VFP COM Interoperability with VB.NET

By Chuck Urwiler (MCT, MCSD, MCDBA), Senior Developer, EPS Software Corporation

Abstract

This whitepaper discusses how you can use interoperability between Microsoft Visual FoxPro (VFP) and Visual Basic .NET.

Contents

Overview

In this paper, you learn about how VFP and .NET can "talk to each other." While .NET is drastically different from the world of COM, Microsoft recognizes that people can't drop all of their current investment in the Component Object Model (COM) for .NET. As you probably know, VFP is not part of the .NET Framework, but it has been a strong player in the world of COM for the past few years.

This paper assumes that you are not familiar with the internals of COM or with the details of .NET. Further, it assumes that you have a desire or need to take advantage of .NET but do not have the time to rewrite your applications with it. Finally, all .NET examples are presented using VB.NET and all COM examples are presented using VFP.

COM Interop

The key component that enables VFP to interoperate with .NET is a feature called COM Interop. COM Interop is built into the .NET Framework, so you do not need any additional downloads or setup routines in order to use it. Also, COM Interop is not a one-way street—it enables you to use .NET components from COM and to use COM components from .NET. You will see examples of each form of Interop later in this paper.

However, before discussing the details of Interop, some background information on COM and .NET is in order. Why is it that.NET and COM can't talk to each other without using some special feature?

What is COM?

For those of you who are familiar with how COM works, you can skip this section. However, if you're like many developers, you may not have done very much with COM or have not learned much about the underlying details of this technology. This section attempts to clarify the operation of COM.

Simply put, COM exists to allow communication between components. A similar concept was first introduced in version 1.0 of OLE when it was released back in the days of Windows 3.0. The goal of OLE 1.0 was to allow applications to expose their functionality to other applications so that users didn't have to buy a product that contained every feature they needed, they could just use a mixture of applications that contained their desired features. Unfortunately, OLE 1.0 wasn't very good at sharing application features, so Microsoft went back to the drawing board to create OLE 2.0, which spawned the concept of COM.

Until .NET, COM has been Microsoft's approach to multi-tier development as well as for application sharing. One key piece of COM is that it enables a developer to reuse the individual features from other applications in their own development tasks. For example, a COM object could be used to communicate over a modem to a credit card authorization system. If this COM object is installed on a user's system, any application on that system can use it to validate credit card numbers.

An important point here is that this COM object does not need to be written in the host language. As long as it was built in accordance with the COM standard pieces in place, any COM-based client application could consume it. This is what allows a VFP developer to use a Visual Basic-built COM component, or vice-versa—the fact that every COM component has a standard way to share their available services.

To understand how COM works, and how it compares with .NET, you have to go “under the hood” and see what happens when a COM server is called by a COM client.

First, you have to identify the COM component’s ProgID, which is a registry entry that points to the specifics of the component. For example, the credit card object mentioned above might have a ProgID of `MyServer.CreditCardSvcs`. In VFP, you can instantiate this COM server with the following code:

```
loCredit = CreateObject("MyServer.CreditCardSvcs")
```

Now, you can use this COM component as if it were a VFP object, like this:

```
llSuccess = loCredit.ValidateCard('Visa','4111111111111111')
```

However, this masks some of what really happens behind the scenes. How does VFP know whether the ValidateCard() method even exists on the COM server? What about the type and number of the parameters? How does VFP get the return value from the COM server? And, how are any errors handled? All of these questions are answered by the COM standard.

First, since the COM component can be simultaneously invoked by multiple client applications, there has to be a mechanism in place to allow the component to be released from memory once all clients have finished using it. In COM, this is done by reference counting (ref count for short), where instantiation increments the ref count by one, and a release decreases the ref count. When the ref count reaches zero, note that the COM *client* must take care to release the COM component from memory.

A COM client only knows that the ValidateCard() method exists because any COM server can be asked whether a method exists or not. This is handled is through the IUnknown and IDispatch interfaces that exist on every COM server. If you are not familiar with interfaces, just think of them as a set of methods and properties. The IUnknown interface exists to allow any COM client to determine what other interfaces exist on the COM server, as it contains a method called QueryInterface. IDispatch dynamically calls any method that exists on the COM server and allows a language like VFP to call the method without prior knowledge of the method. Other interfaces most likely exist on the COM server as well, with each interface exposing a set of methods and properties. Note that these interfaces are not accessible to a VFP developer’s code (you wouldn’t want to have to deal with a COM server at this level anyway).

The type and number of parameters for any method can be found inside of a separate construct called a type library. Type libraries are binary files that provide information about the component, any sub components, interfaces, methods, properties, arguments, structures, and return values. However, type libraries are not always in a separate file—they can be embedded within a COM server as well.

Errors in COM are reported primarily through codes called HResults, which are basically coded values. HResults can report success or failure as well as error information and the source of the error.

Now, fortunately for VFP (as well as VB) developers, you do not need to know these details in order to use COM servers. Instead, all of these details are handled behind the scenes for you.

For example, if you call a method from VFP, it takes care of determining whether the method exists, what its parameters are, converting data types to and from the COM server, and handling any errors that occur.

As you might suspect, not all is rosy with COM. For example, one issue with COM components is versioning. Imagine that you purchased the example credit card COM server used above. Now, you get an email saying a new version is available. What could happen with the new version? In some cases, older COM servers are overwritten by the newer version, which can cause a lot of problems. This has become known as "DLL Hell," where installing a new application overwrites older DLLs and breaks functionality throughout a user's system.

Another issue is that COM interfaces are supposed to be immutable, meaning that the interface should never change (that is, the methods and properties on that interface should always be available and should always take the same number of parameters, etc.). Unfortunately, there is no enforcement for this rule, allowing many COM server developers to break this rule.

A problem that is becoming more prevalent is the issue of type safety. COM does not enforce type safety, meaning that there are no checks done on the values passed into or out of the component. This means that it is relatively simple to break a COM component, or worse, to run malicious code on a user's system.

Installing COM servers can also be problematic, as every installed COM server requires information to be added to the registry. As more and more COM components are created and installed, the registry on the computer grows. It is not uncommon these days to have a registry of 15 to 20MB in size, or even more.

As you may have already suspected, these downfalls to COM have been solved in .NET. However, by solving these issues, .NET programs are not directly compatible with the COM standard. Regardless, let's take a closer look at what .NET is all about.

What is .NET?

You've heard of it, but what exactly is .NET? Yes, it's a new breed of tools that developers can use to build the latest and greatest applications for the Windows platform. It also is a tool partially designed to fix what's "wrong" with COM.

Many people mistake .NET as an Internet-only product. In fact, .NET is usable for nearly any type of application or service that you can think of, such as console applications, rich-client Windows applications, system services, Web applications, middle-tier components, and, of course, XML Web services.

One of the key parts of .NET is its runtime component, known as the Common Language Runtime (CLR). The name basically says it all. With a single runtime, any .NET language can be executed. This means that there's no longer a separate DLL for the VB runtime and the VC runtime—all .NET languages use the same runtime.

This has many positive effects. For example, now it is possible to inherit features from components written in any .NET language. This means a component written in C# can be subclassed by a VB.NET application. It also means that these two languages have nearly the same features, since they both use the same runtime.

Microsoft accomplished a single runtime by creating something called the Microsoft Intermediate Language (MSIL). Any .NET language compiler must generate MSIL, which is then executed by the runtime. This runtime, as it only runs MSIL, performs type safety checks on any data types passed as parameters or return values, creating much more secure components. Further, as all languages are essentially the same, no data type conversion is necessary, unlike the conversion necessary in the world of COM.

To make things even easier in .NET, all of the features exposed by the runtime (including operating system features such as services, performance counters, etc.) are neatly wrapped into something called the Unified Class Library. This library is the same for all .NET languages (thus the term "unified'), so any developer using .NET can take advantage of any feature provided by these classes. This class library is organized in a hierarchical fashion and grouped by namespaces. If you're not familiar with namespaces in .NET, think of them as a hierarchical naming convention for classes that perform similar functions.

As a replacement for type libraries, ProgIDs and other COM installation hassles, .NET includes metadata right inside the finished component. Recall that COM requires several calls to the component to determine whether a method exists, and if so, its signature. .NET handles this by allowing a client program to read what it needs to know from the metadata within the component. This greatly simplifies deployment (that is, you can just copy files without registration) and eliminates the need to have IUnknown or a type library to discover what's available.

One more point about .NET components: they are not ref counted like COM servers. Instead of a ref count, the .NET runtime handles the lifetime of a component. This means that neither the consumer nor server of a .NET component is responsible for releasing the component—the runtime handles it automatically.

At this point, you may already see how the COM concept of "DLL Hell" has been eliminated by .NET. Since a .NET component stores all necessary metadata, versioning is also no longer a problem. In fact, .NET allows different versions of a component to live side-by-side on the same machine with no conflicts (as long as something called a "strong name" is used for the component). How can this be? The metadata includes version information and any calling code can be instructed to use a specific version.

Errors in .NET are all handled the same way, by using a class called the exception class. When an error occurs, an exception object is created and passed to the caller. Since all languages can use the same classes, every language understands this exception object and can deal with it in a very robust manner. This beats the HResults that COM servers use to report errors, since they are typically a cryptic value that must be converted into human-readable messages.

What's the problem?

Now that you've seen the difference between .NET and COM, you may already have figured out why they can't talk to each other. Instead of COM interfaces, .NET uses objects directly. Type libraries have been replaced by the metadata included in the assembly (the term for a .NET component or application). Errors are returned differently in .NET as compared to COM.

All of these issues require that something mediate between the world of COM and the world of .NET. This is where the COM Interop feature comes into play. Essentially, there is a set of .NET classes designed specifically for performing COM Interop, and if you use Visual Studio.NET

(you don't need VS.NET to build .NET applications), much of the grunt work is done for you. You'll see examples of this later in the paper.

By adding the Interop functionality to .NET, Microsoft has recognized the need for people to migrate to .NET without re-engineering entire applications from the ground up. There is a significant investment in COM by many who have been using Microsoft's dev tools, and much of this code cannot change (such as, no source code available or the original developer is no longer available, etc.). This feature protects that investment.

More importantly, Interop allows what's known as an Incremental Migration strategy, where you only "upgrade" a tiny part of an application to a new environment before trying another part of the same application. I'm willing to bet that a good number of VFP developers are familiar with this process for taking a FoxPro 2.x application and moving it to VFP. Regardless of whether I'd win that bet, anyone who has tried to migrate an app from one major version to another would tell you that incremental migration is the way to go, as it limits the number of potential problems that would need to be solved at a single time.

Building a .NET Component

In order to illustrate how Interop works, start by building a .NET component within Visual Studio .NET. Create a new project by clicking **New Project** on the Start page. In the New Project dialog, select the VB Projects folder, and then locate the Class Library project template. Choose your directory, give the project a name, and click **OK** to allow VS.NET to create the new solution.

In the code window that appears, add your code for your class. For example, you might create a component that retrieves data from a SQL Server database, and that you've returned this SQL data as an XML stream. As a simple test, however, you may just want to return a simple "Hello, world!" string.

Once the coding is complete, you won't be able to test the component directly. If you've built any classes in VFP, you're used to this restriction. Instead, you must add another project to your solution to use as a test for the class. To do this, simply right-click the solution in the Solution Explorer window, choose the Add submenu, and then click **Add Project**. Most likely, you'll want to create a Windows or ASP.NET application for testing purposes.

The next step is most important, because without referencing your .NET class properly, the new project will not recognize it. In the Solution Explorer window, right-click the References node that lies directly underneath your test project (not your class library project) and select **Add Reference**. This displays a dialog where you can choose from a variety of components to reference in the selected project. Since you want to "see" the class library project that is in the current solution, click the **Projects** tab at the top of the dialog. Your class library should appear in the list. Select your class library and then click **Select**—now you should see your class library in the bottom list. Click **OK** to set the reference in the project.

Now that you've set your reference, try to write some test code that calls the methods and properties of your .NET component. For example, if you named your class library file "ClassLib1.vb" and your class is named "Customers," you can type this code to create an instance in your test project:

```
Dim oBizObj as New ClassLib1.Customers()
```

If you have done everything correctly, you will not see any “squigglies” under the name of the class name. Now you are able to reference the object as you are accustomed to doing in VFP; therefore, if you have a method called GetCustomerInfo() on the class, you can do the following to invoke the method and capture its return value:

```
lcXML = oBizObj.GetCustomerInfo("MyCustID")
```

Note that IntelliSense should appear along the way to help you with the spelling and parameters of the method.

Exposing a .NET Component to COM

Once you are certain that your .NET class works properly, you are ready to expose it to the world of COM and access it from VFP code. It is possible to perform this step without using Visual Studio .NET; however, going this route requires the use of a couple of command-line tools called TLBEXP and REGASM. These tools are documented in the SDK for the .NET Framework if you wish to use them instead.

To expose your .NET component to COM, start by right-clicking the component’s project in the Solution Explorer. From the shortcut menu, choose Properties, which displays the property pages for the project. Note the format of this dialog, where categories are on the left and the specific property settings are on the right.

Before going any further, you should take note of one of the properties on the General section of the Common Properties group. The assembly name is going to be the first portion of the ProgID for the component, while the second part will be the name of the class within the assembly. You might be asking “What’s an assembly?” Don’t worry, there’s nothing magical about an assembly, it’s just a generic term for a .NET EXE or DLL file.

Now, under the Configuration Properties group on the left, click on the Build section (an arrow appears adjacent to the current selection). Notice the **Register for COM Interop** checkbox on the right? Simply select that checkbox and click **OK** to close the dialog.

To complete the job, press Ctrl+Shift+B or select the **Build Solution** option in the Build menu. Keep watch in the Output window, where you will see the statement “Registering project output for COM Interop...” appear after the other usual steps. At this point, you can access this class from COM, even though it has been built in .NET!

Accessing a .NET Component from COM

To test your efforts so far, start VFP. In the command window, instantiate your .NET component by using the CreateObject() function:

```
loTest = CreateObject("Classlib1.Customers")
```

If everything worked, try to invoke your method as you did from within your test project:

```
?loTest.GetCustomerInfo("MycustID")
```

You should see your return value appear on the desktop. If not, ensure that you've used all the correct names for the ProgID, method and that you passed a valid value for your parameter(s), if any.

That's all there it to it… sort of. You may be asking, "Why didn't I see any IntelliSense when I typed in this code?" There's nothing wrong with VFP if you don't see the Quick Info lists appearing as you reference the loTest variable in the above code. Instead, there is something missing from the type library that was generated by the .NET compiler.

To solve the missing IntelliSense problem, go back to your .NET component. In VB.NET, there is a special attribute that must be applied to your class in order for it to properly expose all of its members. Attributes are special directives that direct the compiler to perform many different features and, in VB.NET, attributes are delineated by angle brackets (for example, "<" and ">"). The attribute to use is the ComClass attribute. You add it to your class declaration like so:

```
<ComClass()> Public Class Customers
```

This ensures that the proper GUIDs are generated for the COM class and that the type library is properly created. Before rebuilding your assembly, be sure to release any instances of the .NET component that you may have created or you will get build errors (as it needs to overwrite the type library and the DLL). Finally, try instantiating your .NET assembly once again to see the IntelliSense in action.

Considerations

So, now that you've accomplished this technological wonder, you may be wondering "Why would I do this?" The answer is fairly straightforward: you would use this Interop feature to expose functionality from a .NET component that would be difficult or perhaps even impossible to do in a COM language.

For example, from VB or VFP, it is rather difficult to create code that knows how to interact with Windows performance counters or the Event log. In .NET, these are very easy features to add to a component, and with relatively few lines of code. In the end, which would you rather maintain: a program full of esoteric Win32 API calls and oodles of code, or a small, compact program that uses native functions?

The downside to this approach is that you need to install the .NET runtime on the machine(s) where you run this code. Perhaps sometime in the future this runtime will be included with the operating system, but that day hasn't come yet. However, even this isn't too big a problem, as VS.NET makes it relatively simple to deploy any .NET assemblies, including those exposed through COM Interop.

For those of you who are wondering, "So how did that work?" the answer is also relatively simple. Among building a type library for COM clients to consume, .NET also built a wrapper class that provides all of the "stuff" that COM needs. This wrapper class is officially called the COM Callable Wrapper, or CCW, and includes things like COM Interfaces (for example, IUnknown and IDispatch), translation of data types, HResult translation from exceptions, and so on. Therefore, even though this .NET assembly runs in the CLR, it has exposed all the necessary bits to ensure that COM can access it as well. And as a final step, when you built the component, .NET also placed the necessary details into the registry so that the component can be found by COM.

Building a COM Server

Now let's turn the tables around in this relationship and see how we can use COM components from the world of .NET. For those of you not familiar with VFP COM components, let's build a small example component that we'll eventually expose to a .NET application.

Start by firing up VFP and create a new project called *VFPCustomer*. After the project is created, expand the Code node, select the Programs node, and then click **New** to create a new program called `Customers.prg`. Create a new class in this program called Customers with the following code:

```
DEFINE CLASS Customers AS Session OLEPUBLIC
  FUNCTION GetCustomerByPK(tcCustPK as String)
    LOCAL lcRetXML, lcSQL, lcConnStr, lnConn, lnStat

    lcRetXML = ""

    IF NOT EMPTY(tcCustPK) THEN
      *create connection string to connect to local SQL Server
      lcConnStr = "Driver=SQL Server;UID=sa;Database=Northwind;Server=(local)"
      *create SQL statement to get customer record
      lcSQL = "SELECT * FROM Customers WHERE customerid = ?tcCustPK"
      *connect to SQL Server with sql pass through
      lnConn = SQLSTRINGCONNECT(lcConnStr)
      IF lnConn > 0 THEN
        *execute sql statement to retrieve data
        lnStat = SQLEXEC(lnConn,lcSQL,'SQLResult')
        IF lnStat > 0 THEN
          *set return value to XML from data set
          CURSORTOXML('SQLResult','lcRetXML')
          *clean up local cursor
          USE IN SQLResult
        ENDIF
      ENDIF
      *disconnect from SQL Server
      SQLDISCONNECT(lnConn)
    ENDIF

    RETURN lcRetXML
  ENDFUNC
ENDDEFINE
```

Note the `OLEPUBLIC` keyword at the end of the `DEFINE CLASS` command. This keyword ensures that VFP exposes this class to COM by adding a type library, exposing the necessary interfaces, and registering it on the local system. However, before trying it as a COM server, test it from the command window as a standard VFP class with the following code:

```
lo = NEWOBJECT('customers','customers.prg')
?lo.GetCustomerByPK('ALFKI')
```

You should see the returned XML appear on the VFP desktop, and IntelliSense should show you the method name as well as the parameters. However, this is expected since you are only testing it as a VFP class and not as a COM server.

Once you are satisfied with the results, build the project to create the COM server and test it from VFP as well. First, click **Build** on the project manager to display the Build Options dialog. Select the **Single-threaded COM Server (dll)** option and, optionally, select **Display Errors**. When you click **OK**, you are prompted for the name of the server. Before you accept the default value of VFPCustomer.DLL, be sure to keep an eye on the status bar of VFP, where you will see the process of how VFP builds the COM server. After clicking **Save**, a step you should see is it creating the COM server and registering it on your system.

At this point, you are ready to test the COM server from within VFP to ensure it works as advertised. Try the following code from within the command window:

```
lo = CREATEOBJECT('VFPcustomer.customers')
* or use NewObject
?lo.GetCustomerByPK('ALFKI')
```

You shouldn't notice any difference in the way this code works as compared to the previous `NewObject` version. For example, IntelliSense should show you the method name as well as the parameters for the method. If this is your first COM server, congratulations… you've succeeded!

Using COM from .NET

Now that you've gotten your COM server to work from a COM client, only a few steps separate you from being able to consume this COM server in a .NET application. Start by creating a new solution (do not add a new project to your previous solution) for a VB.NET Windows application.

Once the new project loads, add three controls to the form: a text box, a button, and a data grid control. The text box will provide the desired customer ID, while the button will retrieve the data from the COM server and populate the grid with the results.

Before writing any code, you need to reference the COM server in much the same way that you created a reference to the .NET component. Right-click on the references icon under your project in the Solution Explorer and click **Add Reference**. This time, however, click on the **COM** tab to display a list of available COM servers on your machine. There will be a pause while VS.NET reads the registry to determine what's available. Scroll down the list until you find the VFPCustomer type library. Select it and then click **Select**. Once you see the component in the bottom list, click **OK** to add it to your project.

What VS.NET just did for you is not readily apparent. Instead of just adding it to your project, .NET has done two things: first, it imported the type library so that .NET is aware of what's available within the COM server; second, it created a wrapper class, known as a Runtime Callable Wrapper (RCW), to encapsulate the COM object within a .NET assembly. The RCW takes care of all the same kinds of things that the CCW did previously, including data type conversion, reference counting, and so on.

Now that you've referenced the COM server, double-click your command button and write the following code:

```
    ' instantiate biz object
    Dim oBizObj As New vfpcustomer.CustomersClass()
    ' declare other vars
    Dim sRetXML As String
```

```
    Dim oStream As System.IO.StringReader
    ' create new dataset object
    Dim oDSResult As New DataSet()

    'call biz object to get data
    sRetXML = oBizObj.GetCustomerByPK(TextBox1.Text)
    'load return value into stream object
    oStream = New System.IO.StringReader(sRetXML)
    'read stream into data set
    oDSResult.ReadXml(oStream)
    'bind data set to grid
    DataGrid1.DataSource = oDSResult
    'force initial view to customers table
    DataGrid1.DataMember = "customers"
```

To run the form, press F5 or click **Start** from the Debug menu. As long as no errors occur, the form will execute and wait for your input. Enter “ALFKI” as a test customer ID, and then press the button. The grid should show you the customer record that matches this customer ID. Try a few other customer IDs (for example, “FISSA” or “PARIS” are two others) to see how quickly the data can be retrieved on subsequent requests.

Considerations

At this point, you’ve seen both forms of COM Interop with .NET, where .NET has played both the client as well as the server. However, let’s ask the same question as before: why would you want to use COM servers from within a .NET application? One simple answer is if you have COM servers that you didn’t write. Perhaps you purchased them from a third-party company or had them custom developed by a company that no longer supports their work. Since you do not have the source code, you may be quite hesitant to rewrite this component within .NET. Instead, as you’ve seen, you can simply register it with your .NET application and use it without modification.

The downside to using COM Interop is the overhead involved in all the translation that must occur. Every call to the COM server from .NET must be converted into using COM data types, must query the interfaces on the COM server, etc. Therefore, you are incurring unavoidable overhead in each call, which could adversely affect performance. If this is the case, consider rewriting the COM server as a .NET component and use it from within your .NET applications.

Summary

In this paper, you’ve seen what makes a COM server different from a .NET component. You’ve also seen how to build a .NET component as well as a COM server, and how to build applications that can consume either type of component. It should also be clear by now what the advantages and disadvantages are of each approach, including the alternatives of rewriting code to eliminate the Interop code.