDna.Integration` that allows us to shorten references to this frequently used qualifier like
   `Dim App As Object = ExcelDna.Integration.ExcelDnaUtil.Application`
   To
   `Dim App As Object = ExcelDnaUtil.Application`
2) The `<ExcelFunction(...)>` attribute in the completed code below, and the `IsMacroType:=True` flag, are used to be able to handle ExcelReferences.
3) The `AllowReference:=True` option only affects parameters of type Object. It changes the registration type of the function to tell Excel to pass an `ExcelDna.Integration.ExcelReference` if the function is called with a range reference. Otherwise the function would always get the values of the range, not the cell references.
4) The `ReferenceToRange` helper function converts an Excel Reference received from the UDF; first to an address using the C API class `XlCall`, and then to a Range using the Range method of the ExcelDna.Integration. ExcelDnaUtil.Application object.

In VBA you can define a function to receive a Range object and work with that directly. That is not possible with VB.Net. The closest we can get is to use AllowReference to tell Excel to pass a reference but even that is not a Range object; it is a C structure and we have to convert that to a Range by getting its address (eg `"[TestFuncs.xlsx]Sheet1!$A$1:$A$6"`) and then passing that to the Excel Application.Range method to return a Range object which the function then returns as a generic Object.

If you define a function but cannot see it from Excel as a UDF, first check that it is marked `IsMacroType:=True` and also check that the parameters are one of Object, Double, or String. A side effect to be aware of is that if the function that is registered as `IsMacroType:=True` has an argument marked `AllowReference:=True`, Excel will treat the function as volatile even though you'd expect it not to be.

```
<ExcelFunction(Category:="Test Functions", Description:="Sums numeric
constants in range", IsMacroType:=True)>
```

The Category and Description are used in Excel's Function Wizard:



All the above looks rather more awkward compared to the VBA version. When you dig in to it, you recognise that the C API calls being used here are based on the original XLM macro language still used within defined Names for certain legacy purposes. For example, GET.CELL is represented by xlfGetCell. These macro-sheet functions and commands are documented in the Help file XLMacr8.hlp (formerly named Macrofun.hlp). That should be available from http://support.microsoft.com/kb/143466

You can learn more by downloading the Microsoft Excel 2010 XLL Software Development Kit from http://msdn.microsoft.com/en-us/library/office/bb687883(v=office.14).aspx

The complete code of the function is now:

```
Imports ExcelDna.Integration    ' for ExcelDnaUtil, XlCall, etc

Public Module MyFunctions
    'Sum all numeric values in a 2-D array, exclude numbers formatted as
dates
    Public Function SumNValues(ByVal Values(,) As Object) As Double
        Dim value As Object
        SumNValues = 0
        For Each value In Values
            Select Case VarType(value)
                Case vbDouble, vbCurrency, vbDecimal  ' exclude vbDate
                    SumNValues = SumNValues + value
            End Select
        Next
    End Function

    <ExcelFunction(Category:="Test Functions", Description:="Sums numeric
constants in range", IsMacroType:=True)> _
    Public Function SumNConstants(<ExcelArgument(AllowReference:=True)>
ByVal RangeToSum As Object) As Double
        Dim cell As Object, rgToSum As Object
        SumNConstants = 0
        rgToSum = ReferenceToRange(RangeToSum)
        For Each cell In rgToSum.Cells
            If Not cell.HasFormula Then
                Select Case VarType(cell.value)
```

```
                    Case vbDouble, vbCurrency, vbDecimal  ' exclude vbDate
                        SumNConstants = SumNConstants + cell.value
                End Select
            End If
        Next
    End Function

    Private Function ReferenceToRange(ByVal xlRef As ExcelReference) As
Object
        Dim strAddress As String = XlCall.Excel(XlCall.xlfReftext, xlRef,
True)
        ReferenceToRange = ExcelDnaUtil.Application.Range(strAddress)
    End Function
End Module
```

You can now press F5 to launch and test this solution. `=SumNConstants(A1:A6)` returns 3 because it sums A3,A4,A5, excluding A1 (text), A2 (formula), A6 (date).

By the way, we can enhance the `ReferenceToRange` function to handle ranges with several areas. A single `XlCall.xlfReftext` is limited to 255 characters, so the following concatenates the addresses from the individual areas of the reference:

```
 Private Function ReferenceToRange(ByVal xlRef As ExcelReference) As Object
   Dim cntRef As Long, strText As String, strAddress As String
   strAddress = XlCall.Excel(XlCall.xlfReftext, xlRef.InnerReferences(0),
True)
   For cntRef = 1 To xlRef.InnerReferences.Count - 1
     strText = XlCall.Excel(XlCall.xlfReftext,
xlRef.InnerReferences(cntRef), True)
     strAddress = strAddress & "," & Mid(strText, strText.LastIndexOf("!") +
2)
' +2 because IndexOf starts at 0
   Next
   ReferenceToRange = ExcelDnaUtil.Application.Range(strAddress)
 End Function
```

The drawback of late binding is that there is no Intellisense and no checking for mistakes at compile time. So you could have a misspelling like this that would only fail at runtime:

```
    Dim myRange
    myRange = Application.ActiveSheet.Ragne("A1")
```

We'll show how to add Intellisense in the next section.

Before we leave this example, let's look at another consequence of not having the Excel types available in DotNet.

In VBA we are used to types like Range, Worksheet, Workbook, and global objects like ActiveCell, ActiveSheet, ActiveChart, ActiveWindow, ActiveWorkbook, and the Workbooks collection; and global constants like xlSheetVisible. These are not provided in VB.Net, so we have to use As Object for the types, qualify all the global objects back to the Application object, and define our own constants.

To illustrate some of these concepts, let's add a Sub to the TestFuncs project to do some operations on an Excel sheet.

```
    Public Sub TestXL() ' test access to the Excel Object Model
```

```
        Dim ws As Object ' not As Worksheet
        ws = ExcelDnaUtil.Application.activesheet
        MsgBox(ws.name)
    End Sub
```

You can run this using Alt+F8 and typing in the macro name TestXL.
Assuming Excel starts with the default Book1, it should show a message "Sheet1".

To add it as a menu button for simpler testing, use:

```
<ExcelCommand(MenuName:="Test&XL", MenuText:="Run Test&XL")>
```

This is the old Excel 2003-style menu that appears in the Addins tab of the Ribbon.
The shortcut key to that is X, and we have specified X as the menu accelerator and the button shortcut. So now you can run it just by typing Alt, X, X, X.
By default the name is the method name, but you can override with an attribute:

```
<ExcelCommand(Name:="MyMacro", MenuName:="Test&XL", MenuText:="Run
Test&XL")>
```

When you go to the Macros dialog box and type in "MyMacro", the Run button should become available and then run your macro if you click it.

To save having to type ExcelDnaUtil before Application, we can import that at the top. The code now reads

```
Imports ExcelDna.Integration                ' for ExcelDnaUtil, XlCall, etc
Imports ExcelDna.Integration.ExcelDnaUtil   ' for Application

Public Module MyFunctions

    ' we use ExcelCommand to attach this macro to a menu button in Excel
    <ExcelCommand(MenuName:="Test&XL", MenuText:="Run Test&XL")> _
    Public Sub TestXL() ' test access to the Excel Object Model
        Dim ws As Object ' not As Worksheet
        ws = Application.activesheet
        MsgBox(ws.name)
    End Sub
```

You can now extend that to try other globals such as :

```
        MsgBox(Application.activeworkbook.name & vbLf _
               & Application.activesheet.name & vbLf _
               & Application.activecell.address  _
               , vbOK, "Test XL")
```

But if you try to specify

```
Application.activecell.address(ReferenceStyle:=xlR1C1)
```

The compiler outputs:

```
Error  1 'xlR1C1' is not declared. It may be inaccessible due to its
protection level.
```

You would have to either specify the numeric value of 2 or add the line

```
    Const xlR1C1 As Integer = 2
```

If you are pasting in code from VBA you either edit references to globals like "Workbooks" to prefix them with "Application." or add helper properties to return a reference to them like this:

```
    ReadOnly Property Workbooks As Object
```

```
        Get
            Return Application.Workbooks
        End Get
    End Property
```

So the code now looks like:

```
Imports ExcelDna.Integration                ' for ExcelDnaUtil, XlCall, etc
Imports ExcelDna.Integration.ExcelDnaUtil   ' for Application

Public Module MyFunctions
    Const xlR1C1 As Integer = 2

    ' we use ExcelCommand to attach this macro to a menu button in Excel
    <ExcelCommand(MenuName:="Test&XL", MenuText:="Run Test&XL")> _
    Public Sub TestXL() ' test access to the Excel Object Model
        Dim ws As Object ' not As Worksheet
        ws = Application.activesheet
        MsgBox("Number of Workbooks open: " & Workbooks.count & vbLf _
           & Application.ActiveWorkbook.name & vbLf _
           & Application.activesheet.name & vbLf _
           & Application.activecell.address(ReferenceStyle:=xlR1C1)  _
               , vbOK, "Active info")
    End Sub

    ReadOnly Property Workbooks As Object
        Get
            Return Application.Workbooks
        End Get
    End Property
```

At this stage you might be wondering how much editing is involved to migrate from ExcelDna, especially with all the xl* constants. Actually, the constants are easily done; just include in your project a copy of the enums downloaded from Mike Alexander's site. You'll have to change the second xlColorIndex to say xlCI.
http://www.datapigtechnologies.com/downloads/Excel_Enumerations.txt

You can also use my contribution of the free addin QualifyVBACode. It takes the VBProject in a workbook and qualifies the xl constants with their enum types, and prefixes ones like ActiveSheet with Application. The search/replace strings are in a XML key/value config file that you can edit. Of course, please read the ReadMe.txt file first after unzipping this (XLL, DNA, DLL, CONFIG) :

http://www.sysmod.com/QualifyVBAcode.zip

Because all the variables have to be Object, the loss of Intellisense is a risk when typing new code rather than simply copying working VBA code. So, let's tackle that problem next.

Boilerplate steps for creating a project in VB Express

The following projects start in the same way, so I'll give the generic steps here using the term "MyProject" which you can substitute with your own solution name. In the subsequent projects, I refer to these steps as "Create the standard ExcelDna project with the name …."

New Project > Class library, enter MyProject in the Name box.
Project > Add Reference > Recent tab, to ExcelDna.Integration.dll
Project > Show All Files and expand the References branch,
Select ExcelDna.Integration, make property *Copy Local=False*
Project > Add New Item > Text File > and name it MyProject.dna
Set in its File Properties the property *Copy to Output Directory* to *Copy if newer*.
Enter the following content:

```
<DnaLibrary Language="VB" RuntimeVersion="v4.0">
<ExternalLibrary Path="MyProject.dll" />
</DnaLibrary>
```

In the Solution Explorer pane, right-click Class1.vb and delete it. Or, you can simply rename this later.

File > Save All, leave the name as MyProject, create a directory for the solution.
Close the project and outside the IDE edit MyProject.vbproj.user to add the Debug functionality:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' ==
'Debug|AnyCPU' ">
    <StartAction>Program</StartAction>
    <StartProgram>C:\Program Files\Microsoft
Office\Office14\EXCEL.EXE</StartProgram>
    <StartArguments>MyProject.xll</StartArguments>
  </PropertyGroup>
  <PropertyGroup>
    <ProjectView>ShowAllFiles</ProjectView>
  </PropertyGroup>
</Project>
```

Outside the IDE, copy the file Excel-Dna-0.29\Distribution\ExcelDna.xll to the subfolder containing MyProject.dna and rename it MyProject.xll. To add it to the project, re-open the project and do:

Project > Add Existing Item, Show All Files (*.*) > MyProject.xll and set in its File Properties the property *Copy to Output Directory* to *Copy if newer*.
You could also do File > Export Template > project template > type a description and click Finish. This creates a file Myproject.zip in the folder Documents\Visual Studio 2010\My Exported Templates. You can then see this template when creating projects in the future.

Stage 3: Adding Intellisense to a EXCELDNA solution

As explained in the background reading at the end of this, there are two methods: the Microsoft Primary Interop Assembly (PIA) or a third party library like NetOffice.

The simplest use of PIA is described at Ross Mclean's blog
http://www.blog.methodsinexcel.co.uk/2010/10/28/ading-intellisense-to-a-dna-solution/
He provides all the PIAs in this zip:
http://www.blog.methodsinexcel.co.uk/wp-content/uploads/PIA-Excel.zip
Project > Add reference > .Net assembly Browser > find the PIA folder you just extracted, and pick the right version for your needs. Then in your module use
`Imports Microsoft.Office.Interop.Excel`
He says *"Otherwise, the drawback with the MS PIA is that it has to be installed on the client machine, and there is a different version for each version of Office. This also means you need admin rights to deploy your add-in."* However, that is overcome in .Net version 4.
http://blogs.msdn.com/b/vsto/archive/2008/05/20/common-pitfalls- during-pia-deployment-and-installation.aspx

Example of using the Excel Primary Interop Assembly (PIA)

I shall target Excel 2007 in this example and then test it in 2003 (for historical interest) and 2010.

Create the standard ExcelDna project with the name TestPIA.
For debugging, specify Excel 2007 in TestPIA.vbproj.user :

```
<StartProgram>C:\Program Files\Microsoft
Office\Office12\EXCEL.EXE</StartProgram>
```

Project > Add Reference > navigate to the Excel 2007 version of the PIA, in this case Microsoft.Office.Interop.Excel.12.dll
In the properties, check "Embed Interop Types" is True.[1]
We can now use Excel types in Dim statements such as Range, Workbook, etc.

Project > Add New Item, Module, name it TestPIA.vb.

Copy in the following code, similar to that in the TestFuncs example.

Because 'Application' is ambiguous between ExcelDna.Integration.ExcelDnaUtil and Microsoft.Office.Interop.Excel, create a global helper to point to the current Excel instance via ExcelDna. If you're preparing an addin for migration, changing

[1] As of Visual Studio 2010, C#/VB projects that target .NET 4 can now use the embedded interop types feature to get around the limitation of having to deploy the PIA on the target PC. When the Embed Interop Types property of a PIA reference is True, then any PIA types used by the application are embedded in the built assembly, so that the PIA is not needed at run time. In this scenario, an application that references an Office 2010 PIA can actually run on a computer with an earlier version of Office installed, provided that it does not use any APIs that are new in Office 2010. For a related walkthrough, see http://msdn.microsoft.com/en-us/library/ee317478.aspx. This will only work under .NET 4, and won't work with the NetOffice assemblies since they are not 'Primary' interop assemblies.

"ActiveWorkbook" to "Application.ActiveWorkbook" etc can be done in VBA and it works just as well.

The ExcelApi.Enums need their prefix so you have to use `XlReferenceStyle.xlR1C1`; or create your own enums module from the text file on the datapigtechnologies web site as described before.
However although Intellisense is fine for suggesting methods and properties as you type, it does not detect invalid methods pasted in from other code.
ws.UsedRagne will not get flagged as an error until runtime.

```
Imports ExcelDna.Integration            ' for ExcelDnaUtil, ExcelCommand,
XlCall, etc
Imports Microsoft.Office.Interop.Excel  ' Interface types from PIA eg
Workbook, Range

'instead of Imports ExcelDna.Integration.ExcelDnaUtil create this global
helper
' because 'Application' is ambiguous between ExcelDnaUtil and Interop.Excel
Module Globals
    ' connect the global Application to the Excel instance via ExcelDna
    Public ReadOnly Property Application As Application
        Get
            Application = ExcelDnaUtil.Application
        End Get
    End Property
End Module

Public Module TestPIA

    ' we use ExcelCommand to attach this macro to a menu button in Excel
    <ExcelCommand(MenuName:="Test &XL", MenuText:="Test &XL PIA")> _
    Public Sub TestPIA() ' test access to Excel via PIA
        Dim wb As Workbook, ws As Worksheet, rg As Range, cell As Range
        wb = Application.ActiveWorkbook
        ws = Application.ActiveSheet
        cell = Application.ActiveCell
        ' start with activecell A1 in Sheet1
        With cell.Offset(1, 1)  ' B2
            .Value = .Address(True, True,
ReferenceStyle:=XlReferenceStyle.xlR1C1)
            .Select()
        End With

        cell = Application.ActiveCell
        rg = Application.ActiveSheet.range("B3:C5")
        rg.Formula = "=Row()"

        MsgBox("Number of Workbooks open: " & Application.Workbooks.Count &
vbLf _
          & wb.Name & vbLf _
          & ws.Name & vbLf _
          & cell.Address(True, True,
ReferenceStyle:=XlReferenceStyle.xlR1C1)  _
             , vbOK, Application.Name & " " & Application.Version)

    End Sub

End Module
```

Test the project with VB Express with F5. Excel 2007 should launch and the menu should be visible. Quit Excel 2007.

Start Excel 2003, double-click …\TestPIA\TestPIA\bin\Debug\TestPIA.xll and the menu 'Test XL' should appear with the item 'Text XL PIA' and it runs.
Start Excel 2010, do the same, and it should also work.

Obviously, you can't access parts of the object model not present in the version of Excel running. If you need to handle this conditionally, then in the code test for the value of Application.Version.

I have Excel 2013 on another PC, so to test there I'll create a single XLL with the components so I have only one file to deploy.

Simplify deployment by packing the components into one .XLL file.

ExcelDnaPack is a command-line utility to pack Excel-Dna add-ins into a single .xll file.

Example: ExcelDnaPack.exe MyAddins\FirstAddin.dna

The packed add-in file will be created as MyAddins\firstaddin-packed.xll.

To pack additional assemblies, you add Pack="true" (note lowercase "true") to the references in the .dna file, eg

```
<ExternalLibrary Path="TestPIA.dll" Pack="true" />
<Reference Path="any other references" Pack="true" />
```

So in the command window it will appear as (warning is from second time output)

```
...\exceldna-0.29\distribution\exceldnapack bin\debug\testpia.dna
Output .xll file bin\debug\testpia-packed.xll already exists. Overwrite?
[Y/N] y

Using base add-in bin\debug\testpia.xll
  ->  Updating resource: Type: ASSEMBLY_LZMA, Name: EXCELDNA.INTEGRATION,
Length: 43546
  ~~> ExternalLibrary path TestPIA.dll resolved to bin\debug\TestPIA.dll.
  ->  Updating resource: Type: ASSEMBLY_LZMA, Name: TESTPIA, Length: 7078
  ->  Updating resource: Type: DNA, Name: __MAIN__, Length: 377
Completed Packing bin\debug\testpia-packed.xll.
```

To test, I copied testpia-packed.xll to the other PC, started Excel 2013, double clicked the xll, enable the addin, and the Alt-X,X,X test passes.

Example of using NetOffice

NetOffice is a version-independent set of Office Interop assemblies put together by Sebastian Lange.
http://netoffice.codeplex.com/

Unzip the distribution zip file into a convenient directory.

When you have time, read the files in the Examples and Tutorials directories.

Create the standard ExcelDna project with the name TestNetOffice.

Project > Add Reference > navigate to the correct version of Netoffice.dll eg
" NetOffice 1.5.1\NET 4.0\Assemblies\Any CPU\NetOffice.dll"
Set in its File Properties the property *Embed Interop types=False* and *Copy Local=True*. We can't use the embedding feature, intended for the MS PIAs, in NetOffice. Do the same two things for ExcelApi.dll

Depending on the features you use you may also need
OfficeApi.dll
VBIDEApi.dll

Project > Add New Item, Module, name it TestNetOffice.vb.

Copy in this code, similar to that in the TestFuncs example, but instead of the generic Object type for the variables, we'll use types from NetOffice.ExcelApi. For convenience and compatibility with VBA we shall name the ExcelAPI as Excel in the Imports clause. This should look familiar enough to a VBA developer. If you're preparing an addin for migration, changing "As Range" to "As Excel.Range" can be done in VBA and it works just as well.

There are some awkward syntax changes; for example, all .Address methods change to .get_Address. Be careful to change .Offset() and .Resize() to .get_Offset and .get_Resize; the first version will not be flagged as an error but has no effect, it returns the original range, so it's an easy mistake to make.

The extra line

```
New Excel.Application(Nothing, ExcelDnaUtil.Application)
```

is needed to initialise the NetOffice globals so we can now simply use ActiveWorkbook etc as in VBA.

You will also notice that the imported ExcelApi.Enums need their prefix so you have to use XlReferenceStyle.xlR1C1; or create your own enums module from the text file on the datapigtechnologies web site as described before.

Finally, I have added a Try…Catch block to illustrate the error handling in VB.Net. If you want to use Resume Next, that is only available with the On Error style of error handling.

```
Imports ExcelDna.Integration                  ' for ExcelDnaUtil, XlCall, etc
'Imports ExcelDna.Integration.ExcelDnaUtil  ' not needed for Application now
we have NetOffice

Imports Excel = NetOffice.ExcelApi            ' For Excel. types
Imports NetOffice.ExcelApi.GlobalHelperModules.GlobalModule ' for Workbooks
collection
Imports NetOffice.ExcelApi.Enums              ' for xlConstants

Public Module MyFunctions

  ' we use ExcelCommand to attach this macro to an Add-Ins menu button in
Excel
  <ExcelCommand(MenuName:="Test&XL", MenuText:="Run Test&XL")> _
  Public Sub TestXL() ' test access to the Excel Object Model
    Dim wb As Excel.Workbook, ws As Excel.Worksheet, rg As Excel.Range, cell
As Excel.Range
    Dim dummy As Excel.Application ' need this to initialise NetOffice
Globals
    Try
      dummy = New Excel.Application(Nothing, ExcelDnaUtil.Application)
      wb = ActiveWorkbook
      ws = ActiveSheet
      cell = ActiveCell
      ' start with activecell A1 in Sheet1
      With cell.get_Offset(1, 1)  ' B2
        .Value = .get_Address(True, True,
referenceStyle:=XlReferenceStyle.xlR1C1)
        .Select()
      End With

      cell = ActiveCell
      rg = ActiveSheet.range("B3:C5")
      rg.Formula = "=Row()"

      MsgBox("Number of Workbooks open: " & Workbooks.Count & vbLf _
        & wb.Name & vbLf _
        & ws.Name & vbLf _
        & cell.get_Address(True, True,
referenceStyle:=XlReferenceStyle.xlR1C1)  _
           , vbOK, Application.Name)

    Catch ex As Exception
      Dim Message As String = ex.Message
      If StrComp(Message, "See inner exception(s) for details.",
vbTextCompare) = 0 Then
        Message = ex.InnerException.Message
      End If
      Message = Message & vbLf & ex.StackTrace
      Debug.Print(Message)
      MsgBox(Message, MsgBoxStyle.Exclamation, ex.Source)
    End Try
  End Sub


End Module
```

NetOffice allows version independence with .Net version 2. As I shall be targeting version 4, I shall use the PIA and embed the Interop to obtain the same result.

Specific NetOffice notes

Range, Offset, Resize

(This may change with Netoffice libraries)
You can use Application.Range(strAddress) to get a range from any qualified address. Be aware that with the ExcelApi.Range type, you MUST change all occurrences of .Offset to .get_Offset and .Resize to .get_Resize.

```
Dim xCell As NetOffice.ExcelApi.Range, oCell As Object
ws.Range("B2:D4").Select() '  3 rows, 2 columns
xCell = Selection
oCell = Selection

'Object references work like in VBA
oCell: Range $B$2:$D$4
oCell.offset(1, 1): Range $C$3:$E$5 as expected
oCell.get_offset(1,1): Range $C$3:$E$5 as expected

'ExcelApi references are different
xCell: Range $B$2:$D$4
xCell.offset(1, 1): Range $B$2 ' cell in row 1, col 1 of B2:D4
xCell.get_offset(1,1): Range $C$3:$E$5 as expected

Explanation:
xCell.offset(1, 1) is evaluated like this in NetOffice
rg=xcell            ' B2:D4
rg=rg.Offset    ' B2:D4
rg=rg(1,1)        ' B2
```

Enumerations and .xl* Constants

With NetOffice, first import the NetOffice.ExcelApi.Enums module. Then prefix enumerated constants with their type, eg XlDirection.xlUp rather than simply xlUp. The prefix can be added in VBA as well which may avoid ambiguities like xlVisible and xlSheetVisible.

Change all occurrences of "As Range" in your VBA to "As Excel.Range" and this will work the same in both VBA and VB.Net

.Characters property gives an error

```
'Error     16    Class 'NetOffice.ExcelApi.Characters' cannot be indexed because
it has no default property.
.Characters(Start:=1, Length:=lPos).Font.ColorIndex = 38
```

This is a byproduct of the way the default properties are accessed in Netoffice. As of 1.5.1, there is no workaround yet.

To use the global Application object

1) Do this in a Public Module GlobalHelper
Property Application As Netoffice.ExcelApi.Application

2) Do this in a Public module

```
Public Module Globals
   ' connect the global Application to the Excel instance via ExcelDna
   ReadOnly Property Application As Application
```

```
      Get
         Application = ExcelDnaUtil.Application
      End Get
   End Property
End Module
```

I need Netoffice.ExcelApi. or I get
'error BC30561: 'Application' is ambiguous, imported from the namespaces or types 'System.Windows.Forms, NetOffice.ExcelApi'.

DrawingObjects

In VBA, drawing objects are a collection but in Netoffice they are a COMObject

```
' For Each obj In ws.DrawingObjects gives Error 155 Expression is of type
'LateBindingApi.Core.COMObject', which is not a collection type.
' instead use:
        For Each obj In CType(ws, Object).DrawingObjects
```

For Netoffice it is also necessary to change some properties to the Excel-Dna get_ names:
.Range → .get_Range
.Address → .get_Address
.Offset → .get_Offset
.Resize → .get_Resize
.End → .get_End
.Characters → .get_Characters
There may be more.

VBA accepts a variable number of parameters, eg for .Offset you can specify only a row offset and the column offset defaults to 0. They must be specified in Excel-Dna so after the search/replace for the .get_ names, some parameters may have to be completed. The only one I found irritating was .get_Address which needs five parameters so I decided to create a function Range_Address which handles the optional parameters and will also convert 0/1 to False/True as required when that kind of lazy shortcut was taken when writing the VBA code.

To do a search and replace of <expression>.Address(…
Visual Studio regular expressions: Find and Replace
'{:a+}\.address
'range_address(\1)

Notepad++ regular expressions: Search for any text followed by a space or bracket then any text followed by .address(and replace it by the first two subexpressions then Range_Address(then the text just before .address, which should be the range object reference:

```
(.*)([ \(])(.*?)(\.address\()
```

Replace with:

```
\1\2 Range_Address(\3,
```

Names

To index into the wb.Names collection, use .get_Name(index)

Range.Sort

.Sort via the PIA needs at least the first key specified; and orientation if you want it sorted top to bottom, eg

```
rg.currentregion.sort( Key1:=rg,
Orientation:=xlSortOrientation.xlSortColumns )
```

The .Sort method in NetOffice was a headache because every possible parameter must be specified, unlike VBA where they can be omitted. Isolate this into its own sub:

```
ws.Cells(1, 2 ).CurrentRegion.Sort(header:=XlYesNoGuess.xlYes, _
   key1:=ws.Cells(1, 3 ), order1:=XlSortOrder.xlDescending, _
   key2:=ws.Cells(1, 2 ), order2:=XlSortOrder.xlDescending, _
   key3:=ws.Cells(1, 1 ), order3:=XlSortOrder.xlAscending, _
   dataOption1:=XlSortDataOption.xlSortNormal, _
   dataOption2:=XlSortDataOption.xlSortNormal, _
   dataOption3:=XlSortDataOption.xlSortNormal, _
   matchCase:=False, orderCustom:=Nothing, _
   orientation:=XlSortOrientation.xlSortColumns, _
   sortMethod:=XlSortMethod.xlPinYin, type:=Nothing)
```

Workbooks.Open()

This has seventeen parameters. If you want to specify one near the end of the list, like AddToMRU, in VBA you can simply use named parameters, in VB.Net you must specify them all up to that point. You cannot omit them by using two commas in succession. You can pass Nothing for all values except 'format' and 'origin' which must have some real value, eg

```
Workbooks.Open(filename:=strFilename, updateLinks:=False, readOnly:=False, _
                        format:=5, password:=Nothing, writeResPassword:=Nothing, _
                  ignoreReadOnlyRecommended:=Nothing, origin:=2, addToMru:=False, _
                  converter:=Nothing, corruptLoad:=Nothing, delimiter:=Nothing, _
                  editable:=False, local:=Nothing, notify:=Nothing)
'http://msdn.microsoft.com/en-
us/library/microsoft.office.interop.excel.workbooks.open(v=office.11).aspx
        'format:=5 any value from 1-6 will do if it's not a text file
        'origin:=Enums.XlPlatform.xlWindows=2
```

You cannot specify Format:=Nothing or origin:=Nothing, get: System.Runtime.InteropServices.COMException (0x800A03EC): Unable to get the Open property of the Workbooks class

Application.GetOpenFilename()

The FileFilter argument consists of pairs of file filter strings followed by the MS-DOS wildcard file filter specification, eg "All Files (*.*),*.*". If you omit the comma, Excel silently errors and the ExcelDna app hangs.

Stage 4: Creating an add-in with Ribbon commands

The above examples show an old Excel 2003 style menu button. It's time to create an addin with a Ribbon button that does more work with Excel.

We shall create an add-in to list the names of sheets to a worksheet named $TOC Create the standard ExcelDna project with the name WorkbookTOC.

Project > Add Reference > navigate to the Excel 2010 version of the PIA, in this case Microsoft.Office.Interop.Excel.14.dll

Project > Add New Item, Module, name it WorkbookTOC.vb.

Enter the following test code; we will replace it with real code later.

```
Public Module WorkbookTOC
    Sub CreateTableOfContents()
        MsgBox("CreateTableOfContents")
    End Sub
End Module
```

Edit the standard .dna text file you created named WorkbookTOC.Dna and copy in the text below. The ExternalLibrary WorkbookTOC.dll will be created when you build the project. The CustomUI contains the Ribbon markup which adds a group to the Excel Ribbon with one button with a standard MSO image and an onAction that calls a standard ExcelDna method that in turn calls (with no parameter) the macro named in the tag.

```
<DnaLibrary Language="VB" Name="Table of Contents Add-in"
RuntimeVersion="v4.0">
  <ExternalLibrary Path="WorkbookTOC.dll" />
  <CustomUI>
  <!--Note the <CustomUI> with a capital 'C' tag that encloses the
<customUI>
      with a small 'c' tag that saved in the .xll. -->
  <customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
  <ribbon startFromScratch="false">
  <tabs>
    <tab idMso="TabReview">
     <group id="WorkbookTOC" label="TOC" insertAfterMso="GroupEditingExcel">
      <button id="CreateTableOfContents" tag="CreateTableOfContents"
        onAction="RunTagMacro" label="&amp;amp;Table of Contents"
        screentip="Insert Table of Contents worksheet" size="large"
        imageMso="TableOfFiguresInsert" />
     </group>
    </tab>
  </tabs>
  </ribbon>
  </customUI>
  </CustomUI>
</DnaLibrary>
```

To support the Excel 2007/2010 ribbon, add a Class module Ribbon.vb with a Public class for the Ribbon handler.

The ExcelRibbon-derived class must also be marked as ComVisible(True), or in the project properties, advanced options, the ComVisible option must be checked. This is not the 'Register for COM Interop' option, which must never be used with ExcelDna.

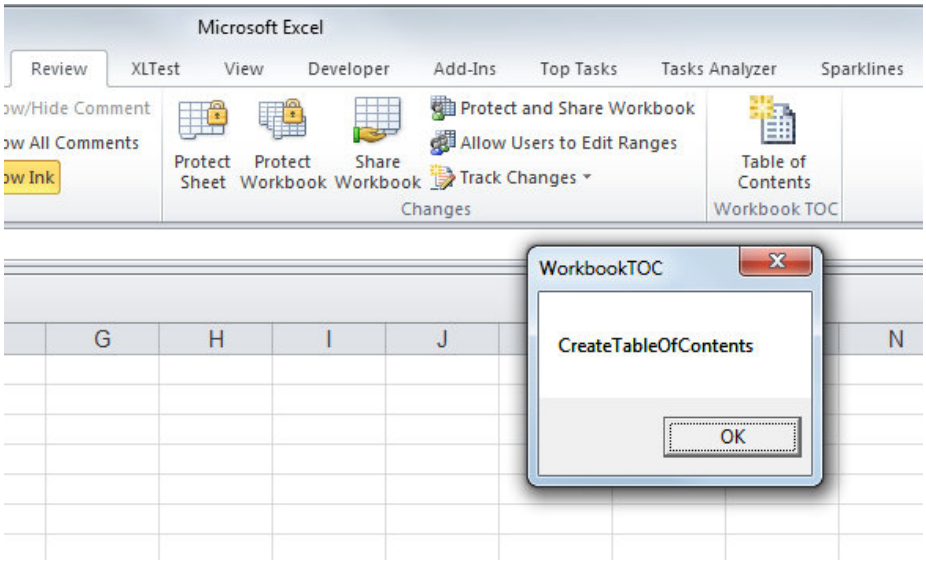
```
Imports ExcelDna.Integration
Imports System.Runtime.InteropServices      ' for <ComVisible(True)>

<ComVisible(True)> _
Public Class Ribbon    ' must be Public
    Inherits CustomUI.ExcelRibbon
End Class
```

If you use any other onAction procedure than "RunTagMacro", put it inside the Ribbon class because only subs in this class are visible to the Ribbon onAction. It is also the only way to pass the calling Ribbon control to the sub, for example:

```
    <ComVisible(True)> _
    Public Class Ribbon
        Inherits ExcelRibbon
        Sub RunControlID(ByVal ctl As IRibbonControl)
            ExcelDnaUtil.Application.Run(ctl.Id)
        End Sub
        Sub RunControlIDWithTag(ByVal ctl As IRibbonControl)
            ExcelDnaUtil.Application.Run(ctl.Id, ctl.Tag)
        End Sub
    End Class
```

Test the project with F5. After enabling the macros, you should see a button to the right of the Review group that displays the test message. If you don't, look at the Ribbon troubleshooting guide.



Having got the skeleton right, now we'll add the code to create the TOC.
It should look like this:

	A	B	C	D	E	F	G	H
1	**Sheet**	**Type**	**Sheet Tab Name**	**Visibility**	**Contents**	**Sum Total**	**Rows**	**Columns**
2	1	Worksheet	$TOC	Visible	(This sheet)			
3	2	Worksheet	Surveys	Visible	SurveyID	669920815.9	2	34
4	3	Worksheet	Questions	Visible	QuestionID	9376506357	15	17
5	4	Worksheet	Category_tables	Visible	Analysis	2543.16712	83	7
6	5	Worksheet	QuestionOptions	Visible	OptionID	866680	186	17
7	6	Worksheet	Respondents	Visible	RespondentID	9.32365E+20	1470	28
8	21	Chart		Visible	No Chart Title, 20 Series			
9	22	Worksheet	Tasks_PctCum	Visible	OptionText	2878123.549	80	80

```
Imports ExcelDna.Integration         ' for ExcelDnaUtil, ExcelCommand, XlCall, etc
Imports Microsoft.Office.Interop.Excel  ' Interface types from PIA eg Workbook, Range

'instead of Imports ExcelDna.Integration.ExcelDnaUtil create this global helper
' because 'Application' is ambiguous between ExcelDnaUtil and Interop.Excel
Public Module Globals
   ' connect the global Application to the Excel instance via ExcelDna
   ReadOnly Property Application As Application
      Get
         Application = ExcelDnaUtil.Application
      End Get
   End Property
End Module


Public Module WorkbookTOC
   Const WS_TOC_NAME = "$TOC"

   Sub CreateTableOfContents()
      Dim wsDoc As Worksheet = Nothing
      'Structured Error Handling (SEH):
      Try

         If Application.ActiveWorkbook Is Nothing Then
            Exit Sub
         End If

         If SheetExists(WS_TOC_NAME) Then
            Application.Worksheets(WS_TOC_NAME).delete()
         End If

         wsDoc = Application.Worksheets.Add()
         wsDoc.Name = WS_TOC_NAME
         CreateTableOfSheetsInfo(wsDoc)


      Catch ex As Exception
         Dim Message As String = ex.Message
         If StrComp(Message, "See inner exception(s) for details.", vbTextCompare) = 0 Then
            Message = ex.InnerException.Message
         End If
         Message = Message & vbLf & ex.StackTrace
         Debug.Print(Message)
         MsgBox(Message, MsgBoxStyle.Exclamation, ex.Source)
      End Try
   End Sub


   Sub CreateTableOfSheetsInfo(ByVal wsDoc As Worksheet)
      Dim wbCheck As Workbook
      Dim ws As Worksheet, lRow As Long
      Dim lSheet As Long, lCol As Long, rgFound As Range, sTemp As String
      wbCheck = wsDoc.Parent

      On Error GoTo OnError
      Application.EnableCancelKey = XlEnableCancelKey.xlErrorHandler
      Application.Calculation = XlCalculation.xlCalculationManual
      Application.Cursor = XlMousePointer.xlWait
      Application.ScreenUpdating = False

      ' Simplified version of columns, add CodeName or other stats if you like

      If SheetIsEmpty(wsDoc) Then
         lRow = 1
      Else
         lRow = wsDoc.Cells.SpecialCells(XlCellType.xlCellTypeLastCell).Row + 1
      End If
      lCol = 1

      ' using Array() UDF for VBA compatibility
      WriteHeadings(wsDoc.Cells(lRow, lCol), _
```

```
            Array("Sheet", "Type", "Sheet Tab Name", "Visibility", "Contents", "Sum Total",
"Rows", "Columns"))

      wsDoc.Cells(lRow, lCol).AddComment(CStr(Now()))

      For lSheet = 1 To wbCheck.Sheets.Count

         lRow = lRow + 1
         lCol = 0

         'Sheet#
         lCol = lCol + 1
         wsDoc.Cells(lRow, lCol).Value = lSheet

         'Type
         lCol = lCol + 1
         wsDoc.Cells(lRow, lCol).Value = TypeName(wbCheck.Sheets(lSheet))

         'Tab name with hyperlink to ws
         lCol = lCol + 1
         'ActiveSheet.Hyperlinks.Add Anchor:=ActiveCell, Address:= "F:\DOCS\TEST\ex1u.xls", _
         '   SubAddress:="'Budget 08'!C69", TextToDisplay:="C69"
         ' must specify all named parameters up to last one used, unlike VBA
         If TypeName(wbCheck.Sheets(lSheet)) = "Worksheet" Then
            wsDoc.Hyperlinks.Add(anchor:=wsDoc.Cells(lRow, lCol), _
                           address:=wbCheck.FullName, _
                           subAddress:=QuotedName(wbCheck.Sheets(lSheet).name) & "!A1", _
                           screenTip:=wbCheck.Sheets(lSheet).Name, _
                           textToDisplay:="'" & wbCheck.Sheets(lSheet).Name)
         End If

         'Visibility
         Select Case wbCheck.Sheets(lSheet).Visible
            Case XlSheetVisibility.xlSheetHidden    ' was .xlHidden
               sTemp = "Hidden"
            Case XlSheetVisibility.xlSheetVeryHidden ' was . xlVeryHidden
               sTemp = "Very Hidden"
            Case Else
               sTemp = "Visible"
         End Select
         lCol = lCol + 1
         wsDoc.Cells(lRow, lCol).Value = sTemp

         lCol = lCol + 1
         ' this section only for worksheets
         If TypeName(wbCheck.Sheets(lSheet)) = "Worksheet" Then
            ws = wbCheck.Sheets(lSheet) 'WS is type Worksheet
            If (ws Is wsDoc) Then ' skip THIS sheet being created
               wsDoc.Cells(lRow, lCol).Value = "(This sheet)"
            Else

               'Contents of first occupied cell
               If Not SheetIsEmpty(ws) Then   '  ws.UsedRange.Cells.Count > 0 Then
                  ' protect against empty sheet giving nonsense usedrange $U$1:$T$58
                  rgFound = ws.Cells(1, 1)
                  sTemp = CStr(rgFound.Value)
                  ' .text may show #### if narrow column and .value of date>2M may give
overflow err 6
                  If Len(sTemp) = 0 Then
                     ' don't use not IsEmpty(rgFound.Value) because single apostrophe
                     '   or ="" return false, we want some text
                     rgFound = FindFirst(ws.UsedRange, "*", XlFindLookIn.xlFormulas,
XlLookAt.xlPart)
                     ' find anything starting top left used range
                     If Not rgFound Is Nothing Then
                        sTemp = CStr(rgFound.Value)
                     End If
                  End If
                  wsDoc.Cells(lRow, lCol).Value = "'" & sTemp
               End If

               'Sum Total
               lCol = lCol + 1
               With wsDoc.Cells(lRow, lCol)
```

```
                    .Formula = ("=sum(" & QuotedName(ws.Name) & "!" & ws.UsedRange.Address(True,
True) & ")")
                    .Value = .Value ' convert to values; remove this if you want it to recalc
                    .NumberFormat = "General" ' in case dates in source
                End With

                ' #Rows, #Cols in used range (may not be real last occupied cell)
                rgFound = ws.Cells.SpecialCells(XlCellType.xlCellTypeLastCell)
                If Not SheetIsEmpty(ws) Then ' show blanks if empty sheet
                   lCol = lCol + 1
                   wsDoc.Cells(lRow, lCol).Value = rgFound.Row
                   lCol = lCol + 1
                   wsDoc.Cells(lRow, lCol).Value = rgFound.Column
                End If
             End If ' being checked

          ElseIf TypeName(wbCheck.Sheets(lSheet)) = "Chart" Then
             With wbCheck.Sheets(lSheet)
                If .HasTitle Then
                   sTemp = "Chart Title:" & "'" & .ChartTitle.Text & "'"
                Else
                   sTemp = "No Chart Title"
                End If
                sTemp = sTemp & ", " & .SeriesCollection.Count & " Series"
             End With
             wsDoc.Cells(lRow, lCol).Value = "'" & sTemp
          Else
             ' not a worksheet, or Chart, what is it? Dialog? Macro?
          End If

       Next lSheet
       wsDoc.Columns.AutoFit()

       GoTo Exitproc

OnError:
       Select Case ErrorHandler()

          Case vbYes, vbRetry : Stop : Resume
          Case vbNo, vbIgnore : Resume Next
          Case Else : Resume Exitproc ' vbCancel
       End Select

Exitproc:
       On Error GoTo 0 ' restore any screenupdating etc
       Application.Calculation = XlCalculation.xlCalculationAutomatic
       Application.Cursor = XlMousePointer.xlDefault
       Application.ScreenUpdating = True

   End Sub


   Function SheetExists(ByVal sName As String) As Boolean  ' check for any type of sheet -
worksheet, chart
       On Error Resume Next
       SheetExists = (StrComp(sName, Application.ActiveWorkbook.Sheets(sName).Name,
vbTextCompare) = 0) ' 0=matches
   End Function

   Function SheetIsEmpty(ByVal ws As Worksheet) As Boolean '-As Worksheet
       Dim rg As Range
       rg = ws.UsedRange
       If rg.CountLarge() = 1 Then  ' only 1 cell, probably A1
          SheetIsEmpty = IsEmpty(CStr(rg.Value))
       Else
          SheetIsEmpty = False
       End If
   End Function

   Function Array(ByVal ParamArray items() As Object) As Array
       Return items
   End Function

   Sub WriteHeadings(ByVal StartCell As Range, ByVal aHeadings As Object)
```

```
      With StartCell.Resize(1, UBound(aHeadings) - LBound(aHeadings) + 1)
         .Value = aHeadings
         .Font.Bold = True
      End With
   End Sub

   Private Function IsEmpty(ByVal p1 As String) As Boolean ' for VBA compatibility
      Return String.IsNullOrEmpty(p1)
   End Function

   Function FindFirst(ByVal rgSearch As Range, ByVal vWhat As Object, ByVal lLookIn As Long,
ByVal lLookAt As Long) As Range
      On Error Resume Next ' should check for err=0 or err=1004 being only two expected
      ' After:=rg.SpecialCells(xlCellTypeLastCell) means the first found could be first cell
in range
      FindFirst = rgSearch.Find(What:=vWhat,
After:=rgSearch.SpecialCells(XlCellType.xlCellTypeLastCell), _
           LookIn:=lLookIn, LookAt:=lLookAt, _
           SearchOrder:=XlSearchOrder.xlByRows, SearchDirection:=XlSearchDirection.xlNext,
MatchCase:=False) ', SearchFormat:=False)
      Debug.Assert(Err.Number = 0 Or Err.Number = 1004 Or Application.ThisWorkbook.IsAddin)
   End Function

   Function ErrorHandler()
      Dim sErrMsg As String
      sErrMsg = "Error " & Err.Number & IIf(Erl() = 0, "", " at line " & Erl()) & " " &
Err.Description
      Debug.Print(sErrMsg)
      ErrorHandler = MsgBox(sErrMsg, vbAbortRetryIgnore, "Error")
   End Function

   Function QuotedName(ByVal sName As String) As String ' return a  name properly quoted
      QuotedName = "'" & Replace(sName, "'", "'" & "'") & "'" ' Dec'08 --> 'Dec''08', My
Budget --> 'My Budget'
   End Function

End Module
```

Tips and workarounds

One of my Excel VBA add-ins had 13,000 lines of code and took about two weeks (full-time equivalent) to convert to VB.Net using Visual Studio 2010, Excel-Dna 0.29 and NetOffice 1.50. Bear in mind that the supporting libraries are being constantly updated so check for changes in more recent version of Excel-Dna and NetOffice. The following list of tips and gotchas was built up from that experience.

Fix these first in VBA before doing the migration

The first group are changes that are safe to make in VBA but will make the transition to VB.Net much safer. Fix any issues with Option Base 1 and ByRef first.

Option Base and Array()

In VBA, the default lower bound of an array dimension is 0 (zero). Using Option Base, you can change this to 1. In Visual Basic .NET, the Option Base statement is not supported, and the lower bound of every array dimension must be 0. Additionally, you cannot use ReDim as an array declaration. One thing to keep in mind when working with Office collections from Visual Basic .NET is that the lower array bounds of most Office collections begin with 1.
When I was converting some old code with Option base 1 I found it easy to make mistakes when converting to the 0-based arrays of vb.net so I replaced the array by a class that contained the properties I had been storing in an array. The VBA function Array() can be replicated by defining an Array() function in a GlobalHelpers.vb module:

```
Function Array(ByVal ParamArray items() As Object) As Array
    Return items
End Function
```

Or by using literal array syntax

```
Dim aHeadings() As String = {"Sheet", "Type", "Sheet Tab Name”}
```

When returning variant arrays from ranges, the only types you will get are String, Boolean, Number, or Error. Dates are returned as numbers.

ByVal and ByRef

VBA defaults to ByRef; VB.Net to ByVal. When passing parameters in VBA, be sure to explicitly specify ByRef so this can be preserved when the module is imported into VS2010. I used to do that for primitive type variables (String, Long, etc) but found I had omitted to do it for Object and Array types. This leads to bugs that can only be detected by executing test cases and comparing output with the VBA output. It would be nice if VS2010 Express could warn us of variables passed byref but changed in the sub. Is this done by tools like Migration Partner and Aivosto Project Analyzer?

‘Variant’ is no longer a supported type: use the ‘Object’ type instead. Or simply Dim with no type, which is ugly but compatible with both VBA and VB.NET. If there are multiple parameters to a function, and some are declared eg As String, then all must be declared, so use As Object where you had nothing in VBA.

There is no Range type in Excel-Dna so if you are not using NetOffice or the PIA use Dim rg As Object.

Fill in default values for Optional parameters in function headers.

Enumerations and .xl* Constants

In VBA, you can simply assign the constant because the enumeration is global to your project:

```
Set rhs = rg.End(xlToRight)
```

In Visual Basic .NET, you could either use the numeric equivalent

```
Set rhs = rg.End(-4161)
```

Or prefix enumerated constants with their type, eg XlDirection.xlUp rather than simply xlUp. The prefix can be added in VBA as well which may avoid ambiguities like xlVisible and xlSheetVisible.

To get VB IDE constants from vbext_ProjectProtection (eg vbext_pp_locked) and vbext_ComponentType (eg vbext_ct_StdModule) use

```
Imports Microsoft.Vbe.Interop
```

.Cells reference

Explicitly specify .Cells in lines like

```
For Each cell In ActiveSheet.UsedRange.Columns(2).Cells
```

Use of Parentheses with Method Calls

In VBA, parentheses are omitted when you call subroutines and only required when you wish to catch the return value of a function. In Visual Basic .NET, parentheses are required when passing parameters in a method call.

Default Properties

In Visual Basic .NET, default properties are only supported if the properties take arguments. In VBA, you can use shortcuts when typing code by omitting the default properties like .Value. in VBA you can write

```
myValue = Application.Range("A1")
```

which returns the .value but in VB.Net you need to be explicit:

```
myValue = Application.Range("A1").Value
```

This is one of the most common incompatibilities between quick & dirty VBA and VB.NET.

Assigning to cell values

Similarly to the above, be explicit about the .Value property, ie not cell = xyz but cell.value = xyz. This avoids bugs when the left hand variable is an Object where you want a variant array of values from the range.

Set Keyword

In VBA, the Set keyword is necessary to distinguish between assignment of an object and assignment of the default property of the object. Since default properties are not supported in Visual Basic .NET, the Set keyword is not needed and is no longer

supported. The IDE automatically removes the Set command from object references in code pasted in from VBA.

Erase

In VBA, the Erase statement clears the value of the elements. The VB.Net Erase statement destroys the elements of an array, and sets the array variable to Nothing.
If you add an array as an item to a collection, it will point to the same array every time unless you re-create the array for each addition to the collection.
You can clear the contents of an array in .Net using:

```
System.Array.Clear(arr, arr.GetLowerBound(0), arr.Length)
```

However, any existing pointers to that array (eg if it has been added to a dictionary collection) will now point to the cleared contents. The nearest equivalent to creating a new array is to simply ReDim the array without Preserve.

ReDim

In VBA you can write

```
Dim distance() as Long
ReDim distance(1,2)
```

But in VB.Net 'ReDim' cannot change the number of dimensions of an array. So declare the array with a comma inside the brackets to give it two dimensions:

```
Dim distance(,) As Long
```

Byte arrays

Byte arrays are a faster way to iterate through the characters of a string than MID$.

In VBA you can write

```
Dim bs1() As Byte, string1 as String
bs1 = string1
```

But in VB.Net a value of type 'String' cannot be converted to '1-dimensional array of Byte'. So use either UTF8 encoding to get the single-byte values for each character or Unicode to get two bytes per character:

```
bs1 = System.Text.Encoding.Unicode.getBytes(string1)
```

Join() function

I had used Join() on an array loaded from a range to concatenate the values into one string. VBA ignored blank cells but vb.net threw an exception because some of the variant array values were Nothing for blank cells. So I created a routine to join only non-blank values.

IsDate()

This is the same trap as in VBA – it returns True for a string value that looks like a date, so use TypeName(var)="Date" if you want to test for a Date type.

Evaluate function

This is also a function in .Net which evaluates math expressions, so change all occurrences to Application.Evaluate to get the Excel function.

Square bracket cell references

The square bracket operators evaluate the Excel expression inside and can be used to get references or range names. Instead of [C2] use .Cells(2,3)

CutCopyMode

The definition is Public Enum XlCutCopyMode As Integer so the old code Application.CutCopyMode = False should now be Application.CutCopyMode = 0

Initialising variables

There were lots of compiler warnings " Warning 5 Variable 'xx' is used before it has been assigned a value. A null reference exception could result at runtime." The solution is to initialize it eg Dim str as String=vbnullstring
Ensure a function return is defined for paths not taken, rather than relying on the default value for an uninitialized function variable.

On Error Resume Next

This practice is known to be error-prone because it ignores all errors after that line, including ones you did not expect, until reset. It should only be used in one-line functions that are intended to wrap an expected error, eg

```
  ' does a sheet of any type exist with this name?
Function SheetExists(ByVal wb As Workbook, ByVal sName As String) As Boolean
Dim oSheet As Object
    On Error Resume Next
    oSheet = wb.Sheets(sName)
    Return Err.Number = 0
End Function
```

Don't use default property of Err, that would get translated to Err(), specify Err.Number. For a better approach see
http://msdn.microsoft.com/en-us/library/ms973849.aspx
Error Handling in Visual Basic .NET

Connection objects

The old .Connection object in Excel pre-2007 has now been replaced by two specific connections: WorkbookConnection and ODBCConnection. The connection string is obtained by either

```
oConn.ODBCConnection.Connection.ToString
oConn.OLEDBConnection.Connection.ToString
```

For web queries, get the connection from the Querytable in the Ranges (if any):

```
oConn.Ranges.Item(1).QueryTable.Connection
```

Shell & Unzip

Add a reference to Windows\System32\Shell32.dll and use this code

```
Dim oShell As Shell32.Shell
oShell = New Shell32.Shell ' CreateObject("Shell.Application")
oShell.NameSpace((sFolder)).CopyHere oShell.NameSpace((sZipName)), 16
'16=Respond with "Yes to All"
```

Accessing the Windows API

In VBA:

```
Public Declare Function GetTickCount Lib "kernel32" () As Long
```

In ExcelDna:

```
<System.Runtime.InteropServices.DllImport("kernel32")> _
      Public Function GetTickCount() As Long
      End Function
```

Or, use the equivalent DotNet function

```
Function GetTickCount() as Long
      GetTickCount=Environment.TickCount
End Function
```

The following changes are specific to DotNet

String$() function

Use StrDup(number,character)

Format$()

```
Format$(0.5, "###%") works, "50%"
Format$(1234.5, "###,##0.00") also, "1,234.50"
Format$(Now, "yyyy-mm-dd hh:mm:ss ")
```

But for time intervals, where in VBA you would use

```
Format$(Now() - tStart, "nn:ss")
```

In VB.Net you have to use .ToString

```
tStart = Now() ……
(Now - tStart).ToString("hh\:mm\:ss")
```

To convert a time interval to seconds, use CLng((Now - tStart).TotalSeconds)

Class definitions

The VBA Get and Let methods are replaced by Get/Set clauses in the properties.

```
Public Property Item(ByVal Key As String) As Object
      Get
        Return KeyValuePairs.Item(Key).value
      End Get
      ' update a scalar value for an existing key
      Set(ByVal value As Object)
        KeyValuePairs.Item(Key).value = value
      End Set
End Property
```

For more examples, see the GlobalHelper class below.

Dictionary Class

The Scripting.Dictionary object can be replaced by a .Net Dictionary, using the syntax

```
dicWords = New Dictionary(Of String, WordCountItem)
```

and .Exists becomes .ContainsKey.

```
VBA:        For Each aWordCount In dicWords.Items
Vb.net:     For Each pair In dicWords
        aWordCount = pair.Value
```

AutoFilter

Similarly.Autofilter must be fully specified:

```
ActiveSheet.UsedRange.AutoFilter(field:=1, criteria1:=sCriterion,
_operator:=XlAutoFilterOperator.xlAnd, criteria2:=Nothing,
visibleDropDown:=True)
```

Named Parameters

Error 9 (Invalid index. (Exception from HRESULT: 0x8002000B (DISP_E_BADINDEX)))
That error could be caused by supplying a value of Nothing for a parameter that needs to have a specific value. For example, in the Range.Replace method, the SearchOrder needs to be xlRows or xlColumns.

Workbook_Open

This is how you implemented a workbook_open handler:

```
Imports ExcelDna.Integration                ' needed for IExcelAddIn
Imports ExcelDna.Integration.ExcelDnaUtil   ' for Application object

Class AutoOpen
    Implements IExcelAddIn
    Public Sub Start() Implements IExcelAddIn.AutoOpen
        Application.Statusbar = "Starting . . ."
    End Sub
    Public Sub Close() Implements IExcelAddIn.AutoClose
        'Fires when addin is removed from the addins list but not when Excel closes
        'This is to avoid issues caused by the Excel option
        'to cancel out of the close after the event has fired.
        ' whatever you want here
    End Sub
End Class
```

Workbooks.Open() error

error BC30469: Reference to a non-shared member requires an object reference.
This means that you need to qualify 'Workbooks'. Here are two ways:

1) ExcelDnaUtil.Application.Workbooks.Open(). This specifies the Workbooks collection completely.
2) Define a global Application object which you initialise in AutoOpen, and a global Workbooks property (see the GlobalHelpers.vb file) and then you can use simply Workbooks.Open same as in VBA.

Document Properties

Cast the workbook properties to Office Document properties like this:

```
Imports Microsoft.Office.Core
 Dim cdp As DocumentProperties
 cdp = CType(wb.CustomDocumentProperties, DocumentProperties)
```

Getting an Object's Property by Name

As well as CallByName you have Invoker.PropertyGet(workBook, "FileFormat")

Mixed .Font.Color returns DBNull

When there is more than one font colour in a cell, VBA returns 0, VB.Net returns DBNull, which can cause an error comparing DBNull with a number.

Decimal cell value type

In VBA the possible return values for VarType(cell.value) are 5,6,7,8,10,11. VB.Net adds type 14 (vbDecimal) which is what the Currency type (6) is returned as.

Controls

Dim ctl as Control requires Imports System.Windows.Forms and a reference to it.

Names index by Name

To get a Name from the string name of the Name, use the .Item method

```
wb.Names.Item(strName)
```

That works in VBA too.

http://msdn.microsoft.com/en-us/library/office/microsoft.office.interop.excel.names.aspx

says "Use Names(index), where index is the name index number or defined name, to return a single Name object." However, it throws an error if you pass a string name. The index number is 0-based. To get by name, use the Item method:

Function Item(Optional ByVal Index As Object = Nothing, Optional ByVal IndexLocal As Object = Nothing, Optional ByVal RefersTo As Object = Nothing) As Microsoft.Office.Interop.Excel.Name

http://msdn.microsoft.com/en-us/library/microsoft.office.interop.excel.names.item(office.14).aspx

Parameters

Index

Optional Object. The name or number of the defined name to be returned.

IndexLocal

Optional Object. The name of the defined name in the language of the user. No names will be translated if you use this argument.

RefersTo

Optional Object. What the name refers to. You use this argument to identify a name by what it refers to.

You must specify one, and only one, of these three arguments.

VBComponents index by Name

To get a Name from the string name of the Name, use the .Item method

```
wb.VBProject.VBComponents.Item(strName)
```

That works in VBA too.

Shapes index by Name

To get a Shape from the string name of the Shape, use the .Item method

```
ws.Shapes.Item(strName)
```

That works in VBA too.

Replace DataObject by Clipboard Class
VB.Net has a Clipboard object with .GetText and .SetText methods which replace the VBA Dataobject.

SendKeys
Use System.Windows.Forms.SendKeys.Send("{TAB 3}")

#Error values, CVErr(), IsError()
This seems to have become a mess in .Net. Excel-Dna has one set of coding to handle ExcelDna.Integration.ExcelError values passed into, or returned from, UDFs:
http://groups.google.com/group/Excel-Dna/browse_thread/thread/31e62ad3e2e218b3
But there are different codes when accessing cell error values in code:
http://xldennis.wordpress.com/2006/11/29/dealing-with-cverr-values-in-net-part-ii-solutions/
The CVErr function is gone, the link above has an article by Mike Rosenblum which provides alternatives. The IsError function in VB.Net is nothing to do with the VBA IsError, so again a workaround is needed. In VBA accessing a cell value with an error display of ###### could cause an overflow error 6 in VBA, eg a negative date; this does not happen in VB.Net, it returns the date before 1900, or a Double. I will try the workaround `StrDup(Len(cell.Text), "#") = cell.Text`

Restore Excel Status
Place the main code in a Try/Catch that resets event handling, screen updating, etc.

Unable to get the Paste property of the Worksheet class
This may be caused by a second instance of Excel running.

Debugging strategies

In VBA Debug.Print is a command that can take multiple arguments with comma and semi-colon separators. In VB.Net Debug.Print(str) takes only one string argument which must be enclosed in parentheses. An alternative is Debug.Write(str) and Debug.WriteLine(str)

Resume or GoTo labels cannot be within a With statement: " Error 69 'GoTo Retry' is not valid because 'Retry' is inside a 'With' statement that does not contain this statement."

With the use of On Error there will be many occurrences of 'first chance' errors, ie those which are trapped by the On Error statement, eg
A first chance exception of type 'System.Runtime.InteropServices.COMException'
Therefore there is no need to worry about 'first chance' exceptions.

You can catch other ones (eg InvalidCastException, NullReferenceException) which should be preventable by enabling debugging on these exceptions but turning off the COMException. In VS2010, click Debug > Exceptions, check 'Thrown' for Common

Language Runtime Exceptions, (User-unhandled should be checked too), expand the tree and Uncheck 'System.Runtime.InteropServices.COMException'

See the example GlobalHelper to illustrate the use of the DebugConsole class.

VS2010 has support for breakpoints etc. If you stop debugging (Ctrl+Alt+Break) the Excel instance will be closed and the next time it is restarted it will warn:
Excel experienced a serious problem with the 'test2 (ribbon helper)' add-in. If you have seen this message multiple times, you should disable this add-in and check to see if an update is available. Do you want to disable this add-in?
Be sure to click No, otherwise the Ribbon handler will be disabled in all Excel-Dna add-ins from that point on.

In VS Express you include this in the .vbproj file so F5 will launch the debugger:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
<StartAction>Program</StartAction>
<StartProgram>C:\Program Files\Microsoft Office\Office14\EXCEL.EXE</StartProgram>
<StartArguments>MY_ADDIN_GOES_HERE.XLL</StartArguments>
</PropertyGroup>

<PropertyGroup>
<ProjectView>ShowAllFiles</ProjectView>
</PropertyGroup>
</Project>
```

XML Documents

In VBA you could use oXDoc.SelectSingleNode("//book") but in .Net you need to specify the namespace:

```
oXDoc = New Xml.XmlDocument
oXDoc.Load(sXMLFileName)
Dim root As XmlElement = oXDoc.DocumentElement
Dim nsmgr As XmlNamespaceManager = New XmlNamespaceManager(oXDoc.NameTable)
nsmgr.AddNamespace("x", root.NamespaceURI) ' x is our temp alias
sNode = "//x:books" ' XPath
oXSectionNode = oXDoc.SelectSingleNode(sNode, nsmgr)
sNode = "//x:books"
oXSectionNodes = oXDoc.SelectNodes(sNode, nsmgr)
'or could do oXDoc.GetElementsByTagName("books")
```

UserForms

These had to be redrawn from scratch, there does not seem to be a way to import them. In one case I renamed a form and started getting an error
The class frmOptions can be designed, but is not the first class in the file.
I could not fix that so simply deleted and re-created the form again.

In VBA the forms can be referred to by name, eg frmOptions. In VB.Net they have to be instantiated, otherwise you get an error
Reference to a non-shared member requires an object reference
http://msdn.microsoft.com/en-us/library/zwwhc0d0(v=vs.80).aspx
says: Avoid adding the Shared keyword
My solution is to name the Class FormCount, instance is Dim frmCount as FormCount, so the same variable name frmCount can be used in the code.

The controls have different properties and methods from VBA.
In VBA I could add listbox or combobox items by .List = array, in vb.net it is
`frmOptions.cboClass.Items.AddRange(ary)`

The VB.Net ListBox is limited to ONE column of strings, no MultiColumn property. I could use a ListView but that's more complex than I want. So I created a sub to load the items from a range, concatenating the two columns into one string. On the code that used the listbox I then had to parse the item back into its elements, so there is no escaping some added complexity.
VBA .Value of a listbox becomes .SelectedValue
The click event of a listbox is `lbxWords_SelectedIndexChanged`
VBA .Value of a checkbox becomes .Checked.
VBA .Caption of a control becomes .Text.

VBA has a .ShowModal property; in vb.net use either the .Show method for non-modal or .ShowDialog for modal.

The events have to have a “handles” clause added to the declaration:

```
Private Sub cmdOK_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles cmdOK.Click
```

The parameters can be omitted for simplicity:

```
Private Sub cmdOK_Click() Handles cmdOK.Click
```

In VBA you can access the .Designer property to get lists of controls, that does not seem to be available in .Net.

Troubleshooting Ribbon Interface complications

The following reference document shows both the VBA and the VB.NET (they call it Visual Basic) signatures for all the callbacks:
http://msdn.microsoft.com/en-us/library/aa722523(v=office.12).aspx
Customizing the 2007 Office Fluent Ribbon for Developers (Part 3 of 3)

The ribbon XML is stored in the ProjectName.dna file. For some reason to do with the Ribbon context, it can only see procedures in the Ribbon handling class. If you give it any other sub to call, it will not find it. The only ways to connect them are either:

a) A button onAction with the special value of "RunTagMacro" which is an Excel-Dna procedure that calls Application.Run(Tag).
`onAction="RunTagMacro" tag="MySub"`
It does not pass any parameter to the macro. Be aware that if the tag name already exists in any other VBA addin loaded before the Excel-Dna XLL addin, that sub will be run instead. You have to be sure that the Tag is unique.

b) Or, a button has an onAction with a sub name that is present in the Ribbon handler class. This is the only way of passing the control to the sub which can then examine the ID and Tag if desired.

Here is some code that illustrates both of those:

```
<DnaLibrary Name="TasksAnalyzer Add-in" RuntimeVersion="v4.0">
   <ExternalLibrary Path="TasksAnalyzer.dll" />
   <CustomUI>
      <customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
         <ribbon startFromScratch="false">
         <tabs>
         <tab id="Tab_CCTWAVB" label="Tasks Analyzer" keytip="Q" insertAfterMso="TabAddIns">
            <group id="CCTWAVBgroup" label="Task Analyzer (VB)">
            <button id="CCTWAVBButton1" onAction="RunTagMacro" tag="TA_CountInSelection"
               label="Analyze Task" screentip="Analyze Tasks Frequency "
               size="normal" imageMso="WordCountList" />
            <button id="CCTWAVBButton2" onAction="CCTWAVBButton2" tag="TA_ShowFrmCount"
               label="Filter Form" screentip="Show Form to filter analyzed data" size="normal"
               imageMso="FilterAdvancedByForm" />
            </group>
         </tab>
         </tabs>
         </ribbon>
      </customUI>
   </CustomUI>

</DnaLibrary>
```

The Ribbon class is

```
Imports ExcelDna.Integration.CustomUI   ' needed for IRibbonControl
Imports ExcelDna.Integration.ExcelDnaUtil   ' for Application object
Imports System.Runtime.InteropServices  ' needed for <ComVisible(True)>

<ComVisible(True)>
Public Class Ribbon

    ' implement ExcelDna.Integration.CustomUI.ExcelRibbon to get full Ribbon access.
    Inherits ExcelRibbon
    ' Regular Macros can be called from the Ribbon using the onAction='RunTagMacro' helper,
    ' which calls a macro named in the tag attribute of a button.
    ' Under the covers, RunTagMacro just calls ExcelDnaUtil.Application.Run(tagName).
    'the onAction="MyMacro" attribute will only run methods on the ExcelRibbon-derived class.
    'http://groups.google.com/group/Excel-Dna/browse_thread/thread/60b7fac567c17505
    'IDs have to be unique, can't use same ID for something like "RunIDMacro" more than once.
```

```
    ' This is another way I can think of to pass a parameter from the Ribbon into a macro
    Sub RunTagMacroWithID(ByVal ctl As IRibbonControl)
        Application.Run(ctl.Tag, ctl.Id)
        ' If I try to pass ctl as an object I get
        'Cannot run the macro 'TestRibbon'. The macro may not be available in this workbook or
all macros may be disabled.
    End Sub

    Sub CCTWAVBButton2(ByVal ctl As IRibbonControl)
        Try
            TA_ShowFrmCount()
        Catch ex As Exception
            MsgBox(ex.Message, MsgBoxStyle.Exclamation, ex.Source)
        End Try
    End Sub

End Class
```

The available keytip letters in Excel are B, C, G, K, Q, S; others are already used by Excel builtin ribbon commands, eg H for Home.

If the addin menu does not appear in the Ribbon

First paste the XML into the Custom UI Editor (open any document in the editor first) and click Validate, to check for bad XML syntax or duplicate control or group IDs.

Ensure you have <ComVisible(True)> in the Ribbon class (and therefore Imports System.Runtime.InteropServices)

In your .dll you need to be sure that the class is Public and ComVisible.
You can do this by marking the class as <ComVisible(True)> or by setting COM-Visible for the assembly:
Go to the project properties, and select the Application tab. Then click on Assembly Information and set the "Make assembly COM-Visible" checkbox. (This sets the assembly-wide [assembly:ComVisible(true)] attribute in your AssemblyInfo.cs file.)

By default (if there is no assembly-wide attribute) assemblies are ComVisible, which is why the code in the .dna file works. But the Visual Studio library template sets the assembly attribute to false by default. The assembly-wide attribute can always be overridden by an attribute on the particular class.

(Note that COM-Visible is a different setting to "Register for COM Interop" which you should never set for an Excel-Dna library.)

If a Ribbon at some point causes Excel to crash, all custom Ribbons will be disabled in the future. To re-enable custom ribbons, go to Excel Options => Add-Ins => Manage: Disabled Items Go... => Click on the disabled Excel-Dna addin and click Enable.

Global Helper Functions

Prefix enumerated constants with their type, eg XlDirection.xlUp rather than simply xlUp (which is Global in VBA). This can be done in the VBA as well, so it is compatible both ways. In the absence of that, the change has to be done in the VB.Net IDE after copying the code from the VBA addin. One way of simplifying the amount of editing is to define a GlobalHelper.vb module that provides some compatibility code for constants and properties like Application.Selection.

The code is in the GlobalHelper.vb file on the web site:
http://www.sysmod.com/GlobalHelper.vb
It defines functions and properties to support the VBA Array(), IsEmpty(), IsNull(), IsObject(), Round(), Selection, ActiveCell, ActiveSheet, ActiveChart, ActiveWindow, ActiveWorkbook, and the Workbooks collection.

"Reference to a non-shared member requires an object reference"

This error from "Workbooks.Open(…" illustrates a typical need for the helpers such as Workbooks. Firstly I must create an OBJECT called "Application" Then EITHER I change all "Workbooks" in the VBA code to an explicit reference to Application.Workbooks which requires editing the code

OR I create a public module with a Property Workbooks. That is Govert's solution, and results in a Global Helper with lots of ReadOnly Properties like:

```
Public Module GlobalHelper

    ReadOnly Property Application As Application
        Get
            Return ExcelDnaUtil.Application
        End Get
    End Property

    ReadOnly Property ActiveWorkbook As Workbook
        Get
            Return Application.ActiveWorkbook
        End Get
    End Property

    ReadOnly Property Workbooks As Workbooks
        Get
            Return Application.Workbooks
        End Get
    End Property

End Module
```

Compatibility with VBA code that references ThisWorkbook

I typically store some setup parameters in properties and worksheets in the xlam file. Instead of worksheets, you can use configuration files, see the following section. Config file Appsettings can only be simple key/value string pairs. For complex structures, it may be simpler to distribute a spreadsheet file that contains the setup worksheets, as long as that is always installed with the .XLL file.

As an exercise I defined a ThisWorkbook class in the AddInMain.vb module and created properties as follows:

```
'Create an ExcelAddIn-derived class with AutoOpen and AutoClose,
'and add a module called AddInMain to hold the Application object reference:
Imports LateBindingApi.Core
Imports NetOffice.ExcelApi
Imports ExcelDna.Integration

' This class is implemented only to allow us to initialize NetOffice
' We hook up a public field in the Module AddInMain
'    that will be usable anywhere in the project.
Public Class AddIn
    Implements IExcelAddIn

    Public Sub AutoOpen() Implements IExcelAddIn.AutoOpen
        ' must initialise here because XlCall cannot be used from Ribbon context, only in a
macro context
        ThisWorkbook.Name = System.IO.Path.GetFileName(XlCall.Excel(XlCall.xlGetName))
        ThisWorkbook.Path = System.IO.Path.GetDirectoryName(XlCall.Excel(XlCall.xlGetName))

        Factory.Initialize()
        ' Set a public field in a module, so that Application will be available everywhere.
        Application = New Application(Nothing, ExcelDnaUtil.Application)

    End Sub

    Public Sub AutoClose() Implements IExcelAddIn.AutoClose

    End Sub
End Class

Public Class ThisWorkbook
    'Shared means we don't need to instantiate ThisWorkbook to call these

    Shared Property Title As String = DnaLibrary.CurrentLibrary.Name

    Shared Property Name As String = "ThisWorkbook"

    Shared Property Path As String = "."

    Shared ReadOnly Property Worksheets As Object
        Get
            MsgBox("No Worksheets in ThisWorkbook")
            Return Nothing
        End Get
    End Property

    Shared Function IsAddin() As Boolean
        Return True ' for debugging
    End Function

End Class
```

Using .config files

You need to:

1) Create a text file Myaddinname.xll.config and specify key & value pairs
2) In the VB code, add a reference to System.Configuration and import it so that the code need only refer to ConfigurationManager.AppSettings("keyname")

ExcelDna will include the .config file if you created a packed .xll, thereby simplifying deployment.

```
' Myaddinname.xll.config
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="DatabasePath" value="%USERPROFILE%\Database" />
    <add key="SupportEmail" value="me@sysmod.com" />
  </appSettings>
</configuration>
```

```
' MyAddinname.vb
Imports System.Configuration            ' Add Reference: System.Configuration
'...
Dim dbPath As String = ConfigurationManager.AppSettings("DatabasePath")
Dim email As String = ConfigurationManager.AppSettings("SupportEmail")
'...
```

To iterate all elements:

```
Dim appSettings As Specialized.NameValueCollection = ConfigurationManager.AppSettings
For i As Integer = 0 To appSettings.Count - 1
   Debug.Print("Key : {0} Value: {1}", appSettings.GetKey(i), appSettings.get(i))
Next i
```

See also:

http://msdn.microsoft.com/en-us/library/system.configuration.configurationmanager.appsettings(v=vs.80).aspx

Unit Testing using NotePad++ as the code editor

To simplify testing isolated pieces of code, I used NotePad++ to edit a .Dna file and copied ExcelDna.xll to the test filename.xll. Then to run the test, I saved a command in Notepad++ assigned to Ctrl+F5:

```
"C:\Program Files\Microsoft Office\Office14\EXCEL.EXE" "$(CURRENT_DIRECTORY)\$(NAME_PART).XLL"
```

Here is a sample test .Dna file:

```
<DnaLibrary Language="VB" RuntimeVersion="v4.0" >  <Reference
Path="F:\Programs\_PIA Excel\Microsoft.Office.Interop.Excel.14.dll"
/><![CDATA[
  ' lines above CDATA are not counted in the compiler error messages
  '    Add 1 to reported error line number to get line in this file

Imports Microsoft.Office.Interop.Excel

Public Module MainProcs

Sub TestASub()
try
       ' this works thanks to the Global Helper class Property "Workbooks"
       dim wb as Workbook=Workbooks.Add()

       dim ws as Worksheet=wb.worksheets(1)

       ' use Application.Range for refs to other workbooks or sheets
              msgbox(Application.range("Sheet2!A2:C3").Address(false,false,
xlReferenceStyle.xlA1,true) _
                     & vbcrlf &
ws.range(ws.cells(2,1),ws.cells(3,3)).address(false,false,
xlReferenceStyle.xlA1,true))

              wb.Close(SaveChanges:=False)

       catch Ex as Exception
              msgbox(Ex.toString ,vbExclamation,"Error")

              Console.WriteLine(Ex.toString)
       end try

       Application.Quit

End Sub

End Module

Public Class DnaAddIn
       Implements IExcelAddIn ' ExcelDna.Integration included implicitly from
ExcelDna.xll copied to $(NAME_PART).xll

    Public Sub AutoOpen() Implements IExcelAddIn.AutoOpen

              TestASub()

       End Sub

    Public Sub AutoClose() Implements IExcelAddIn.AutoClose ' Must be
declared even if not used
    End Sub

End Class

' This module contains <global> shortcuts to the Application members.
Public Module GlobalHelper

    ReadOnly Property Application As Application
```

```
        Get
            Return ExcelDnaUtil.Application
        End Get
    End Property

    ReadOnly Property ActiveWorkbook As Workbook
        Get
            Return Application.ActiveWorkbook
        End Get
    End Property

    ReadOnly Property Workbooks As Workbooks
        Get
            Return Application.Workbooks
        End Get
    End Property

End Module

]]>
</DnaLibrary>

<!-- Notepad++      Ctrl+F5
       Run "C:\Program Files\Microsoft Office\Office14\EXCEL.EXE"
"$(CURRENT_DIRECTORY)\$(NAME_PART).XLL"
-->
```

Performance Testing

As an example, I tested the Levenshtein distance function with two 100 character strings. This does 100x100 or 10,000 MID$() operations to compare those two strings. This was in turn called 100 times. The timings are:

1. 3900 ms for 100 iterations of the function using VBA and MID$() operations and WorksheetFunction.MIN().
2. 234 ms for 100 iterations of the function using VBA and Byte arrays and in-line logic for MIN().
3. 2886 ms for only one iteration of the XLL function using WorksheetFunction.MIN().
4. 156 ms for 100 iterations of the XLL function using in-line logic for MIN(), and the MIDS() operations.
5. 63 ms for 100 iterations of the function using the XLL and Byte arrays.

Bear in mind that performance optimisation applies just as much to VBA as VB.Net. For example, a test of 1 million iterations of three versions of a Minimum function in pure VBA performed as follows:
The longest time was Application.Min at 7862 ms
A UDF took 889 ms
In-line logic took 62 ms

```
Sub testmin()
Dim m As Long, min1 As Long, min2 As Long, min3 As Long
Dim i As Long, lTime As Long
min1 = 3
min2 = 2
min3 = 1
lTime = GetTickCount()
For i = 1 To 1000000
   m = Application.WorksheetFunction.Min(min1, min2, min3)
Next
Debug.Print GetTickCount - lTime; " ms"
'Application.Min 7862 ms
'Application.WorksheetFunction.Min 3292 ms
'WorksheetFunction.Min 3166 ms

'Since Min() function is not a part of VBA, use UDF

lTime = GetTickCount
For i = 1 To 1000000
   m = Min(min1, min2, min3)
Next
Debug.Print GetTickCount - lTime; " ms" ' 889 ms/million

'Finally test using inline logic for min of 3
lTime = GetTickCount
For i = 1 To 1000000
   If min1 <= min2 And min1 <= min3 Then
       m = min1
   ElseIf min2 <= min1 And min2 <= min3 Then
       m = min2
   Else
       m = min3
   End If
Next
```

```
Debug.Print GetTickCount - lTime; " ms" ' 62 ms/million
End Sub

Function Min(ParamArray values()) As Double 'VBA ParamArray must be Variant
Dim i As Long
Min = values(0) 'always Base 0 for Paramarray
For i = 1 To UBound(values)
   If values(i) < Min Then Min = values(i)
Next
End Function
```

Background reading

Going further to C#, C++

Charles Williams recommends this automatic code converter to ease the transition from VB to C# and C++ :
http://tangiblesoftwaresolutions.com/

http://smurfonspreadsheets.wordpress.com/2010/02/
Simon Murphy reviews ExcelDna, XLL+

Currently (August 2012) the only documentation on Excel-Dna is effectively the Google group. http://ExcelDna.typepad.com is old, not updated since 2006.

The following is extracted from http://groups.google.com/group/Excel-Dna

On Feb 2, 11:36 pm, Govert van Drimmelen <gov...@icon.co.za> wrote:
With Excel-Dna you can talk to Excel using either
1. using the C API or
2. using the COM object model.
If you're coming from VBA, the COM automation interface will be more familiar. So I'll focus on option 2.

Excel-Dna has no built-in definitions for the COM object model types like 'Range' and 'Worksheet'. However, Excel-Dna gives you a way to get to the 'root' Application object that you need for any other access to the COM object model - just call ExcelDnaUtil.Application and you'll get an Application object that refers to the instance of Excel hosting your add-in. (Something like CreateObject("Excel.Application") may or may not give you the right Excel instance.)

From your VB.NET add-in you can now talk to the COM automation types either:
2(a) Late-bound. Your variables are types as 'Object' (the .NET equivalent of Variant), either explicitly or implicitly by not giving a type. Then code like this will work:

```
Dim xlApp
Dim myRange
xlApp = ExcelDnaUtil.Application
myRange = xlApp.Range("A1")
```

The disadvantage is that you have no intellisense and no checking at compile time.

2(b) Early-bound using an interop assembly. In this case you reference a .NET assembly that contains definitions for the COM object model. This interop assembly defines the types like 'Range' and 'Worksheet' to your .NET code.

There are two options for the interop assembly:
2(b)(i) Use the official version-specific Primary Interop Assembly (PIA). This is where the namespace Microsoft.Office.Interop.Excel comes from. You can download and install the Office 2010 versions here:http://www.microsoft.com/download/en/details.aspx?id=3508. Once installed, you'll be able to add references to the assembly called 'Microsoft.Office.Interop.Excel', where the 'Range' and 'Worksheet' types are defined.

2(b)(ii) Use a version-independent interop assembly like the NetOffice assemblies.

In both cases you need to make sure that you use the ExcelDnaUtil.Application object as the root of your access to the object model.

The type ExcelDna.Integration.ExcelReference is defined by Excel-Dna as a thin wrapper around the C API datatype used to indicate a sheet reference. In the sense that it denotes a part of a sheet, it is similar to the COM Range type, but to actually use the ExcelReference type you would typically pass it as an argument to some other C API calls. The only helper methods currently there are methods to to get and set data in the referenced area.

On Feb 3, 11:50 am, Govert van Drimmelen <gov...@icon.co.za> wrote:

If you use any of the interop assembly options (NetOffice or PIA) you can say:
Dim ws As Worksheet

Then you have to have some definition of that type at runtime too, so you need to have the interop assembly at the client too (however, see the 'Embed Interop Types' option below).

Otherwise, if you have no interop assembly referenced, you can say
Dim ws As Object
or equivalently (and also compatible with VBA, as you understand)
Dim ws

Not having an interop assembly means no intellisense and no checking for mistakes at compile time. So you could have a mistake like:

```
Dim myRange
myRange = Application.ActiveSheet.Ragne("A1")
```

and the error would only appear at runtime.

The performance of the late-binding (particularly from VB.NET) is very good, so not really an issue. And there is no deployment issue since you are not referencing additional libraries.

The PIA assemblies are installed in the Global Assembly Cache (GAC) so you should reference then from the .NET tab of the Add Reference dialog, and look for "Microsoft.Office.Interop.Excel" and "Office" -so do not browse to them explicitly. They should not be copied to your output directory either, since they live in the .NET GAC. To deploy to another machine, you need to run the downloaded installer for the Office Primary Interop Assemblies, which puts the PIA assemblies in the GAC and does some registration in the registry.

For the NetOffice assemblies you can just copy them somewhere, Add Reference and Browse there and Copy to Output. Then with the packing you can put them inside the .xll, so no other files or registration would be needed.

Actually if you are using .NET 4 (Visual Studio 2010) and targeting a single Excel version, say Excel 2010, there is another option I have not mentioned yet. The option was added in .NET 4 to embed some interop information in the assembly that you compile. I have not tried this myself, but it might work well for you. To do this you reference the PIA (via the .NET tab on Add References, not browsing) and then in the Reference properties you set "Embed Interop Type: True". That should put the required information in your compiled assembly, and then you don't have to distribute the interop assemblies to other users. This will only work under .NET 4, and probably won't work with the NetOffice assemblies since they are not 'Primary' interop assemblies.

Microsoft and other sources

http://blogs.msdn.com/b/pstubbs/archive/2004/01/15/59049.aspx
Convert VBA to VB .Net and C#

http://msdn.microsoft.com/en-us/vstudio/ms788236
Free Book - Upgrading Microsoft Visual Basic 6.0 to Microsoft Visual Basic .NET

http://www.upsizing.co.uk/Art52_VBAToNet2.aspx
Converting Access VBA to VB.NET – General Principals

http://msdn.microsoft.com/en-us/library/aa192490%28v=office.11%29.aspx
Converting Code from VBA to Visual Basic .NET
Office 2003
For more information, see Introduction to Visual Basic .NET for Visual Basic Veterans.
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vboriintroductiontovisualbasic70forvisualbasicveterans.asp

Data Access: see Comparison of ADO.NET and ADO.

Conversion of UserForms to Windows Forms. VBA UserForms cannot be copied or imported into Visual Studio .NET. In most cases, you will need to recreate your forms as Windows Forms. Many new form controls are also available in Visual Basic .NET, such as data-entry validators, common dialog boxes, hyperlinked labels, system tray icons, panels, numeric-up/downs, on-the-fly designable tree views, Help file linkers, ToolTip extenders, and more.

http://msdn.microsoft.com/en-us/library/aa168292(v=office.11).aspx
Understanding the Excel Object Model from a .NET Developer's Perspective

http://msdn.microsoft.com/en-us/library/kehz1dz1(vs.71).aspx
Introduction to Visual Basic .NET for Visual Basic Veterans

http://msdn.microsoft.com/en-us/library/kehz1dz1(v=vs.90).aspx
Help for Visual Basic 6.0 Users

http://msdn.microsoft.com/en-us/library/office/bb687883(v=office.14).aspx
Microsoft Excel 2010 XLL Software Development Kit

http://oreilly.com/catalog/vbdotnetnut/chapter/appa.html

DEBUGGING notes

http://social.msdn.microsoft.com/forums/en-US/netfxbcl/thread/25fd387a-ca88-4ac6-8aee-8e208a4e66cd/
The Debug class only uses those listeners that are assigned to it. By default that is the output window. You can programmatically add more listeners to it. However the Debug class is only available in debug builds so it is limited in use for production tracing. The Trace class is for tracing in either mode and happens to share the same listeners. You can assign additional listeners to the Debug/Trace classes either programmatically or through the config file. In the config file you do so using the <system.diagnostics>\<trace>\<listeners> element. These listeners have no impact on trace sources.
So, in a nutshell, create a trace source for each distinct area that you want to trace. Use the trace source to log information. In your config file define each source and associate it with one or more listeners. This also allows you to specify the level of tracing. For example you might want to see all messages from the data layer but only the errors from the business layer.
If you just want to have your Trace statements copied to a log file, you don't need to use FileLogTraceListener directly. Just add the following to your app.config system.diagnostics section:

```
    <trace autoflush="true">
      <listeners>
        <add name="myListener"
type="System.Diagnostics.TextWriterTraceListener" initializeData="trace.log"/>
      </listeners>
    </trace>
```

For an example in ExcelDna, see the sample in \Distribution\Samples\Packing\PackConfig\

Name the file as MyAddInName.xll.config and set Copy To Output to If Newer. Define a <trace....> section as shown above. You don't need the <sources...> stuff that VS automatically adds if you add an Application config file rather than a simple text file.

In VB.Net, TRACE and DEBUG are automatically defined; in VS it's in Project > Properties > Compile > Advanced

#END# version 6-Nov-12
Patrick O'Beirne
Mail3 at sysmod.com
Web: http://www.sysmod.com
Blog: http://sysmod.wordpress.com
LinkedIn: http://www.linkedin.com/in/patrickobeirne
Skype: sysmodltd