Calling .NET Components from Visual FoxPro with wwDotnetBridge

By Rick Strahl
www.west-wind.com
rstrahl@west-wind.com

The Microsoft .NET Framework has now been around for 10 years and has steadily been gaining ground on the Microsoft Platform. .NET reaches now into most aspects of Windows and for desktop Windows applications .NET is now fairly standard. As Windows has evolved more and more, .NET code is also available to take advantage of Windows system features, replacing the heavy reliance on the Windows API of yore.

As a FoxPro developer you might not immediately see .NET as a relevant tool to integrate with, but you would be surprised how much useful functionality is buried in the .NET Framework Runtimes alone. And that's not even talking about Microsoft System SDKs that aren't directly part of Windows, third party toolkits and open source libraries of which there are plenty. There's a lot of .NET functionality out that is ready to be plucked by your FoxPro code!

In this session I'll cover a brief introduction of COM Interop with native .NET. COM Interop is a feature that is built into .NET that allows it to both access external content via COM and be used as a COM Server. Via COM Interop it's possible for FoxPro applications to access .NET content.

I've written extensively about this topic and if you'd like a more thourough introduction on native COM Interop with .NET I suggest you take a look at this article:

Using .NET COM Components from Visual FoxPro

This document only covers traditional COM Interop as it works out of the box with .NET very briefly. Instead in this article the real focus is on a library called **wwDotnetBridge** that vastly expands on the capabilities of COM Interop by providing many additional features.

wwDotnetBridge is free and open source and you can use it freely in your applications. You can find the latest version including all source code for the library itself on GitHub or the project's home page:

- **wwDotnetBridge Home Page**
- **wwDotnetBridge on GitHub**

Without further ado let's jump in.

A quick Review of COM Interop

Before jumping into wwDotnetBridge it's a good idea to review the basics of COM Interop with .NET to understand what works, and what doesn't with what .NET provides natively.

.NET natively provides COM Interoperability to allow both calling of COM Components from .NET and to call .NET Components via COM. In this article I only cover the latter – accessing .NET Components from Visual Foxpro code.

For .NET to be exposed to COM natively, the COM components created must be registered as .NET COM objects. This means that they must be compiled with COM registration enabled, and must be marked to be visible to COM.

When you create a .NET class you have an option to specify that you want that type to export to COM. Once the component has been compiled is has to be registered via a special tool called RegAsm in order to be COM accessible.

.NET objects compile for COM are registered in the registry like COM objects, but they include additional registry keys.

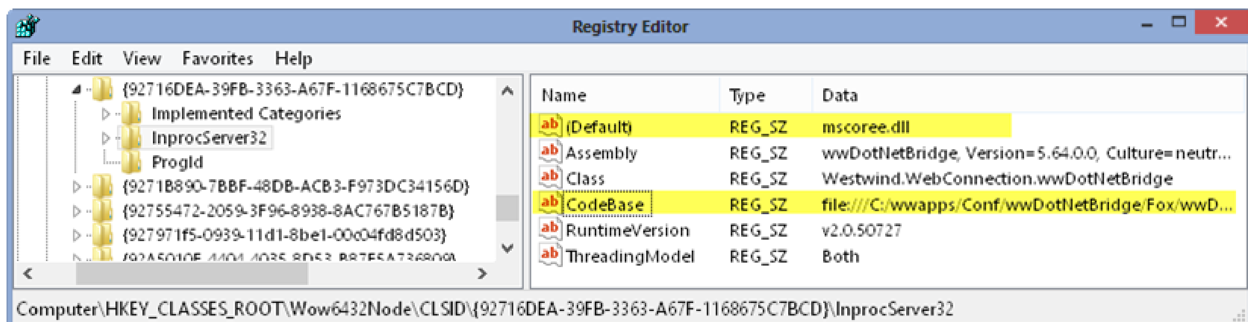

Figure 1 – *.NET Components registered with use special keys to load .NET classes. Mscoree.dll is the .NET COM proxy that routes and invokes the actual .NET class.*

When a .NET COM object is invoked it actually invokes mscoree.dll which acts like COM proxy. Based on the registry keys the .NET runtime figures out which class to instantiate and then routes all COM calls to and from this class.

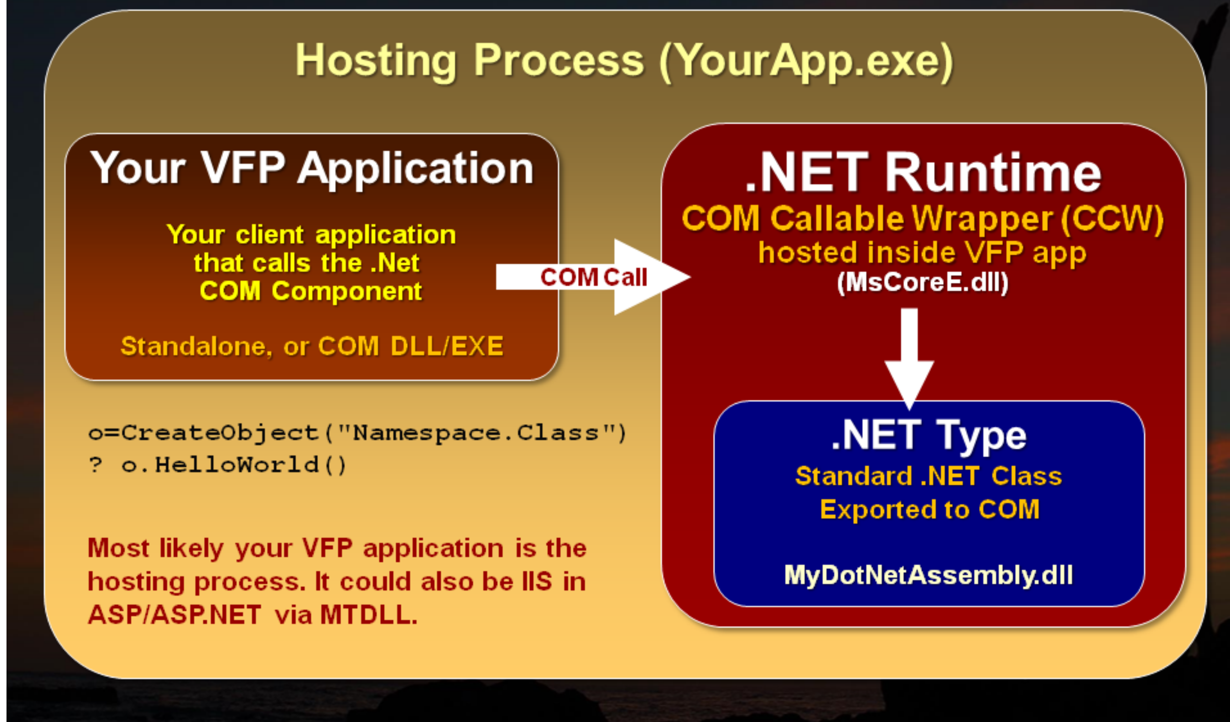

Figure 2 – *.NET COM access invokes the .NET Proxy which then forwards the COM calls into the .NET class.*

It's fairly easy to create .NET COM object, but one big shortcoming of this COM access scheme is that a .NET object has to be explicitly exported to COM in order for it to be COM accessible. Very few native .NET Components are exported to COM.

This leaves this mechanism of COM Interop primarily for compiling your own .NET classes and compiling and registering them to COM. It's not really a general purpose mechanism to access components.

Creating a .NET Component and calling it from Visual FoxPro

So let's take a quick look of what it takes to create a .NET COM Component and call it from FoxPro.

In this article I'll use Visual Studio 2012 but any version of Visual Studio will work to play along. Preferably you'll want to use .NET 4.0, but everything also works with newer versions of .NET.

To start:

- Open Visual Studio

- Create a new Project and call it InteropExamples
- Select Visual C# then Class Library
- Name the class Examples.cs

Now go to the examples.cs file and add a simple .NET class like this:

C# - Our first .NET Class Ready for COM Interop

```
using System;
using System.Runtime.InteropServices;

namespace InteropExamples
{
    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.AutoDual)]
    [ProgId("InteropExamples.Examples")]
    public class Examples
    {

        public string HelloWorld(string name)
        {
            return "It's a helluva World, " + name;
        }

        public decimal Add(decimal number1, decimal number2)
        {
            return number1 + number2;
        }

    }
}
```

Next make sure you configure your project to automatically export any COM objects marked up with COM references by going to the project node in the solution explorer, right clicking and then choosing properties. Go to the Build tab and check the *Register for COM Interop* checkbox.

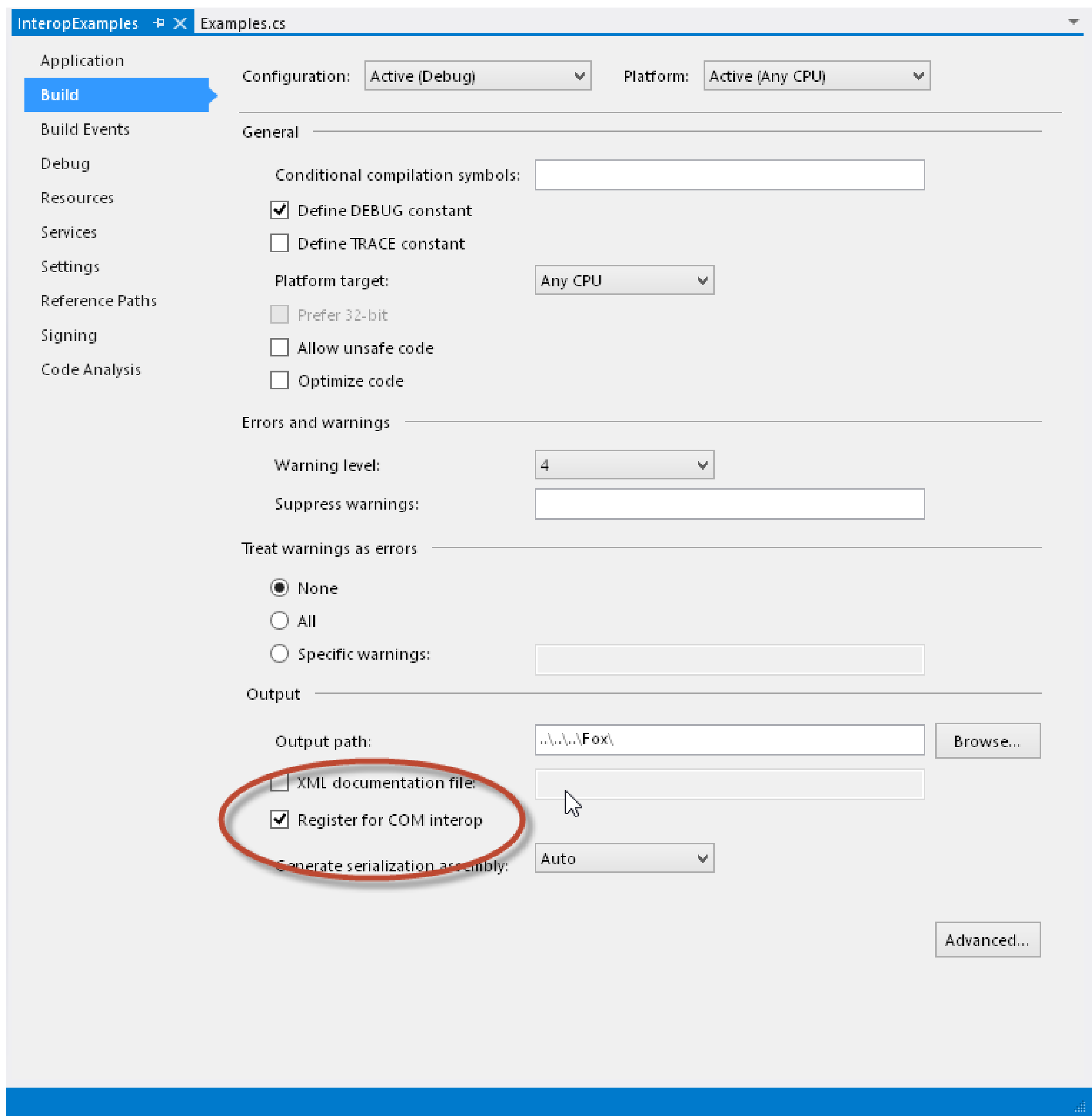

Figure 3 – *In order for your COM object to be COM accessible it needs to be registered on your machine.*

In order for your component to be accessible over COM it needs to be registered on your dev machine and it's easiest to do this with the checkbox in Figure 3. When you deploy to your client however you need to register this component using the .NET RegAsm utility. RegAsm lives in the .NET Framework directory and should be run like this:

RegAsm "c:\dev\ComInteropExamples.dll" /codebase

This can be done as part of an installer if necessary and some installer provide options to do this explicitly for you, but it's tricky because you have to find right version for RegAsm etc.

There's more info on this process and a FoxPro helper that can automate this process for you in the old COM Interop article. There is also a registercomponent.prg that demonstrates how to perform registration from Visual Foxpro code.

Using the .NET COM Component in FoxPro

Now that the component has been compiled and registered it's very easy to use it in Visual FoxPro:

```
o = CREATEOBJECT("InteropExamples.Examples")
? o.HelloWorld("Southwest Fox")
? o.Add(10,20)
```

The name of the COM object is the name of the [ProgId] attribute I specified in the .NET class. If I omit this the ProgId the default is the .NET namespace.classname. In this case the ProgId attribute wasn't necessary since the name is the same. As you can see it's pretty easy to create a COM object and use it from FoxPro

Let's add some more functionality to the class by returning an object. Let's create two .NET classes in a Person.cs file:

C# - A couple of small business entities to use as examples

```
[ComVisible(true)]
[ClassInterface(ClassInterfaceType.AutoDual)]
public class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Address Address { get; set; }
    public DateTime Entered { get; set; }

    public Person()
    {
        Address = new Address();
    }
}

[ComVisible(true)]
[ClassInterface(ClassInterfaceType.AutoDual)]=
public class Address
{
    public string Street { get; set; }
    public string City { get; set; }
    public string Zipcode { get; set; }
    public string CountryCode { get; set; }
    public string Country { get; set; }
}
```

These classes are nested with address being a property in the Person class. Next let's create a couple of methods that can create, update and read Persons in the .NET component:

C# - Simple object access methods for the Person class

```
public List<Person> Persons { get; set; }

public Person GetNewPerson()
{
    return new Person();
}

public bool SavePerson(Person person)
{
    if (person == null)
        return false;

    if (Persons == null)
        Persons = new List<Person>();

    if (person.Id != 0)
    {
        var matched = Persons.Where( p=> p.Id == person.Id)
                             .FirstOrDefault();
        if (matched != null)
        {
            matched = person;
            return true;
        }
    }

    Persons.Add(person);

    return true;
}

public Person GetPerson(int id)
{
    Person person = Persons.Where(p => p.Id == id)
                             .FirstOrDefault();
    return person;
}
```

The code sets up a list of persons (List<Person> using generics) that holds any persons I add. This will be our 'data store' that temporarily holds person objects while the object is in scope.

GetNewPerson() gives me an empty person record that I can fill from FoxPro, a SavePerson() method to save the data and GetPerson() that lets me retrieve a previously saved person record. It's like a mini data-less repository.

Now compile the code. If you still have your FoxPro window open you probably find that you can't compile the .NET DLL because it's locked – FoxPro has the DLL loaded in memory and while loaded the compiler can't update the physical file on disk. To get the compile to work you have to shut down Visual FoxPro.

Once compiled let's write some code to use the sample code from FoxPro:

FoxPro – Accessing the Person Repository Functionality is pretty straight forward

```
LOCAL loFox as InteropExamples.Examples
loFox = CREATEOBJECT("InteropExamples.Examples")

loPerson = loFox.GetNewPerson()
loPerson.Id = 1
loPerson.FirstName = "Rick"
loPerson.LastName = "Strahl"
loPerson.Address.Street = "32 Kaiea Place"
loPerson.Entered = DATETIME()
loFox.SavePerson(loPerson)

loPerson = loFox.GetNewPerson()
loPerson.Id = 2
loPerson.FirstName = "Markus"
loPerson.LastName = "Egger"
loPerson.Address.Street = "213 Mud Lane"
loPerson.Entered = DATETIME()
loFox.SavePerson(loPerson)

loPerson = null

loPerson = loFox.GetPerson(1)
? loPerson.FirstName + " " + loPerson.Address.Street
```

The code is really straight forward and as you would expect. I create a new Person record which passes a .NET object to FoxPro. I populate the .NET object with data in FoxPro and then call SavePerson() with the original instance to update the data in .NET. I repeat for the second record.

The last two lines then retrieve a customer by ID by searching the list of Person object in the Persons collection and returning a match or null. It all works as expected. As you can see it's pretty straight forward to access .NET objects in FoxPro.

Running into Problems

So far so good, but now let's look at something that doesn't work with regular COM Interop: Notice that the loFox object has a Persons property, which happens to be of type List<Person>. This is a .NET generic type which means the compiler fixes up the type of List object created at compile time. Generics are a compiler templating tool that creates customized classes using the generic parameter (Person in this case) as a template – essentially the compiler creates a custom list class with Person elements here. It's a very powerful and popular feature for creating strongly typed classes of generic types like lists, dictionaries or even things like business objects or other objects that work with other subtypes.

The problem is that COM doesn't marshal generic types when using plain COM Interop. The issue is that the types are semi-dynamic and are not exported to COM. If you use the

command line to access the object and type loFox. you'll see that the Persons list is not shown:

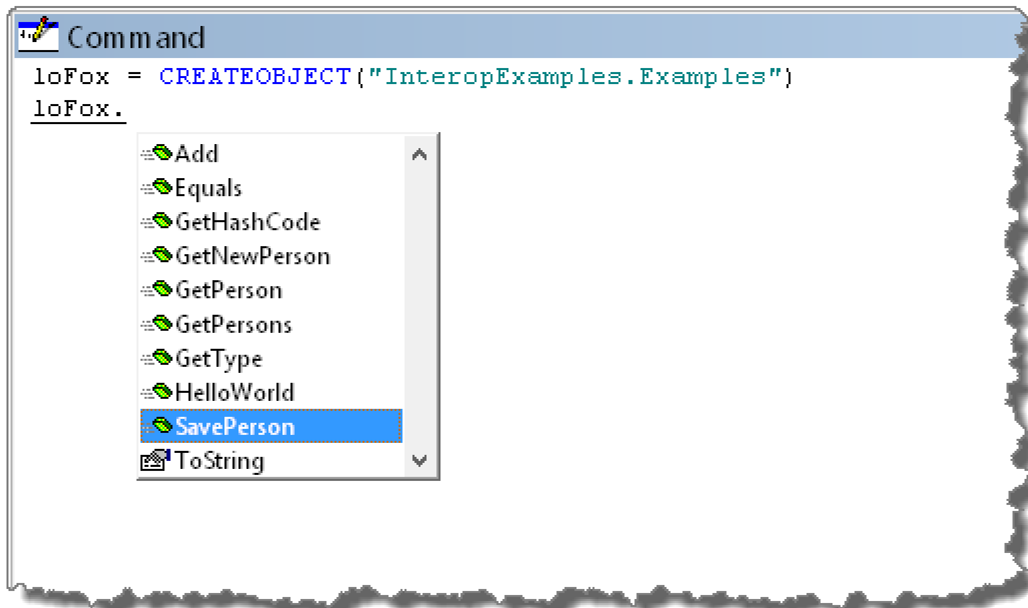

Figure 4 – *The native COM Interop object doesn't show the Persons List property because it's a generic type.*

Because the type is generic COM Interop doesn't publish the type. It's still there and you can sort of access it, but it doesn't work quite as you'd expect:

```
? loFox.Persons  && (Object)
? loFox.Persons.Count  && Error
```

In short for this example, accessing the loFox.Persons object is not possible directly. The workaround for this, is to create another method to return the list as an array that can be used by FoxPro.

```
public Person[] GetPersons()
{
    return Persons.ToArray();
}
```

In FoxPro you can then use this code:

```
loPersons = loFox.GetPersons()
? loPersons[1].FirstName  && Rick
```

While this works it has a couple of other problems. First it requires that you control the .NET assembly you are accessing – if you are dealing with a CLR or third party component you can't easily add a method to that component. One option is to create a .NET wrapper class

that provides this functionality but this can be tedious because there quite a few things beyond just generics that doesn't work natively.

The other problem is that COM Interop marshals the array to FoxPro as a FoxPro array rather than the original .NET array. .NET arrays have a .Length property and a bunch of methods to allow you to manipulate the array, but that functionality is lost when the array is marshaled to FoxPro and turned into a FoxPro array. Once in FoxPro you can use FoxPro array functions (like ALEN()) to determine size and re-dimension, but any manipulation of the array is local to FoxPro. Effectively this means the array is a copy of the original so once the data comes into FoxPro there's no way to update the array and have those values reflected in .NET. The array is effectively passed by value.

Shortcomings in native COM Interop

Even in this very simple example we've seen some short comings of native .NET COM Interop. The biggest is issue is that the 'official' .NET COM engine accesses .NET objects very conservatively and performs a ton of conversions that – at least in the case of Foxpro – cause objects to lose their original .NET type information.

Type Access Problems

This is a common problem with native COM Interop. It can't deal with a lot of .NET type structures. Here is a list of a few things that native COM Interop can't access:

- Generic Types
- Structures and Value Types beyond the basic built into .NET
- Enumerations
- Static Methods and Properties
- Collections and Dictionaries
- Binary Data
- Guids
- Numeric conversions (FoxPro numbers always cast as Double)

This is a just a sampling of some of the high level object types that don't work. There are lots of other scenarios.

Array Handling

FoxPro can't deal with many of .NET's Collection types, generics, or raw IEnumerable lists directly. In some cases it simply doesn't work, in others the syntax is hard to discover and even if it does work arrays are marshaled into FoxPro arrays which make by reference updates of array data nearly impossible.

COM Registration

In order .NET COM objects to be accessible they have to be registered using a custom tool called RegAsm. While COM registration provides nice and easy syntax using familiar CreateObject, registering with a special tool that requires admin rights is a pain. Few

installers support RegAsm registration to date (because it's such a marginal case as .NET COM Interop is not very widespread) and so it's often left up to the application itself to handle the registration process. I've covered this topic in the old article if you are interested.

wwDotnetBridge to the Rescue

wwDotnetBridge provides relief for many of these scenarios. It provides an alternate .NET Runtime hosting environment plus a powerful .NET proxy class that allows you to access and execute most of .NET's functionality from within the context of the .NET runtime, avoiding some of the COM Marshaling issues that are the main cause for the shortcomings. This proxy mechanism opens up most of .NET to your FoxPro applications, where native COM Interop only allowed you to access those few components that are COM enabled.

Here are some of the things that wwDotnetBridge provides:

No COM Registration Required

Perhaps one of the best reasons to use wwDotnetBridge is the ability to access .NET components that are not explicitly exported to COM. You can access just about any .NET component in the .NET Framework itself, in third party libraries, or even your own .NET components directly and without any COM registration requirements. This means if you need to call .NET components from your FoxPro code you can simply copy them with your application during installation and access them directly from disk with simple xCopy deployment. No COM registration required!

To be clear wwDotnetBridge still uses COM Interop, but it uses a different runtime hosting mechanism, that provides a hook inside of .NET to instantiate and pass object instances to and from FoxPro. By using the .NET proxy to load new .NET object instances the COM instantiation process and the registration requirement has been removed.

The code activation code is a little different – you're not using CREATEOBJECT() but rather functions in wwDotnetBridge's library to load dependency assemblies (.NET dlls) and instantiate the actual .NET class:

```
do wwDotNetBridge  && Load library

loBridge = CreateObject("wwDotNetBridge","V4")
loBridge.LoadAssembly("bin\InteropExamples.dll")
loFox = loBridge.CreateInstance("InteropExamples.Examples")

loPerson = loFox.GetNewPerson()
…
```

LoadAssembly is required only if you need to load explicit assemblies. Some of the core .NET libraries (System, System.Data, System.Web, System.Web.Services) are automatically loaded, so no need to load those explicitly if you access .NET Base Class Library (BCL) functionality. If you load external assemblies and they have dependencies, .NET automatically loads the dependent assemblies when the requested assembly is loaded. Just

make sure that all dependent assemblies are either in the same folder as the loaded assembly.

wwDotnetBridge works with .NET 2.0 by default and if you want to load .NET 4.0 (or 4.5 if it's installed) use "V4" as a parameter on the CREATEOBJECT() call. Only one version of the runtime can be loaded and first call wins.

Note that the wwDotnetBridge can be reused and typically you'll create it once and then cache it in a global Application property or public variable.

Create Objects with Parameterized Constructors

loBridge.CreateInstance() also supports instantiating types that have parameterized constructors which is a common requirement in .NET and not possible with native COM Interop. You can simply pass parameter values after the type name to access a non-default constructor of a .NET object.

```
loEventLog = loBridge.Createinstance("System.Diagnostics.EventLog",;
                                     "Application",".",;
                                     lcSource)
```

Access to Static Methods and Properties

A lot of useful native.NET Runtime functionality is contained in static methods. Static methods (and properties) are similar to global functions FoxPro meaning they are invoked without an explicit type instance. Native COM Interop doesn't have a way to access anything static. With wwDotnetBridge you can easily use code like this:

```
? loBridge.InvokeStaticMethod(
     "System.Net.NetworkInformation.NetworkInterface",;
     "GetIsNetworkAvailable")
```

to call a static method. Here no parameters are passed, but you can pass any parameters after the method name. There are also methods for GetStaticProperty() and SetStaticProperty(). Static support also enables access to .NET Enumerables (which are in essence static members on the Enum type), which is another frequently required feature of many .NET APIs.

Support for Problematic Types

We already saw one problem type in our simple example earlier - a generic type that couldn't be accessed with native COM Interop. wwDotnetBridge provides a host of indirect execution proxy methods that execute code within the .NET framework dynamically. Rather than a FoxPro object invoking a method or accessing a property over COM, wwDotnetBridge executes the .NET command **from within the .NET runtime itself**. This allows access to many features that simply don't work over plain COM Interop because the values and member invocation never actually pass over COM.

This accomplished in wwDotnetBridge with a few powerful indirect execution methods:

- InvokeMethod()
- GetProperty()/GetPropertyEx()
- SetProperty()/SetPropertyEx()
- InvokeStaticMethod()
- GetStaticProperty()
- SetStaticProperty()

These methods all run natively inside of .NET and allow access to Structures, Value Types, Enums, Generic Types, Guids, binary data and much more. Further these methods know about a few types that FoxPro can't deal with and automatically convert them. For example, Guids are a value type, which cannot be passed over COM (even with wwDotnetBridge). Instead wwDotnetBridge creates a custom ComGuid object that wraps the original GUID. All COM Nulls are passed as DbNull objects – wwDotnetBridge automatically converts DbNull to null values. FoxPro binary values do not map directly to byte[] but wwDotnetBridge automatically fixes up the binary data. There are quite a few more automatic fixups in place.

Array and Collection Handling

A lot of .NET APIs hold values in arrays and collections. There are quite a few common problems with collections passed to FoxPro. We already saw one common problem in that collections often use Generics (like List<T>) that are not supported directly via COM Interop. Using indirect referencing however you can access the Persons collection we couldn't access previous with:

```
loPerson = loBridge.GetPropertyEx(loFox,"Persons[0]")
? loPerson.FirstName
```

GetPropertyEx() allows providing the name of the property as an object hierarchy or as in this case for array or collection indexers. This works fine

Another option is to use the ComArray (Westwind.WebConnection.ComArray .NET type) functionality built into wwDotnetBridge. ComArray is a wrapper around an actual .NET array and leaves that array inside of .NET. You then use ComArray's methods to manipulate the array – adding, removing, updating elements, creating new items, clearing and so on. The key feature of ComArray is that the .NET array is never marshaled to FoxPro, so any changes you make to array elements or elements you add are immediately reflected in the live .NET array instance that still lives in .NET. You can take the ComArray instance and pass it back to .NET in lieu of the array as a parameter of a InvokeMethod() or SetProperty() call.

Another feature of ComArray is the ability to turn IEnumerable types into arrays. In the Persons example, the Persons property actually is of type List<T> which FoxPro can't access because generic types are not supported over COM Interop period. List<T> - like all arrays, collections and dictionaries – implements IEnumerable. ComArray can be used to turn the unsupported generic list into an array with the following ComArray code:

```
*** Convert List<T> into an plain array
loPersonArray= loBridge.CreateArray()
loPersonArray.FromEnumerable(loFox.Persons)

FOR lnX = 0 TO loPersonArray.Count-1
      loPerson = loPersonArray.Item(lnX)
ENDFOR
```

Voila – we can now access the generic type in FoxPro. The same approach can be used for many IEnumerable implementations. One common use case where IEnumerable gets returned is with LINQ which always returns IEnumerable<T> results which can't be passed directly over COM. With ComArray.FromEnumerable() the LINQ result can be turned into an array and accessed.

Automatic ComArray Conversion

When using the indirect wwDotnetBridge methods like InvokeMethod, GetProperty, SetProperty arrays are automatically converted to and from COM Arrays. If you pass a FoxPro array to .NET, wwDotnetBridge creates a ComArray from the FoxPro array and adds the individual items to the array. When a result comes back as an array from .NET the result will be a ComArray object.

So if I now call the GetPersonArray() method using the loBridge.InvokeMethod() the result automatically is converted to a ComArray for me:

```
*** Result is Person[] - automatically returned as ComArray
loPersonArray = loBridge.Invokemethod(loFox,"GetPersons")

FOR lnX = 0 TO loPersonArray.Count-1
      loPerson = loPersonArray.Item(lnX)
      ? loPerson.FirstName + " " + loPerson.Address.Street
ENDFOR
```

Note that the indirect methods are powerful, but optional. You can still use direct COM property/method access just as you could with plain COM Interop – you only need to use the indirect access methods when the direct access doesn't work or you need the extra fix up features that they provide.

These fix up functions are extremely powerful and give access to a ton of functionality that otherwise wouldn't be available.

Reviewing the original Example with wwDotnetBridge

To put what I described in perspective let's look at our original example, redone using wwDotnetBridge. We don't have to change anything in the .NET code – that code will work as is. However, if we use wwDotnetBridge we can remove the automatic COM registration we set up when compiling – let's do that to verify that we can instantiate the .NET Component without COM registration.

To disable COM registration go back to the Build Properties shown in Figure 3 and uncheck the *Register for COM Interop* checkbox. Make sure you exit VFP and then Re-compile your project.

Now inside of VFP try to invoke the COM object with:

```
loFox = CREATEOBJECT("InteropExamples.Examples")
```

This should no longer work – the .NET class is not registered with COM anymore.

Now let's rewrite the Persons sample code from earlier to use wwDotnetBridge.

FoxPro – Running the original example with wwDotnetBridge

```
do wwDotNetBridge && Load library

LOCAL loBridge as wwDotNetBridge
loBridge = CreateObject("wwDotNetBridge","V4")

*** Load our custom assembly and check for errors
IF !loBridge.LoadAssembly("InteropExamples.dll")
   ? loBridge.cErrorMsg
ENDIF

loFox = loBridge.CreateInstance("InteropExamples.Examples")
IF loBridge.lError
   ? loBridge.cErrorMsg
ENDIF

*** This code is identical to the native COM code
loPerson = loFox.GetNewPerson()
loPerson.Id = 1
loPerson.FirstName = "Rick"
loPerson.LastName = "Strahl"
loPerson.Address.Street = "32 Kaiea Place"
loPerson.Entered = DATETIME()
loFox.SavePerson(loPerson)

loPerson = loFox.GetNewPerson()
loPerson.Id = 2
loPerson.FirstName = "Markus"
loPerson.LastName = "Egger"
loPerson.Address.Street = "213 Mud Lane"
loPerson.Entered = DATETIME()
loFox.SavePerson(loPerson)

loPerson = null

*** Indirect referencing allows direct access to Generic list
loPerson = loBridge.GetPropertyEx(loFox,"Persons[0]")
? loPerson.FirstName + " " + loPerson.Address.Street


*** Better: Convert List<T> into a plain ComArray
```

```
loPersonArray= loBridge.CreateArray()
loPersonArray.FromEnumerable(loFox.Persons)

FOR lnX = 0 TO loPersonArray.Count-1
      loPerson = loPersonArray.Item(lnX)
      ? loPerson.FirstName + " " + loPerson.Address.Street
ENDFOR
```

The code starts out by loading the wwDotnetBridge library. It's a single PRG file plus two DLL files that comprise wwDotnetBridge.

Next you create an instance with:

```
loBridge = CreateObject("wwDotNetBridge","V4")
```

By default wwDotnetBridge loads Version 2.0 of the .NET runtime. You can load 4.0 by using "V4" or a full version number for the initialization parameter. The version number is important and only one version can be loaded at a time. This is one reason why I recommend you only create the wwDotnetBridge instance once on startup and then cache it globally for shared access in your application to ensure you always use the same load mechanism.

Next I load the InteropExamples assembly that contains the examples. If you're accessing native .NET functionality in the base library you don't need to load any assemblies as they are loaded by default. You can use a full path for the DLL if necessary – here the DLL is in the current path.

```
IF !loBridge.LoadAssembly("InteropExamples.dll")
   ? loBridge.cErrorMsg
ENDIF
```

It's a good idea to check for errors when loading assemblies so you know when something failed. Most common things are that required dependencies couldn't be loaded or you're loading a V4 DLL into the V2 runtime. Check the error for loads!

To create an instance of a .NET class use the loBridge.CreateInstance() method with a fully qualified .NET type name.

```
loFox = loBridge.CreateInstance("InteropExamples.Examples")
```

A fully qualified type name is *namespace.classname.*

Once you have a reference, it's same as if you used CREATEOBJECT() to instantiate the COM object in native COM Interop and you can run the exact same direct access code. wwDotnetBridge still uses COM to pass objects around just like COM Interop, but you get extra functionality using the indirect invocation functionality in the loBridge instance. For example, to access the Persons collection (which is an unsupported generic type) you can use this syntax:

```
loPerson = loBridge.GetPropertyEx(loFox,"Persons[0]")
? loPerson.FirstName + " " + loPerson.Address.Street
```

Using the indirect access functions you always pass a reference to the base object, plus a string of the member to access. The non-Ex methods need to access an exact member name ("FirstName","StreetName","Persons"). The Ex version allows access to nested properties ("Address.Street" and numeric Indexers ("Persons[0]").

Finally automatic result value and parmeter fixup occurs on this call:

```
*** Result is Person[] - automatically returned as ComArray
loPersonArray = loBridge.Invokemethod(loFox,"GetPersons")

FOR lnX = 0 TO loPersonArray.Count-1
      loPerson = loPersonArray.Item(lnX)
      ? loPerson.FirstName + " " + loPerson.Address.Street
ENDFOR
```

The GetPersons method returns a Person[] result in .NET, and the Invokemethod call fixes this result up to a ComArray. So you get methods like Item(x), Add(element), Remove(x), Clear() and a Count property to make it easy to manipulate the array elements and add new ones. There's also a useful CreateItem() method that can be used to create a new .NET element and pass it back to FoxPro so you can easily create new elements for most arrays.

How does wwDotnetBridge Work

wwDotnet bridge works with a few interoperating components:

- ClrLoader.dll – A Win32 loader for the .NET Runtime
- wwDotnetBridge.dll – A .NET proxy that helps access and call .NET members
- wwDotnetBridge.prg – FoxPro front end for the .NET proxy class

Figure 5 shows what the architecture of wwDotnetBridge looks like.

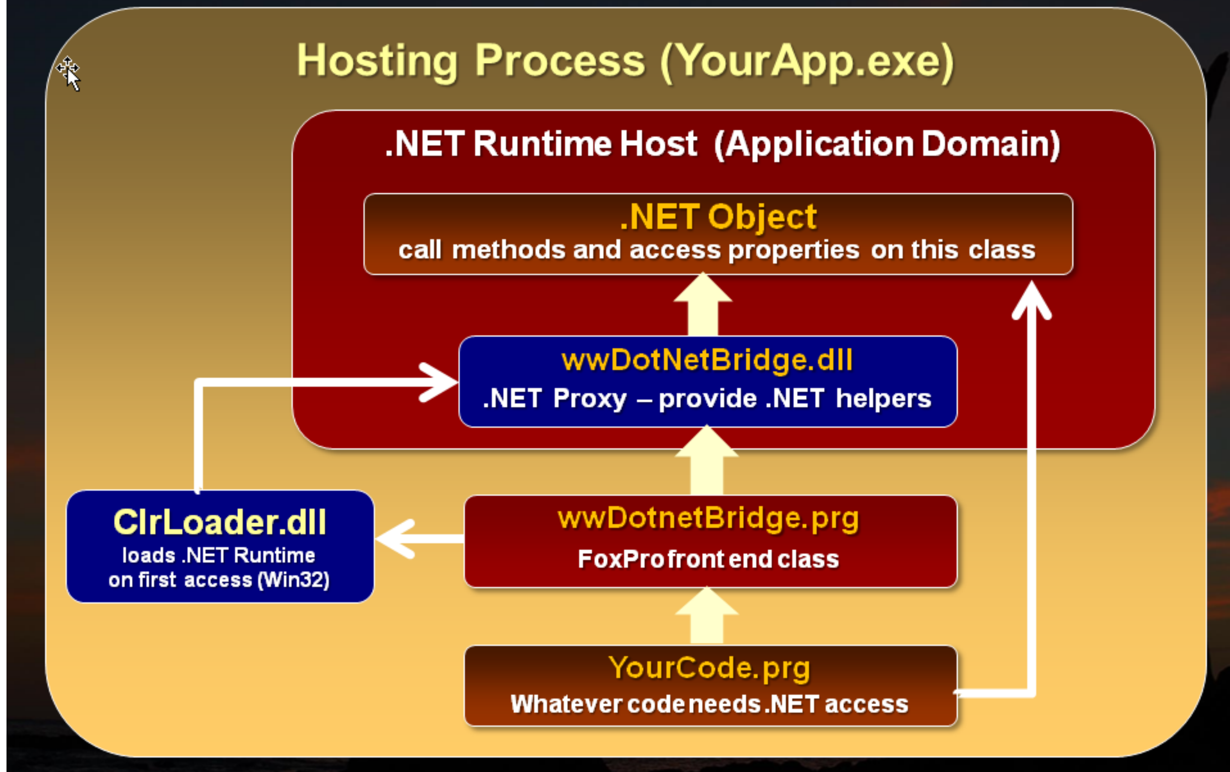

Figure 5 – *wwDotnetBridge loads the .NET Runtime on the first hit through a small Win32 loader that instantiates the .NET proxy and passes it back to FoxPro.*

When you use wwDotnetBridge and first create an instance of it, the library loads up the .NET Runtime you specify in the startup parameter:

```
loBridge = CreateObject("wwDotNetBridge","V4")
```

This accomplished through a small ClrLoader Win32 DLL (or wwIPstuff.dll in the commercial West Wind tools). ClrLoader creates an instance of the .NET Runtime, loads wwDotnetBridge.dll and creates an instance of the .NET wwDotnetBridge proxy class. The proxy is then returned back to FoxPro as a COM reference using a raw COM interface pointer.

FoxPro – The wwDotnetBridge::Load method loads the .NET runtime

```
FUNCTION Load()

IF VARTYPE(this.oDotNetBridge) != "O"
	this.SetClrVersion(this.cClrVersion)

	IF this.lUseCom
		this.oDotNetBridge = CREATEOBJECT("Westwind.wwDotNetBridge")
	ELSE
```

```
            *** Load by filename - assumes wwDotNetBridge.dll is in path
            DECLARE Integer ClrCreateInstanceFrom IN WWC_CLR_HOSTDLL
                          string, string, string@, integer@

            lcError = SPACE(2048)
            lnSize = 0
            lnDispHandle = ClrCreateInstanceFrom(
                        FULLPATH("wwDotNetBridge.dll"),;
                        "Westwind.WebConnection.wwDotNetBridge",
                        @lcError,@lnSize)

            IF lnDispHandle < 1
               this.SetError( "Unable to load Clr Instance. " + ;
                              LEFT(lcError,lnSize) )
               RETURN NULL
            ENDIF

            *** Turn handle into IDispatch object
            this.oDotNetBridge = SYS(3096, lnDispHandle)

            *** Explicitly AddRef here -
            *** otherwise weird shit happens when objects are released
            SYS(3097, this.oDotNetBridge)

            IF ISNULL(this.oDotNetBridge)
                  this.SetError("Can't access CLR COM reference.")
                  RETURN null
            ENDIF
      ENDIF

      this.oDotNetBridge.LoadAssembly("System")
ENDIF

RETURN this.oDotNetBridge
```

The *this.oDotnetBridge* instance of the .NET Proxy is what the FoxPro wwDotnetBridge class then interacts with internally with as you use the class' methods. Load is only called once when the object is instantiated. If you call this code multiple times in your application the C++ loader code tries to reload the .NET runtime, but the internal .NET APIs detect that the runtime is already running and simply loads the wwDotnetBridge .NET component into that runtime, handing back the pointer to FoxPro. While this is pretty quick, it's still a good idea to not repeatedly call this code because it has some overhead. This is one reason why caching the wwDotnetBridge instance is a good idea.

Once the runtime is loaded ClrLoader is no longer used – it only serves to retrieve the .NET proxy instance to the FoxPro wwDotnetBridge class in the form of the wwDotnetBridge::oDotnetBridge property which is then used internally to forward calls to .NET.

The wwDotnetBridge .NET proxy contains a large number of methods that provide indirect object access, which are then mapped to the Visual FoxPro wwDotnetBridge class. The proxy class serves two main purposes:

- **It’s a factory for .NET instances**
  Without the .NET wwDotnetBridge.CreateInstance() method you couldn’t load new .NET classes.
- **It’s Proxy inside of .NET**
  It’s like a Window into the inside of .NET. You can invoke methods and store content on other properties or ComValue instances allowing you essentially keep code inside of .NET without COM marshaling which provides access to many features that otherwise wouldn’t work over COM.
  FWIW, the proxy functionality could also be implemented using native COM Interop and wwDotnetBridge actually can be instantiated using native COM Interop and interact with natively invoked (CREATEOBJECT()) COM objects.

The FoxPro class holds a reference to the .NET proxy class and essentially makes pass through calls to the .NET proxy. So when you call:

```
loPersons = loBridge.GetProperty(loFox,"Persons")
```

The FoxPro code internally does:

```
FUNCTION GetProperty(loInstance,lcProperty)
RETURN this.oDotNetBridge.GetProperty(loInstance, lcProperty)
ENDFUNC
```

Some of the methods like InvokeMethod are a bit more complex to deal with multiple parameters, but in general the idea is that the FoxPro class mainly just forwards the calls to the .NET proxy and returns the results plus some parameter fixups to handle overloads a bit cleaner than the .NET calls. In theory one can also call the .NET methods directly using the loBridge.oDotnetBridge property, but the FoxPro wrappers provide a cleaner interface for FoxPro developers. For the FoxPro developer it’s an easy, single class interface to interact with.

The full source code for wwDotnetBridge – the Win32 DLL, the .NET Proxy and the FoxPro class is available and you can download it and check it out in deail from GitHub:

wwDotnetBridge Project on GitHub

wwDotnetBridge Examples

Ok, now that you have a good idea how wwDotnetBridge works let’s look at a few more examples and see what you can do in practice.

Let’s look at an example that calls a static method in built-in .NET framework function. Using plain COM Interop it’s not possible to do this as static methods are not invoked off a type reference, but rather are directly accessed as static functions.

The following receives the status of the machine’s network connection:

```
loBridge = CreateObject("wwDotNetBridge","V4")
 ? loBridge.InvokeStaticMethod(
        "System.Net.NetworkInformation.NetworkInterface",;
        "GetIsNetworkAvailable")
```

If you're connected to the network, this method call returns true. If you disconnect your network cable or shut off the wireless adapter it will return false. Super simple, but very useful especially for applications that might be mobile and need to check for connectivity first before up or downloading data.

InvokeStaticMethod() works by providing the full .NET type that contains the static method – in this case System.Net.NetworkInformation.NetworkInterface. You specify the class and method as string values, followed by any additional parameter values (if the method accepts parameters – this one doesn't).

Static methods are pretty common especially for system functionality in the base .NET libraries. Let's look at another example, that allows you to write to the Windows event log – which is a pretty complex process using Windows API. With .NET the code is much simpler and using wwDotnetBridge you can access the .NET component from FoxPro.

FoxPro – Writing two entries into the Windows Event Log

```
loBridge = CreateObject("wwDotNetBridge","V4")

lcSource = "FoxProEvents"
lcLogType = "Application"

IF !loBridge.Invokestaticmethod("System.Diagnostics.EventLog",;
                                "SourceExists",lcSource)
    loBridge.Invokestaticmethod("System.Diagnostics.EventLog",;
                                "CreateEventSource",;
                                Source,lcLogType)
ENDIF

*** Write out default message - Information
* public static void WriteEntry(string source, string message)
loBridge.Invokestaticmethod("System.Diagnostics.EventLog",;
                         "WriteEntry",lcSource,;
                         "Logging from FoxPro " + TRANSFORM(DATETIME()) )


*** Using COM Value to specifically cast the enum type
loValue = loBridge.CreateComValue()
loValue.SetEnum("System.Diagnostics.EventLogEntryType.Error")


* public static void WriteEntry(string source, string message,
EventLogEntryType type, int eventID)
loBridge.Invokestaticmethod("System.Diagnostics.EventLog",;
                            "WriteEntry",;
                            lcSource,;
                            "Logging error from FoxPro " +
                                     TRANSFORM(DATETIME()),;
```

```
                            loValue, 10 )
```

This is another example of system functionality using a bunch of static methods to access .NET functionality. The first method checks to see whether an event source exists with SourceExits() function. Event logs are created by a source that identifies the log by name and the type of log (Application, System, Security etc.) to write to. A process creates an event source with CreateEventSource() and then writes to it with WriteEntry(). Once an event source exists you can then call WriteEntry() to write to the source. This writes a default type of entry into the event log – which is an information entry.

The second log entry is a little more complex as I specify an EventLogEntryType of error rather than the default. This is an Enum value that needs to be passed. wwDotnetBridge includes some simple function to retrieve Enum values with GetEnumValue:

```
leValue = loBridge.GetEnumValue("System.Diagnostics.EventLogEntryType.Error")
? leEvalue  && 1
```

In most cases this simpler function works to pick up the enum value which can then be passed to any .NET methods that require the enum value. Here however it doesn't work because WriteEntry accepts many overloads and the integer value that GetEnumValue() returns doesn't map properly to the method overloads.

ComValue to provide 'real' .NET Values

wwDotnetBridge includes a .NET *ComValue* type (Westwind.WebConnection.ComValue) which has a value property. Methods on ComValue can then load up the value structure directly from within .NET without first marshaling the value to FoxPro. For our Enum value this means I can create an Enum value and directly store it on the value structure like this:

```
LOCAL loValue as Westwind.WebConnection.ComValue
loValue = loBridge.CreateComValue()
loValue.SetEnum("System.Diagnostics.EventLogEntryType.Error")
```

Now, when using InvokeStaticMethod() instead of passing the enum, I pass the ComValue() instance and wwDotnetBridge automatically picks up the loValue.Value property for the and uses that as the parameter:

```
loBridge.Invokestaticmethod("System.Diagnostics.EventLog",;
                            "WriteEntry",;
                            lcSource,;
                            "Logging error from FoxPro " +
                              TRANSFORM(DATETIME()),;
                            loValue, 10 )
```

This sounds a bit complex and it is, but fortunately it's rare that you have to resort to use ComValue. But it's nice to have this as a fallback and it demonstrates some of the power of wwDotnetBridge to avoid passing values into FoxPro to avoid COM type conversions. Other methods on ComValue can set the value from SetValueFromProperty(), SetValueFromInvokeMethod() and SetValueFromStaticProperty() as well as a number of type specific loaders that create .NET types from FoxPro values.

After all that we now have written two entries into the Windows event log.

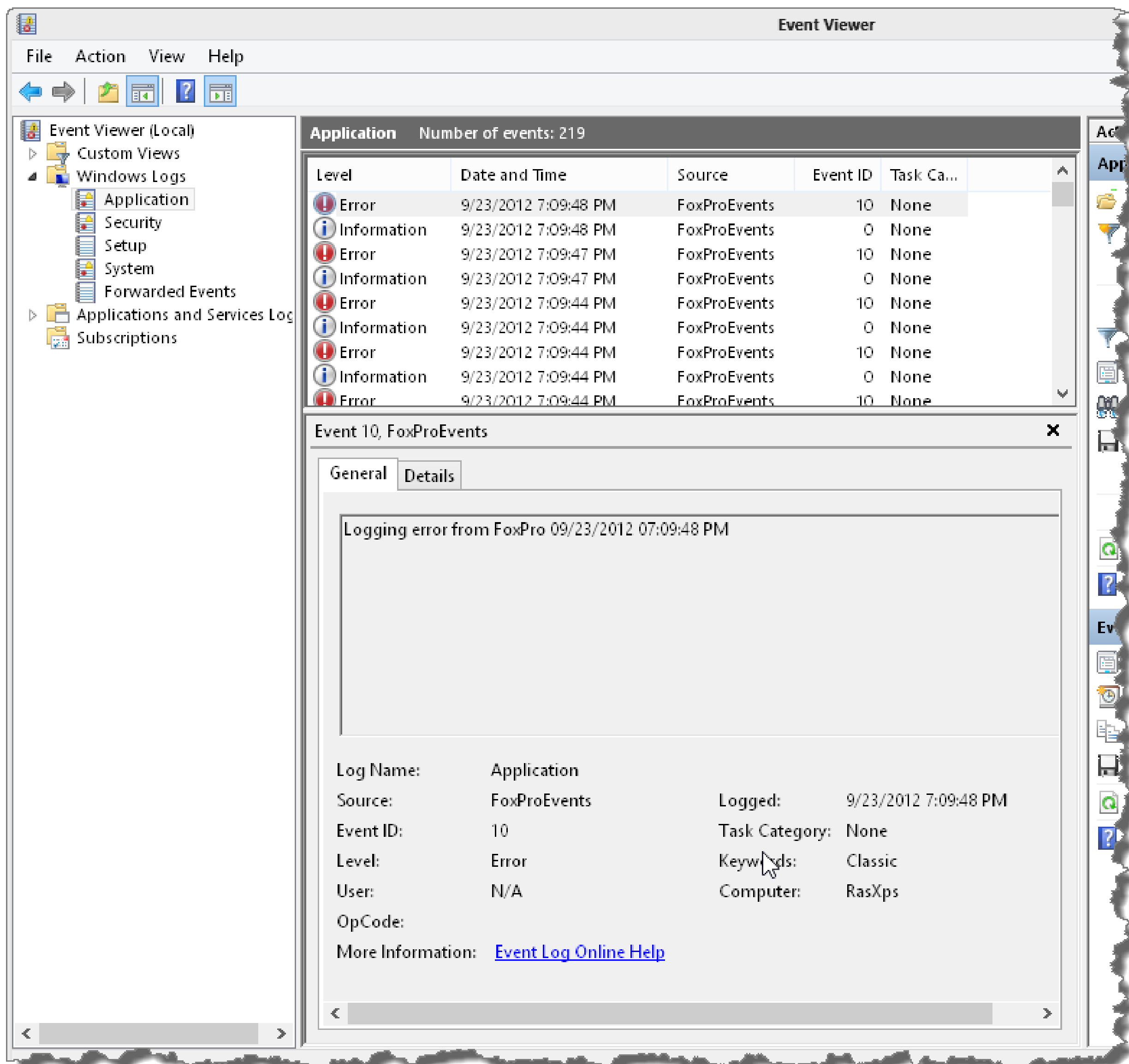

Figure 6 – *You can see the events written from our FoxPro code. The first entry is the error log type and has the Event ID set, the second entry just has the message and default icon.*

Just to complete this Eventlog theme, let's see how to list event log items as well:

```
*** Display Event Log Entries
loEventLog = loBridge.CreateInstance("System.Diagnostics.EventLog")
loEventLog.Source = lcSource
loEventLog.Log = "Application"

*** Turn Eventlog Entries into a ComArray Class
*** Indirect access automatically turns .NET array into ComArray
loEvents = loBridge.GetProperty(loEventLog,"Entries")
```

```
? "Entries: " + TRANSFORM(loEvents.Count)

lnTo =  MIN(loEvents.Count,10)
FOR lnX = loEvents.Count-1 TO loEvents.Count-lnTo STEP -1
      loEvent = loEvents.Item(lnX)
      ? loEvent.message
      ?
ENDFOR
```

Here the EventLog .NET type is created as an instance and we specify the source and log via its properties. Note that I can simply use these values directly. The Entries property however is an array and in order to make it easier to use it in FoxPro and keep the array in .NET I can call GetProperty() to retrieve the array as a ComArray instance. By doing so I gain the ability to easily manipulate the array and iterate over the array using its Item() method.

Finding .NET Type Signatures

At this point you might be thinking that this is all nice and neat, but how do you know how how do you find them in the first place?

Luckily you don't need to rely on documentation. There are a number of tools you can use to explore .NET assemblies and see what classes and members they contain. Here are a few of these tools are:

- Red Gate's .NET Reflector (Versions prior to 7 are free)
- IL Spy (open source)
- JetBrain's DotPeek (free)
- Telerik's JustCode (commercial)

Personally I still like Reflector best out of all of these, but any of them will do the trick (Reflector 6.5 is included in the Tools folder of the examples) so I use that here (and it's included in the samples in the Tools folder).

For example to look up the EventLog functionality we can look at that System.dll. In Reflector the core .NET framework assemblies are typically loaded by default and I can navigate to the System.dll and System.Diagnostics namespace and then to the EventLog class.

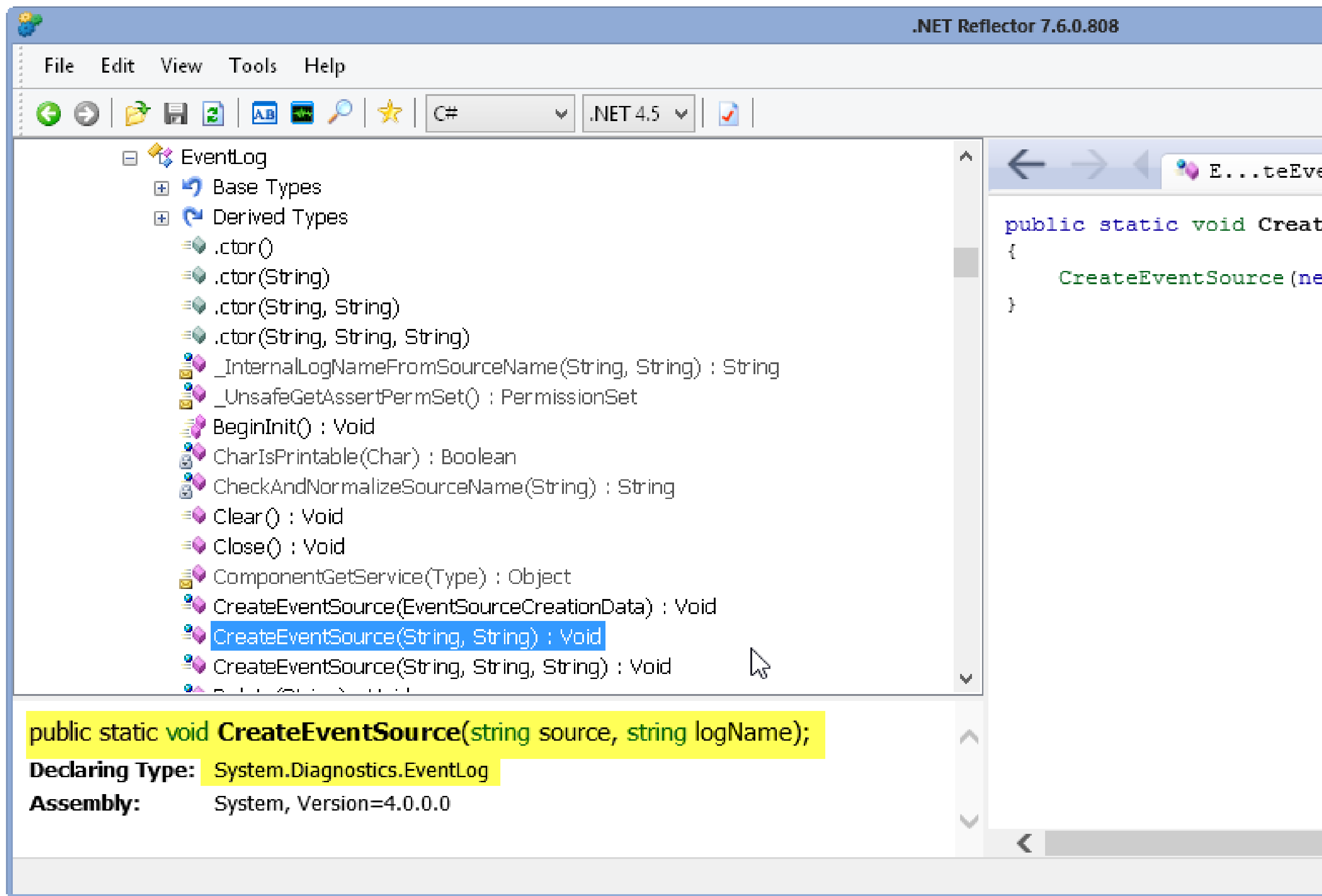

Figure 7 – *Reflector is a great tool to browse the .NET Framework libraries or any .NET assembly and see what functionality is available. Most code also is also viewable in decompiled mode, unless obfuscated explicitly.*

Reflector makes it easy to browse .NET assemblies and find the classes, methods or members you are looking for. The important things you typically need to know to invoke a .NET type are:

- The name of the type to create (or static instance name to access)
- The exact parameter signature for method calls

In Figure 5 you can see both of those things highlighted for the CreateEventSource method, which when invoked through wwDotenetBridge looks like this:

```
loBridge.Invokestaticmethod("System.Diagnostics.EventLog",;
                            "CreateEventSource",;
                            Source,lcLogType)
```

Invoking 3rd Party Components

One good reason to integrate with .NET is to invoke third party functionality from FoxPro code. There is a ton of functionality available for .NET both in terms of open source components and commercial products and you can tap this functionality easily from FoxPro with wwDotnetBridge. The process is pretty much the same as I showed above, except that you have to load the .NET assembly explicitly.

Whether it's your own .NET assemblies you've created or whether it's from some open source project, you can hook up and access external .NET code.

Here's another example: I'm going to use the OpenPop .NET POP3 library to access a POP3 mailbox and retrieve a list of emails. As we did with the EventLog class we can sneak a peek at the API of OpenPop with Reflector. Open Reflector and load Openpop.dll.

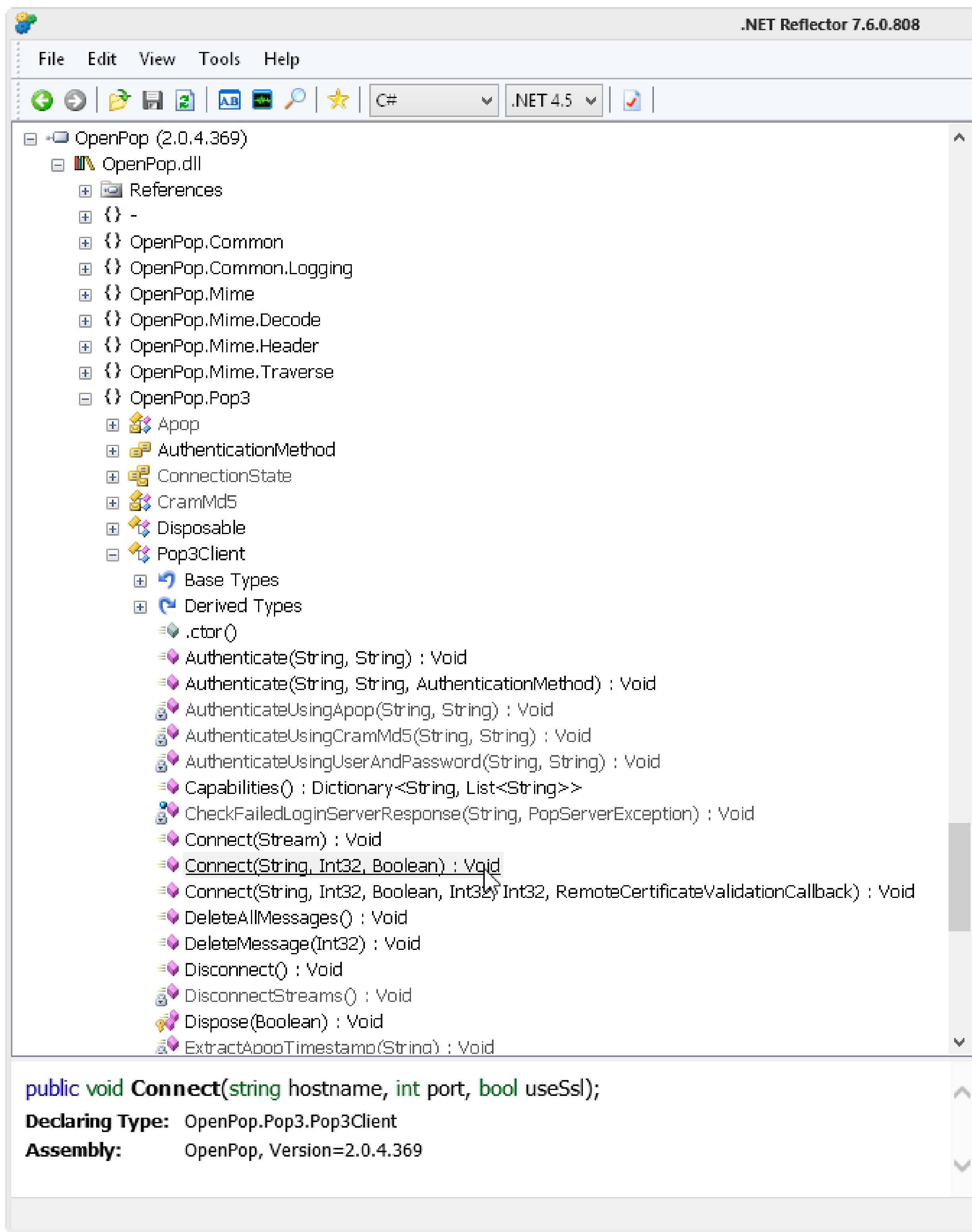

Figure 8 – *Checking out the OpenPop API in Reflector. It's a great way to discover APIs and see what you can access and what method signatures*

Looking at the API we can now loop through messages using this code:

FoxPro – Accessing a POP3 account with the OpenPop .NET Library

```
LOCAL loBridge as wwDotNetBridge

loBridge = CreateObject("wwDotNetBridge")

? loBridge.LoadAssembly("bin\OpenPop.dll")

loPop = loBridge.CreateInstance("OpenPop.Pop3.Pop3Client")


*** Connect overloads doesn't work directly
* loPop.Connect("mail.gorge.net",587,.f.)
? loBridge.InvokeMethod(loPop,"Connect","pop3.gorge.net",110,.f.)

? loPop.Authenticate("rstrahl",STRTRAN(GetSystemPassword(),"0",""))

lnCount =  loPop.GetMessageCount()
? StringFormat("{0} Messages",lnCount)

*** NOTE: OpenPop is 1 based because pop3 is 1 based!
** show last messages
FOR lnX = lnCount TO 1 STEP -1
   loHeader = loPop.GetMessageHeaders(lnx)
   ? loHeader.From.DisplayName
   ? "   " + loHeader.Subject
   ?
   IF lnX < lnCount - 10
      EXIT
   ENDIF
ENDFOR

loPop.Disconnect()
```

This code loops through last 10 messages in your POP3 inbox and displays the sender and message subject.

This code explicitly has to call LoadAssembly() on the OpenPop library by specifying it's location on disk. Provide a relative path or a full path to the DLL. You can also load assemblies out of the Global Assembly Cache (GAC) by using it's fully qualified assembly name (which you can also find in Reflector on the class node) and using the *LoadAssembly()* method. For example, here is a LoadAssembly call to load the System.Web.Extension assembly which is registered in the GAC:

```
loBridge.LoadAssembly("System.Web.Extensions, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35")
```

The instance is then created using the full name of the class which you can find in Reflector when you navigate to the class or any member of the class.

Next to open the connection I have to use InvokeMethod() on the Connect() method of Pop3Client. The direct call does not work because the method is heavily overloaded and .NET doesn't see the FoxPro numeric parameter as an integer but rather as a double. FoxPro numbers typically are passed as doubles to .NET which can make method calls fail. Using the intermediate InvokeMethod() call and Reflection causes some additional type conversion to occur in .NET that makes the call work properly.

Once connected the rest of the code just accesses the OpenPop API directly without further indirect calls to wwDotnetBridge – the COM instance has simple parameter types that just work on their own in FoxPro.

Best practice is to try using direct access first, and if that doesn't work or doesn't produce the expected result, *then* use wwDotnetBridge's methods and helpers.

For completeness sake, here some additional OpenPop code to retrieve an individual message and pull out the content. This process is a bit involved as it needs to deal with different kinds of content (Html,plain, attachments etc.).

```
*** Find the last message by count (after listing) and display
loMsg = loPop.GetMessage(lnCount)
loMsg.Headers.Subject

loPart = loMsg.FindFirstHtmlVersion()
IF ISNULL(loPart)
   loPart =loMsg.FindFirstPlainTextVersion()
ENDIF

IF !ISNULL(loPart)
   ? StringFormat("Is Text: {0}",loPart.IsText)
   ShowHtml( loPart.GetBodyAsText() )
ENDIF
```

Creating your own .NET Wrappers

Sometimes accessing .NET code directly from FoxPro can become pretty tedious as you have to figure out the API in FoxPro and Reflector, and you use trial and error to see what works and what doesn't. If APIs are straight forward and not terribly complex it might be OK to use FoxPro, but if you are dealing with a lot of complex structures it might actually be quite a bit easier to create a wrapper of some .NET component and then call this custom front end component from FoxPro.

Creating .NET Wrappers to abstract complex .NET Functionality

Keeping in the spirit of Email I created a .NET wwSmtp component that wraps the System.Net.SmtpClient classes' functionality. In fact I expose wwSmtp in our West Wind Web Connction and West Wind Internet and Client Tools products with a FoxPro wwSmtp class. While I could have directly implemented the wwSmtp class by using wwDotnetBridge to call the native SmtpClient MailMessage APIs in .NET, I opted to creating a .NET class that wraps the behavior and provides a host of helpers. This has two advantages – it's much easier to develop the class in .NET (if you know some .NET) as Visual Studio gives you full

Intellisense to discover how APIs work. The wrapper .NET type then exposes an interface that FoxPro COM friendly and abstracts the interface to make it as easy as possible to use – and in my case to mimic the exiting email functionality already provided in Web Connection through a C++ component. Finally the .NET component – when done – can be used both in FoxPro and .NET.

You can check out the source code of the wwSmtp.cs wrapper for the SmtpClient class in the provided samples if you like as it's fairly sizable and doesn't fit here.

Here's the code to actually access that class using wwDotnetBridge:

FoxPro – Using the wwSmtp .NET class to send an email

```
loBridge = CreateObject("wwDotNetBridge","V4")

loBridge.LoadAssembly("InteropExamples.dll")
loBridge.cErRORMSG

LOCAL loSmtp as Westwind.wwSmtp
loSmtp = loBridge.CreateInstance("Westwind.wwSmtp")

*loSmtp.AddAttachment_3("c:\sailbig.jpg")
loBridge.InvokeMethod(loSmtp,"AddAttachment","c:\sailbig.jpg")

loSmtp.MailServer = "smtp.server.com:587"
loSmtp.UseSsl = .T.

loSmtp.Username = "user"
loSmtp.Password = "secret"

loSmtp.Recipient = "Rick Strahl<rstrahl@west-wind.com>"
loSmtp.SenderEmail = "admin@west-wind.com"

loSmtp.Subject = "Test Message"
TEXT TO loSmtp.Message NOSHOW
<html>
      <head>
            <style>
                  body { font-family: Verdana; background: cornsilk; }
            </style>
      </head>
<body>
<p>
Hello Rick,
</p>
<p>
This is a test message from <b>Southwest Fox</b>
</p>

<p>
Enjoy,
</p>
<p>
+++ Rick ---
```

```
</p>
</body>
</html>
ENDTEXT

loSmtp.ContentType = "text/html"

IF (!loSmtp.SendMail())
 ? loSmtp.ErrorMessage
ENDIF

? "Mail sent"
```

As you can see the component is loaded with wwDotnetBridge, but once loaded there's not much use of wwDotnetBridge's methods. That's because wwSmtp abstracts the SmtpClient() component and simplifies it and explicitly makes it easy to access with simple property values and simple method calls.

The one wwDotnetBridge call on the instance:

```
*loSmtp.AddAttachment_3("c:\sailbig.jpg")
loBridge.InvokeMethod(loSmtp,"AddAttachment","c:\sailbig.jpg")
```

which is necessary because the method is overloaded which otherwise wouldn't work.

Creating a .NET wrappers to simplify complex .NET APIs for use in FoxPro is a great way to expose complex .NET functionality to FoxPro in simpler more abstract ways. While wwDotnetBridge allows accessing most .NET functionality generally it's not the best of ideas to create reams and reams of .NET access code. wwDotnetBridge code can be verbose and often requires some trial and error to get the .NET calls right. If you are at all familiar with .NET it's going to be much easier to create .NET code to access complex functionality within .NET by taking advantage of the C# compiler and Intellisense.

Event Handling for COM Interop

Another feature of the wwSmtp class and one of the reasons I created it, is to support asynchronous sending of emails. You can fire an email and not have to wait for completion using

```
? loSmtp.SendMailAsync()
```

instead of the Send() call. This works great and the .NET wwSmtp class takes care of spinning up a new thread and running the operation asynchronously in .NET which is as easy as this in .NET:

```
public void SendMailAsync()
{
    Thread mailThread = new Thread(this.SendMailRun);
    mailThread.Start();
}
```

```
protected void SendMailRun()
{
    // Create an new reference to insure GC doesn't collect
    // the reference from the caller
    wwSmtp Email = this;
    Email.SendMail();
}
```

Bingo – you've got multi-threaded code.

With multi-threaded code frequently come events. Events are tricky in COM Interop and this is one place where wwDotnetBridge can't help. In fact, if you want to handle events from .NET components you *have to register your .NET Components with COM in order for the event interface to be available*. COM Events require COM for FoxPro to sink them, so if you create components that require event handling COM registration with RegAsm unfortunately is unavoidable.

The first thing to understand about .NET events and COM is that COM events must be explicitly set up. There's no way to hook up arbitrary .NET Events and have them handled over COM, if the events weren't explicitly published to COM. This means .NET event handling over COM is pretty much limited to components that you create yourself and publish the events yourself.

To create a COM component that handles events two things are required:

- A class declaration that links the class to an Event Interface
- An interface definition that exposes each event as a method

Let's take a look how this works for wwSmtp. The first step is the class declaration which looks like this:

```
[ComVisible(true)]
[ClassInterface(ClassInterfaceType.AutoDual)]
[ComSourceInterfaces(typeof(IwwSmtpEvents))]
[ProgId("Westwind.wwSmtp")]
public class wwSmtp   : IDisposable
```

The event specific attribute is the ComSourceInterfaces attribute which describes the event interface that will expose events to COM. The interface then needs to declare each of the events exported in the interface definition.

The wwSmtp class contains two event declarations:

```
public event delSmtpNativeEvent SendComplete;
public event delSmtpNativeEvent SendError;

public delegate void delSmtpNativeEvent(wwSmtp Smtp);
```

The delegate is the function definition that defines the event signature – this event fires with a wwSmtp object instance as a parameter. When called from .NET code you can directly hook up both of these events.

But over COM wwSmtp has to be registered and expose the event interface that maps these to events. An interface is just a declaration, not an implementation, so the class is very simple:

```
[ComVisible(true)]
[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
public interface IwwSmtpEvents
{
    [DispId(1)] void SendComplete(wwSmtp smtp);
    [DispId(2)] void SendError(wwSmtp smtp);
}
```

The class simply maps the the two event handlers on the wwSmtp class. When the C# compiler compiles this code it automatically the necessary COM Event interfaces that are required in order for COM events to fire.

In FoxPro you need to capture these events by implementing an event interface. This COM interface has to be OLEPUBLIC and effectively inherits from the COM Interface. This is the reason why the component has to be registered.

```
DEFINE CLASS wwSmtpEvents AS session OLEPUBLIC
      IMPLEMENTS IwwSmtpEvents IN "Westwind.wwSmtp"

      PROCEDURE IwwSmtpEvents_SendError(smtp AS VARIANT) AS VOID
      ? "Sending message '" + smtp.Subject + "' failed..." +;
              smtp.ErrorMessage
      ENDPROC

      PROCEDURE IwwSmtpEvents_SendComplete(smtp AS VARIANT) AS VOID
      ? "Sending message '" + smtp.Subject + "' complete..."
      ENDPROC

ENDDEFINE
```

Each of the methods in the event class matches the methods of the .NET COM interface exported by .NET.

To bind the Event Interface to the COM class you use the Foxpro EVENTHANDLER() function that binds the source object and the event interfaces together:

```
loEvents = CREATEOBJECT("wwSmtpEvents")
EVENTHANDLER(loSmtp,loEvents)

? loSmtp.SendMailAsync()
? "Code is done executing..."
```

With this code in place change run the code again. You should now see the mail request run and immediately print *Code is done executing* to the screen, followed shortly after by either a mail sent completion message or an error message if the send failed.

You can automate creating this interface for you by using the FoxPro Object Browser and selecting the .NET COM .TLB file and selecting the Event interface in the object browser, then dragging and dropping the interface into a FoxPro PRG file. When you do you get an empty class auto-generated for you.

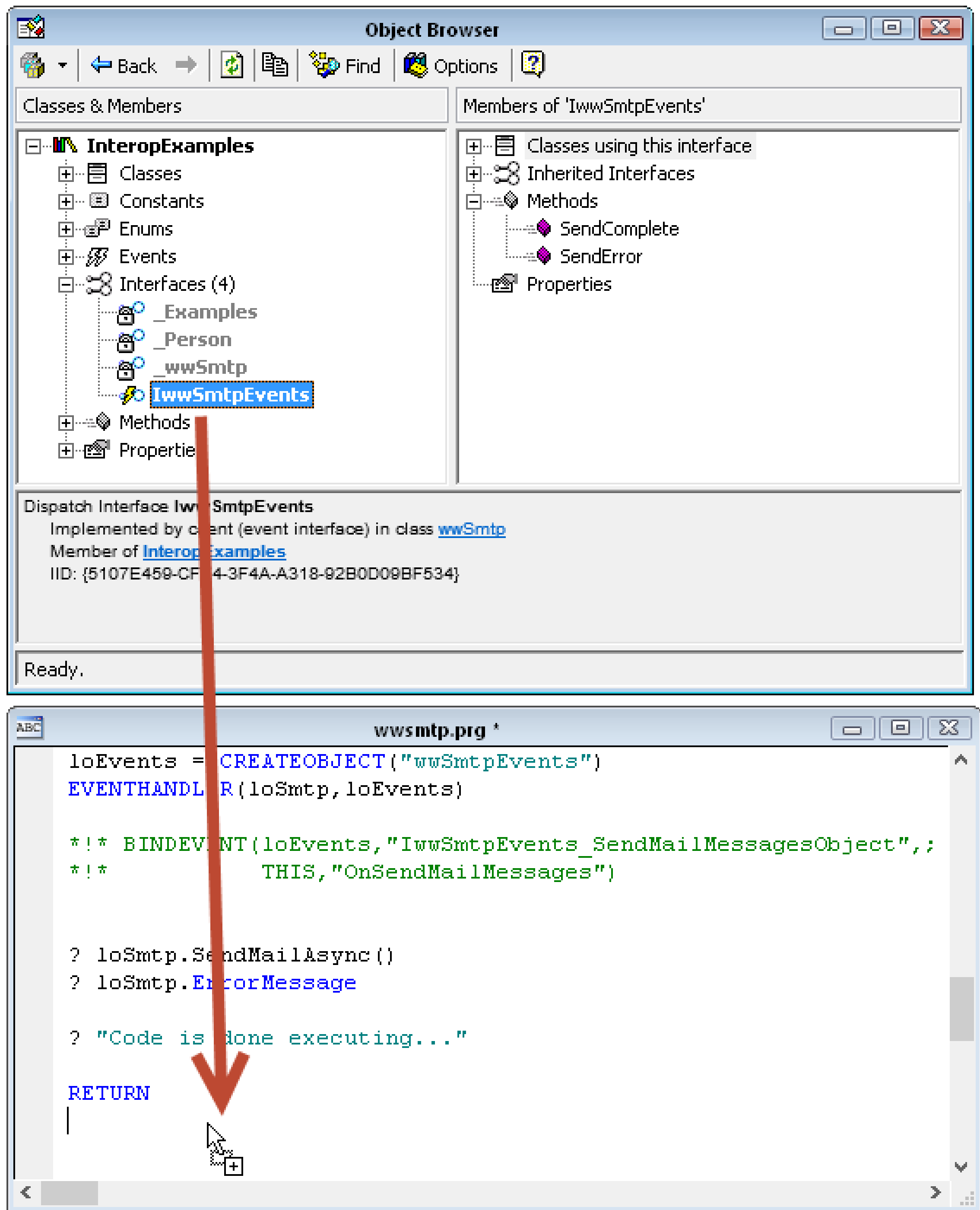

Figure 9 – *Dragging and dropping an Event Interface into a FoxPro PRG from the Object browser creates the COM event interface*.

wwDotnetBridge Object Fixups

You've already seen that it's pretty easy to pass values from .NET to FoxPro. Most simple types, objects, and even arrays work pretty much straight out of the box. For a few other problematic types ComArray and ComValue can help.

Let's look at a few special types and how they are handled. Let's start with arrays because they are a common source of pain in COM Interop. Let's go back to our InteropExamples project we used earlier and add a few methods to the Examples class.

Arrays and ComArray

We briefly looked at the ComArray class earlier when we looked at the Persons example. We had a GetPersons() method, let's add an AcceptPersons method to the class that accepts an array of persons as input.

```
public Person[] GetPersons()
{
    return Persons.ToArray();
}

public bool AcceptPersons(Person[] persons)
{
    Persons.Clear();
    Persons.AddRange(persons);
    return true;
}
```

Working with arrays that have to be passed over COM is difficult because COM marshaling destroys the original .NET type and results in a FoxPro array instead. The FoxPro array can not easily be sent back to .NET because it's lost its dotnet-ness. But with the ComArray implementation this is actually fairly easy to do.

Let's create some code to GetPersons() to FoxPro, then add a new person to the array and ship the entire array back to .NET.

FoxPro – Passing an array between FoxPro and .NET

```
loPersons = loBridge.Invokemethod(loFox,"GetPersons")   && ComArray

? StringFormat("Initial Array Size: {0}", loPersons.Count)

*** Create a new Person
loNewPerson = loPersons.CreateItem()

loNewPerson.FirstName = "Billy"
loNewPerson.LastName = "Nobody"
loNewPerson.Entered = DATETIME()
```

```
loNewPerson.Address.Street = "121 Nowhere lane"

*** Add the person to the array
loPersons.AddItem(loNewPerson)

? StringFormat("After add array Size: {0}", loPersons.Count)

*** Pass the array back to .NET
loBridge.Invokemethod(loFox,"AcceptPersons",loPersons)

 ? StringFormat(".NET Persons Array Size after update: {0}",
loBridge.GetPropertyEx(loFox,"Persons.Count"))
```

The code starts by retrieving a list of Persons and retrieving that result as a ComArray instance. If you recall ComArray is a COM wrapper around a .NET array that has an instance property that holds the .NET array and a bunch of methods that can manipulate the array.

We note the count of the array which starts off at 2 persons from our previous demo. I then call loPersons.CreateItem() which creates a new Person object. CreateItem creates a new .NET instance of the element type of the array – in this case a Person object. The object is populated with values and the added to the loPersons ComArray with the AddItem() method.

Finally we call AcceptPersons and pass the ComArray instance to the .NET method. wwDotnetBridge automatically fixes up the ComArray and internally passes the instance variable to the server. This magic works only on the indirect methods (InvokeMethod in this case) – it doesn't work with direct access. If you try to call:

```
? loFox.AcceptPersons(loPersons)
```

You'll get an error because the target method does not accept a ComArray object. You also can't do:

```
loFox.AcceptPersons(loPersons.Instance)
```

which doesn't work, because as soon as you reference the Instance it's converted into a FoxPro array and can no longer be passed back to .NET. InvokeMethod() is required to make this work.

Enumerable .NET Types and ComArray

.NET has an IEnumerable interface that is used to present enumerable collections. IEnumerable is used for ForEach iteration, but it's also a meachism that's used to represent collection data in an incremental way. Rather than loading the data into a datastructure the data is read and provided one item at a time which can be very efficient.

However there's no corresponding COM interface for IEnumerable and so enumerable types that are loaded one at a time fail with a resource error.

The ComArray class has a nifty helper however that can help in some scenarios via the FromEnumerable() method. An example of this is the Persons member on the example class which is defined as a generic type

```
public List<Person> Persons { get; set; }
```

which cannot be accessed directly from FoxPro. However, List<t> implements IEnumerable and as part of IEnumerable you turn an enumerable into an array with ToArray(). By doing so you can receive the result in FoxPro.

```
loPersonArray= loBridge.CreateArray()
loPersonArray.FromEnumerable(loFox.Persons)
```

and now you can fire away at the ComArray stored in loPersonArray.

DataSet Conversions

wwDotnetBridge also includes some dataset conversion routines that make it very easy to accept and pass back dataset and cursor results.

If you have a method that returns a .NET DataSet:

```
public DataSet GetWebLogEntries()
{
    var sql = new Westwind.Utilities.SqlDataAccess(
      "server=.;database=Weblog;integrated security=true");

    return sql.ExecuteDataSet("TWebLog",
      @"select top 20 * from blog_entries
        where entrytype=@entryType
        order by entered Desc",
        sql.CreateParameter("@entryType",1));
}
```

This code uses the SqlDataAccess class in Westwind.Utilities.dll from my .NET toolkit to execute a query against a SQL Server and returns the result as a DataSet.

You can now receive that dataset in Foxpro and turn it easily into one or more cursors (if the dataset contains multiple tables):

```
loDS = loFox.GetWebLogEntries()
loBridge.DataSetToCursors(loDS)
SELE TWEBLOG
BROWSE
```

If you prefer you can also go to an XmlAdapter instead of cursors, so you can manipulate the data or control which tables are turned into cursors using the DataSetToXmlAdapter() method.

To send a FoxPro cursor to .NET you can use the following:

```
CREATE CURSOR TPersons (FirstName c(30), LastName c(30),Entered T)

INSERT INTO TPersons (FirstName,LastName,Entered) VALUES
                     ("Rick","Strahl",DATETIME())
INSERT INTO TPersons (FirstName,LastName,Entered) VALUES
                     ("Markus","Egger",DATETIME())
BROWSE

loDs = loBridge.CursorToDataSet("TPersons")
? loFox.AcceptDataSet(loDs)
```

wwDotnetBridge at West Wind Technologies

I've been using wwDotnetBridge in several applications for several years and it's been a life saver for extending the lease on life for FoxPro apps for me. It's allowed me to integrate with .NET with several applications in a way that otherwise would have been difficult or not possible at all.

West Wind Html Help Builder

Html Help Builder is a help and documentation generation tool that makes it easy to build help files, online documentation and Word output for developer and end user or other technical documentation.

Help Builder supports importing of .NET classes and assemblies and auto-documenting them based on the meta-data contained in .NET DLL files. It allows me to provide fairly complete documentation from .NET components. Internally Help Builder uses .NET Reflection to retrieve the type import information.

To make this work I built a custom .NET component that handles all the type import information, creates a FoxPro friendly .NET structures that are then passed back to FoxPro for easy parsing and outputting into help documentation. There's a lot of logic in this wrapper - it parses types, handles merging data from XML documentation files and walking the type hierarchy, fixing up type references for auto-linking and much more. It's a complex piece of code, but the FoxPro interface to it is very simple that only looks at the final results in the form of a class with properties and arrays and no methods.

Help Builder also handles Visual Studio RTF text imports via bit of .NET code I dug up some time ago and integrated into Help Builder NET helper assembly. There's some fairly tricky .NET code that fixes up the HTML text in .NET and in the end the FoxPro interface is a single method call on the class – all the logic is abstracted inside of .NET.

Help Builder can also import existing CHM files and it uses a .NET library called Html Agility Pack to do this. Agility pack has the ability to read the CHM help file storage format and retrieve the individual help files and since it is a lightweight HTML DOM parser allows me to parse the resulting HTML documents including picking out image and resource links and pulling those resources from the old help file. Again I'm using a small .NET wrapper to perform the automation of Html Agility Pack and then call that code from FoxPro.

All three of these scenarios are quite crucial to the feature set in Help Builder and the .NET integration makes it possible. In all three cases I used wrapper classes to abstract the .NET functionality I needed into something that's much easier to use in FoxPro.

West Wind Web Service Proxy Generator

The Web Service Proxy Generator is a tool for dynamically creating FoxPro proxies to SOAP Web Services using .NET as an intermediary. It uses a Wizard interface to import a Web Service and create a .NET proxy and FoxPro class that accesses the .NET proxy. wwDotnetBridge is a crucial part of this product as it is used to access the generated .NET proxy from FoxPro without COM registration. It's used both in the Wizard to handle the compilation of the .NET proxy classes generated by .NET tools and more importantly by the generated proxy classes. The FoxPro class makes passthrough calls to the .NET Web Service proxy class.

wwDotnetBridge also is crucial in allowing access to the myriad of types that come back from Web Services, especially arrays which are extremely common for Web Services. Some of the most complex wwDotnetBridge .NET access scenarios I've seen have involved deep Web Services result hierarchies. Not surprisingly this tool has spurred many, many enhancements in wwDotnetBridge as different user problems have allowed me to capture many common use cases and abstract them.

Summary

COM Interop may be one of the most underused extensibility features for Visual FoxPro. .NET offers tons of functionality that is available for the taking with lots of built-in functionality in the .NET framework itself, 3rd party libraries and a wide selection of open-source libraries available. wwDotnetBridge opens up most of this .NET market for access from FoxPro with only a minimal level of complexity.

COM Interop has a number of limitations, but wwDotnetBridge provides a lot of helper functionality that lets you access most features of .NET directly.

wwDotnetBridge is now free and open source and the code is available online. It's also part of West Wind Web Connection and the West Wind Internet Client Toolkit where it is part of an officially supported set of tools.

Resources

- **wwDotnetBridge Home Page**
- **Samples and Source Code**
- **wwDotnetBridge on GitHub**
- **Using .NET COM Components from Visual FoxPro**
- **Open Pop**
- **Red Gate's .NET Reflector**