Why use C# code from a Visual Objects application?
The best reason in the world to consider using Visual Objects and C# together is because it works! Your VO application works already (or it should if you are considering this technique), and the C# code works too (or it should).

By now many VO developers have large production applications that constantly need enhancements, either to continue to make them attractive to new buyers or in response to customer requests. VO developers may have spent many years developing these apps, and they are very well tested in the field. So there is always a risk when making major changes, and with customers depending on your VO apps to run their businesses, the risk isn’t just for the developer alone. The lowest risk solution is the solution the VO developer has already used during the lifetime of the production app – small incremental changes added gradually that are thoroughly tested as they are added.

If you are reading this paper, I assume you are already familiar with the benefits of working in C#. The language and Visual Studio development environment are very solid and productive, and Microsoft spends millions of dollars annually making sure they continue to be so. The latest enhancements to .NET are generally available first in C#. There are a lot of samples and shared source code available as well as popular commercial 3rd party libraries. There are books, articles, and videos available as well as blogs and newsgroups to help with any issues. Probably you are using C# already, and if you aren’t, probably you should consider it! But you don’t have to use C# to take advantage of the .NET framework and development environments. If you prefer VB or Vulcan.NET, that’s ok too. (And if you prefer Vulcan.NET, the Vulcan Getting Started Guide already shows you how to achieve the COM-visible approach to extending VO applications—that’s why this paper is for the C# developer.)

How do you create a COM-visible .NET class?
Visual Studio (I’m using the 2008 version here) makes creating a C# library that is exposed as COM very easy. First create a new project that is a class library. (Choose the name of the library with some care as it will become part of your ProgId.)
You will get code that looks something like this:
Right-click on the Project name in the solution explorer to view the project properties.
On the Application tab, click the Assembly Information button.
Check the Make assembly COM-Visible checkbox and then click OK.

Next go to the Build tab. Set the Platform target as x86. (The COM-visible class will be hosted in-proc inside your VO app, which is a 32-bit application.) Also check the Register for COM interop checkbox.
Now build the project. If you are running with User Account Control enabled, you may see this sort of error message:

The solution is included in the error description – you just need to run Visual Studio as administrator.

So now you have a class library that compiles but does nothing. The next step is to change the name of the class to something more descriptive like Calculator and then add some methods. (Source code listings are provided at the end of this paper.)

1. You must have a public constructor.
2. Add any other public methods that you might need like this:
Build your library again. Next test to see if it is usable from VO:

Create a new Terminal app in VO, adding the OLE library. Using a COM-visible C# class is as easy as this:
Now run the code to see if it works:
How easy is that?

You may notice that I am using late-bound calls to the COM object from VO. You can make one small change to the C# code to make the public methods visible for early-bound access if you prefer like this:
Now you can use OLE-COM Object Viewer to see the public interface for CalculatorLibrary by selecting the File, View Typelib... menu option and navigating to the TLB file in your project’s \bin folder.

Similarly, you can use the Visual Objects Tools, OLE server tool to generate wrapper code:
### Automation Server Base Class Generation

<table>
<thead>
<tr>
<th>Server</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>CalculatorLibrary</td>
<td>C:\Testing\CalculatorLibrary\CalculatorLib</td>
</tr>
<tr>
<td>CAPICOM v2.1 Type Library</td>
<td>C:\Program Files (x86)\Common Files\Microsoft</td>
</tr>
<tr>
<td>CashFlowProjector</td>
<td>C:\Program Files (x86)\Common Files\Intuit</td>
</tr>
<tr>
<td>CDInf</td>
<td>C:\Windows\SysWOW64\cdinf.dll</td>
</tr>
<tr>
<td>CertCli 1.0 Type Library</td>
<td>C:\Windows\system32\certcli.dll</td>
</tr>
<tr>
<td>CertEnc 1.0 Type Library</td>
<td>C:\Windows\system32\certenc.dll</td>
</tr>
<tr>
<td>CertEnroll 1.0 Type Library</td>
<td>C:\Windows\system32\certenroll.dll</td>
</tr>
<tr>
<td>cic 1.0 Type Library</td>
<td>C:\Windows\system32\cic.dll</td>
</tr>
<tr>
<td>ClassLibrary6</td>
<td>C:\cavo28\bin\ClassLibrary6.lib</td>
</tr>
<tr>
<td>OpenHealer 1.0 Type Library</td>
<td>C:\Windows\system32\infoTMM.dll</td>
</tr>
</tbody>
</table>

#### Code Generation Options

<table>
<thead>
<tr>
<th>Class name: MyCalculator</th>
<th>Interfaces: Calculator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include description info</td>
<td>Remove Underscores</td>
</tr>
<tr>
<td>Include containing objects</td>
<td>Prefix</td>
</tr>
</tbody>
</table>

[Show interfaces] [Generate source] [Open TypeLib] [Close]
And you can use the code in your Visual Objects app with Intellisense support for the available methods:
I am showing the early-bound approach for completeness and because sometimes it becomes a religious belief with programmers, but I do not recommend it for production code. Of course you can do whichever you prefer, but I see COM-visible C# classes intended for use by production VO apps as organic – something that you will continue to add to over time by just adding additional public methods. As you do so, you would need to regenerate the VO code required for early binding, and that could destabilize parts of your production VO app that you don’t intend to change. The benefit of early binding – that you get compile time checking of the method names – might be helpful in some situations where you have tons of exposed methods you’re dealing with. But it still doesn’t guarantee that the current version of the COM class has that method, so you have to test anyway. And if you don’t have a correct method name, you will find out immediately so it is an easy thing to test. So for the rest of this paper, the ClassInterface attribute is commented out.
Write test code while you are still in C#!

Already I have broken two of my rules for working with COM-visible .NET classes. The first is showing you the technique for early bound access, and the second is jumping straight into a VO demo. Although it may be tempting to just go straight to wiring up your new C# library to the VO app, and for such a simple class as this one it’s not a big deal, but I don’t generally recommend doing that yet.

What I recommend is first testing the C# code in C# where it’s easier to debug. Add a new project to your CalculatorLibrary solution call Test, remembering to set the build target to x86:

And set a reference to the CalculatorLibrary project:
Now write the test code:
Now you can set breakpoints in the Calculator class code and debug it:
Debug into your Visual Studio-built library from the VO application

It’s not as easy as debugging on the C# side, but it is possible.

1. Create an EXE from your VO test app.
2. In Visual Studio, right-click on the CalculatorLibrary and select Properties, then from the Debug tab, select the Start external program in the Start Action section, providing the full path to your VO exe:
Now you can run Debug on the CalculatorLibrary.dll directly from Visual Studio, and the Visual Studio debugger will first launch the VO executable. (Right-click on the CalculatorLibrary project and select Debug, Start new instance from the context menu):
Usually the code in the COM-visible methods will be quite simple and mainly just call private helper methods to do the real work. These helper methods can be as complex and powerful as you like (and I generally put them in their own source files). In my production code, I have some that launch .NET forms. These forms may even include an ActiveX control on them (COM calling .NET calling into COM!) or they may work with a web service. The only limit to what they can do is your imagination, because you are on the .NET side of the world now! You probably don’t need to worry about the time it takes to move from the Win32 world of VO to COM to .NET and possibly even to COM again – it’s really not noticeable in my production apps and in one case the result is faster than the previous VO code using an ActiveX control was, but then I don’t do that stuff over and over again in a loop either!

The one exception I make to my rule of making C# do the hard work is in the rare situation where I need to read or write data to a DBF file. VO is really good at working with DBF data, so I make the VO app responsible for handling that task.

And although C# can work with SQL data just fine using ADO.NET, in practice I usually leave that task to the VO side too since I am generally already working with SQL on the VO side and have the data I need before I need to invoke the COM-visible C# method. Perhaps in a future version of my big apps I’ll write a middle data layer in C# that handles all the SQL access, and just call that data layer from VO.
Host a .NET form in a VO application.
Now let’s make the C# code a bit more interesting by adding a form to the CalculatorLibrary project. Right-click on the CalculatorLibrary project, then select Add, Windows Form from the context menu:

![Add New Item - CalculatorLibrary](image)

I’ve added a few controls and buttons to the C# form so the user can test a variety of Sums and Products:
And I’ve added some code to the buttons’ click event handlers by double clicking on each button in the designer, then typing in some code:
Finally I added one more public method to the Calculator class so the C# form can be launched from VO:
Then build the CalculatorLibrary.dll again. It’s probably also a good idea to add test code to the C# Test project, but I won’t show that step here. Instead I’ll just show the change to the VO code:
And finally run the VO test app again:
This is obviously a contrived, very simple example, but there’s really no limit to the complexity you can have on the .NET side of things. The .NET Framework is very powerful, and there are many 3rd party controls available, so don’t feel constrained by your Visual Objects limitations! I use the new form approach in production for credit card validation, which is handled by a web service. I don’t think my users have any idea that what they see as a single application is created in two very different environments and languages using different tools. All they care is that it works!

**Consume a web service from a VO application.**

Often web services provide some of the most powerful and useful ways to extend an application with new functionality. I use a proprietary third party web service for credit card processing in my production VO apps. Mapping web services such as those from Bing and Google are also popular. But if you want to just test using web services in an application, there are also a number of free web services that are available from sites such at [http://www.webservicex.net/ws/default.aspx](http://www.webservicex.net/ws/default.aspx) or XMethods ([www.xmethods.net](http://www.xmethods.net)). For my sample I’m using the Ip Address to Country service listed on XMethods at [http://www.ecubicle.net/iptocountry.asmx?wsdl](http://www.ecubicle.net/iptocountry.asmx?wsdl).

To use this web service from your C# library, right-click on the CalculatorLibrary project and select Add Service Reference from the context menu. When the Add Service Reference dialog appears, click the Advanced button. Then on the Service Reference Settings dialog click the Add Web Reference button:
Enter the url of the web service and click Go. Rename the web reference to IpToCountryWebService and click the Add Reference button.

Now you have added a reference to the web service to the project, you can use it like any other class. I’ve added a method to the Calculator class called GetCountryFromIp to use the web service:
The FindCountryAsXml() method returns a .NET object of type XmlNode, and the data I want is in the node’s InnerText, so that’s what I return to the calling function. Other web services I’ve worked with may return simply the string of data I want or perhaps more complex Xml which I have to parse (but again this is pretty easy using the .NET XmlDocument class.) So you may have to modify your .NET code slightly based on what the web service you’re using returns to you.

In production you should always wrap calls to web methods like this inside a try...catch block since the web service may not be available sometimes, but I’m not showing that code here for simplicity. In production I generally return an empty string (if a string type is expected) in the catch block.

The next step is to add code to your VO app to test consuming the web service from VO:
And the result is the United States for my company’s web site:
How do you deploy your VO app with the COM-visible assemblies?

The easiest way to do this is to use a modern installer that can register DLLs for COM as part of the installation process. Here’s a screenshot from the installer I’m using, Advanced Installer, showing that I’ve checked the Auto register file checkbox for the DLL’s properties, but most modern installers support this feature in some fashion. In this case I have selected build time synchronization, since I don’t want to have to remember to take an extra step with each build if a COM-visible assembly is changed:

![Screenshot of Advanced Installer showing auto registration settings](image)

That’s not the only way to get a COM-visible .NET assembly installed on a user machine though. There is a utility called regasm.exe that can be used directly, which I include in the installer with the other files that my apps need. I also have a splash app that checks a network location provided in a config file to see if newer executables should be copied to the user’s machine from a network share, then it launches my big VO app. Here’s the code that shows updating a COM-visible assembly programmatically:
Since I’m using file version to determine if a newer version is needed, I need to be sure to update the AssemblyInfo.cs file for the library when I make changes. The screenshot below doesn’t show any changes, but for the next version I’d change the assembly and file version numbers. I usually log revision history here too as comments in production code as well as filling out the AssemblyCompany, AssemblyProduct, AssemblyCopyright and AssemblyTrademark attributes so my assemblies don’t get released with a Microsoft copyright.
Tips and Tricks
I’ve used COM-visible C# libraries with my production VO apps for several years now, so I’ve picked up a dozen tips along the way which I’m listing in no particular order:

1. Use data types on both the VO and C# side that are understood by both. Generally simple data types like strings, ints and bools work fine, but don’t expect a C# method to understand what to do with a VO DATE data type or expect VO to know what to make of a return value of a .NET DateTime! Instead use the string representation of the date on the interface and convert between DATE and DateTime on each side. For collections, I use a delimited string on the interface and convert between a VO ARRAY and a .NET List<T> (or other .NET collection type) on each side. A delimited string is also handy for returning multiple values from a C# method to VO.
2. Keep the interface code simple. Hide the complex details in private C# methods.
3. Set VO and Visual Studio to drop executables into the same folder to make testing easier if both executables, for example, need to access a file in the executable folder.
4. Always test C# methods in C# to make sure they work correctly before attempting to use them from VO. Yes, you can debug from VO, but it’s easier to do this in C# for something complex.

5. Make sure all .NET executables target the x86 platform. I really wish Microsoft had made this the default for Visual Studio!

6. When working with 3rd party .NET code, I write wrapper code – a shim – that I expose to COM, rather than changing the 3rd party code to expose it directly to the VO app. Use a try...catch block around web service calls since the service might not always be available.

7. Group functionality into libraries based on how (and to whom) they’ll be deployed. It’s easier if you keep the number of production libraries you’ll be maintaining reasonably small.

8. Use a modern installer to handle COM registration for you. You’re using an installer anyway for a big production app and all its DLLs, so why not use a good one?

9. But also use regasm.exe directly to manage COM registration if that makes sense for your app.

10. Run Visual Studio as administrator whenever you’ll be building a COM-visible assembly.

11. Use the OLE/COM Object Viewer to see how your assembly is visible to the rest of the world. The main thing to check is the spelling of the ProdId.

12. And finally, be sure to back up all your source code – the VO code and the C# code.

Conclusion
Those of us who have large production apps written in Visual Objects over the years are really lucky. We have solid, thoroughly tested code that’s benefitted from years of user feedback, and if we’re smart, we are reluctant to “throw the baby out with the bath water” if new features are easier to implement in .NET (as increasingly is the case). Using COM-visible .NET assemblies to extend VO applications gives us the best of both the managed and unmanaged worlds, and is an easy, low-risk solution to extending the useful life of Visual Objects applications.

Acknowledgements
Thanks to Geoff Schaller for his suggestions for making the samples both simpler and also more extensible. Thanks also to Terry Bourne and Lars Broberg for testing the samples in Visual Studio 2010 Beta 2. And thanks to everybody else who read early versions of this paper and tried it out.

For More Info
Rich Newman’s blog post on calling a .NET library from Excel:
http://richnewman.wordpress.com/2007/04/15/a-beginners-guide-to-calling-a-net-library-from-excel/ The references he lists at the bottom of his article are also well worth a look.

Paul Piko’s “Vulcan.NET at Warp Speed” provides a clear explanation of how to do the same thing in Vulcan.NET that my paper shows for C#:
http://www.govulcan.net/portal/GettingStarted/tabid/59/Default.aspx
Appendix 1: Source listing for class1.cs

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
/// add this line for early bound access
///using System.Runtime.InteropServices;

namespace CalculatorLibrary
{
    // Decorate the class with this attribute to make early-bound access possible:
    ///[ClassInterface(ClassInterfaceType.AutoDual)]
    public class Calculator
    {
        // public constructor is required
        public Calculator()
        {
        }

        public int Sum(int i1, int i2)
        {
            return i1 + i2;
        }

        public int Product(int i1, int i2)
        {
            return i1 * i2;
        }

        public string ShowCalculator()
        {
            Form1 form = new Form1();
            form.ShowDialog();
            return "Done!";
        }

        public string GetCountryFromIp(string ipAddress)
        {
            // Use web service to get country info, which is returned as an XmlNode obejct
            IpToCountryWebService.ipToCountry service =
                new CalculatorLibrary.IpToCountryWebService.ipToCountry();
            System.Xml.XmlNode node = service.FindCountryAsXml(ipAddress);

            // The data we want is the text from the xml node
            return node.InnerText;
        }
    }
}
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace CalculatorLibrary
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnSum_Click(object sender, EventArgs e)
        {
            int i, j;
            if (int.TryParse(textBox1.Text, out i) && int.TryParse(textBox2.Text, out j))
                textBox3.Text = (i + j).ToString();
            else
                MessageBox.Show("Can't convert test values to ints!");
        }

        private void btnProduct_Click(object sender, EventArgs e)
        {
            int i, j;
            if (int.TryParse(textBox1.Text, out i) && int.TryParse(textBox2.Text, out j))
                textBox3.Text = (i * j).ToString();
            else
                MessageBox.Show("Can't convert test values to ints!");
        }
    }
}
Appendix 3: Source listing for UpdateDll method

```csharp
private void UpdateDll(string exeName, string exePath, string newExePath)
{
    string exeFullPath = Path.Combine(exePath, exeName).Substring(6);
    if (File.Exists(exeFullPath))
    {
        string newExeFullPath = Path.Combine(newExePath, exeName);
        string regasmPath = Path.Combine(exePath, "regasm.exe").Substring(6);
        if (File.Exists(newExeFullPath))
        {
            FileVersionInfo info = FileVersionInfo.GetVersionInfo(exeFullPath);
            FileVersionInfo newInfo = FileVersionInfo.GetVersionInfo(newExeFullPath);
            if (String.Compare(newInfo.FileVersion, info.FileVersion) > 0)
            {
                if (MessageBox.Show("A newer version of " + exeName + " is available.\r\nOverwrite current version?", "New Version Available", MessageBoxButtons.YesNo, MessageBoxIcon.Question, MessageBoxIcon.Question, MessageBoxButtonsDefaultButton.Button2) == DialogResult.Yes)
                {
                    try
                    {
                        // Unregister old DLL
                        ProcessStartInfo pInfo = new ProcessStartInfo(regasmPath, @" /u " + exeName);
                        Process p = Process.Start(pInfo);
                        if (p == null)
                        {
                            MessageBox.Show("Unable to run " + regasmPath + @" /u " + exeName);
                            return;
                        }
                        p.WaitForExit();
                        File.Copy(exeFullPath, exeFullPath + info.FileVersion.Replace(’,’, ’-’), true);
                        File.Copy(newExeFullPath, exeFullPath, true);
                        // Regasm new DLL
                        ProcessStartInfo pInfo2 = new ProcessStartInfo(regasmPath, @" /codebase " + exeName);
                        Process.Start(pInfo2);
                    }
                    catch (Exception e)
                    {
                        MessageBox.Show(e.Message);
                    }
                }
            }
        }
    }
}
```
Appendix 4: Source listing for TestCalculator  Start function

FUNCTION Start()
LOCAL o AS OleAutoObject

o := OleAutoObject("CalculatorLibrary.Calculator")
IF o:init
// ? o:Sum(2,3)
// ? o:Product(5,6)
// ? o:ShowCalculator()
? o:GetCountryFromIp("24.199.182.238")

WAIT

ENDIF

RETURN NIL
Appendix 5: Source listing for program.cs in C# Test app

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using CalculatorLibrary;

namespace Test
{
    class Program
    {
        static void Main(string[] args)
        {
            Calculator o = new Calculator();
            Console.WriteLine(o.Sum(3, 4).ToString());
            Console.WriteLine(o.Product(5, 8).ToString());
            Console.ReadLine();
            o.ShowCalculator();

            Console.WriteLine(o.GetCountryFromIp("24.199.182.238"));
        }
    }
}
```